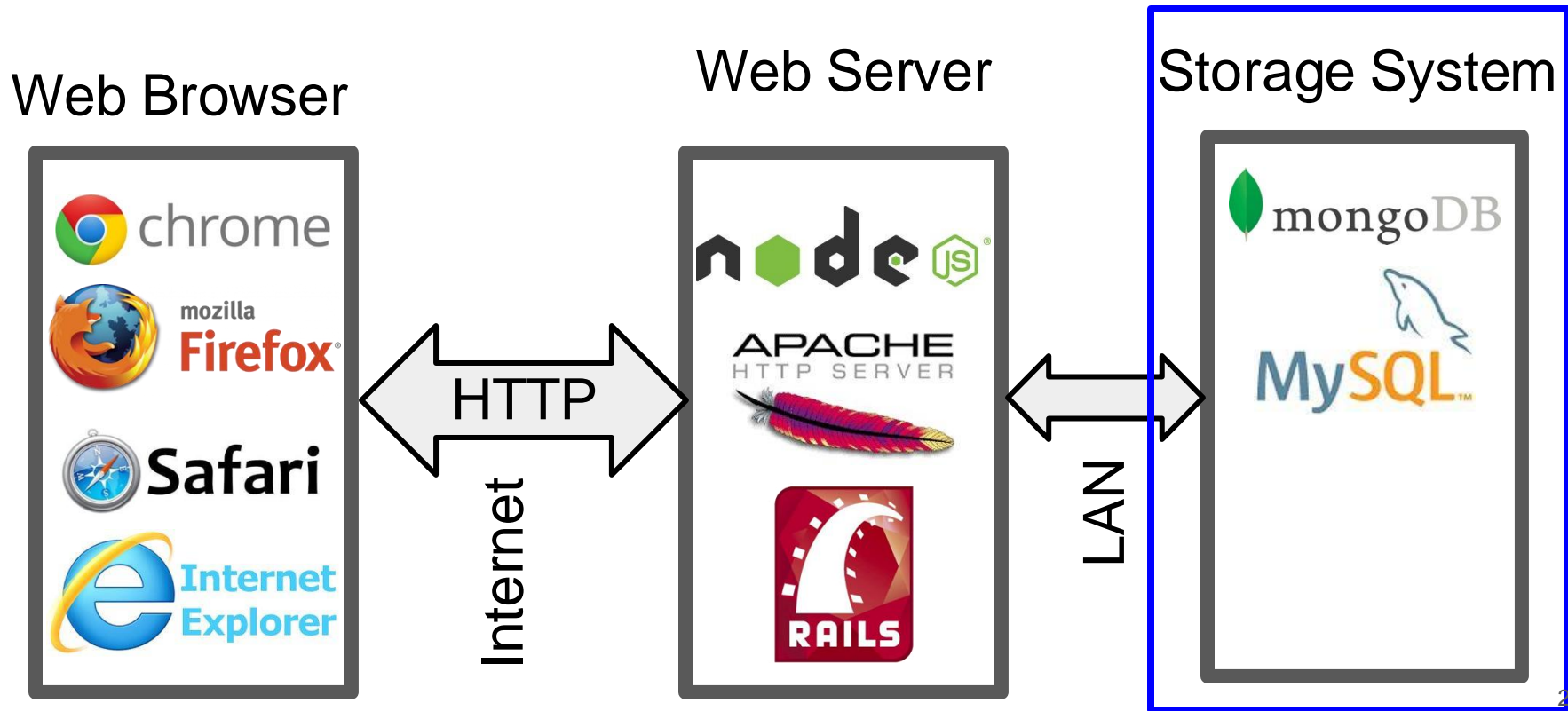


Web Databases (SQL and NoSQL)

CPEN320

Web Application Architecture



Why/when do we need a Database?

Why/when do we need a Database?

Need to handle large volumes of data:

- can't keep everything in memory
- store and search for data

Need persistent data:

- Users log in and out
- Power outages
- Network failures

Need accurate data manipulation

- e.g., Bank account transactions

Web App Storage Properties

- Always available - Fetch correct app data, store updates
 - Even if many requests come in concurrently - Scalable
 - From all over the world
 - Even if pieces fail - Reliable / fault tolerant
- Provide a good organization of storing application data
 - Quickly generate the model (data) of a view
 - Handle app evolving over time
- Good software engineering: Easy to use and reason about

Outline

- **Relational Databases (SQL-based)**
- ACID semantics
- Non-traditional Databases (NoSQL)
- MongoDB Primer

What's a Database ?

- Have you used a database? Which? Why?

What's a Database ?

- In its simplest form, it's a collection of data
 - Allows applications to modify/access data through standard interfaces
 - Separates data storage from logical organization
- Many types of databases
 - Hierarchical
 - Object oriented
 - **Relational**
 - Document-based
 - Graph-based

Relational Database

- Stores the data in the form of tables (Relations) to map one kind of data to another
- Why tables ?
 - Separate data storage from logical view of data
 - Easy to express relationships between data
 - Aggregate data from multiple tables on demand (table joins)
 - Allow declarative queries to be executed

Relational Database System

A table is made of up of **rows** (also called **tuples** or **records**) A row is made of a fixed (per table) set of typed **columns**

- String: VARCHAR(20)
- Integer: INTEGER
- Floating-point: FLOAT, DOUBLE
- Date/time: DATE, TIME, DATETIME
- Others

Example of a Table

- Much like a spreadsheet, except the columns are of fixed type and rows are identified by a unique key (known as primary key)

id	given_name	middle_name	family_name	date_of_birth	grade_point _average	start_date
1	Giles	Prentiss	Boschwick	3/31/1989	3.92	9/12/2006
2	Milletta	Zorgos	Stim	2/2/1989	3.94	9/12/2006
3	Jules	Bloss	Miller	11/20/1988	2.76	9/12/2006
4	Greva	Sortingo	James	7/14/1989	3.24	9/12/2006
...

Database Schema

Schema: The structure of the database

- The table names (e.g. User, Photo, Comments)
- The names and types of table columns
- Various optional additional information (constraints, etc.)

Column Name	Type
id	INT
given_name	VARCHAR (20)
middle_name	VARCHAR (20)
family_name	VARCHAR (20)
date_of_birth	Date
grade_point_average	Floating Point
start_date	Date

Exercise: What is the Schema of the User table below

ID	first_name	last_name	location	game
1	Ian	Malcolm	Austin, TX	2/11/2024
2	Ellen	Ripley	Nostromo	11/11/2024
3	Peregrin	Took	Gondor	14/11/2024
4	Rey	Kenobi	D'Qar	23/11/2024
5	April	Ludgate	Awnee, IN	8/12/2024
6	John	Ousterhout	Stanford, CA	21/12/2024

Schema of User table

Column types

ID - INT
first_name - VARCHAR(20)
last_name - VARCHAR(20)
location - VARCHAR(100)
game - DATE

ID	first_name	last_name	location	game
1	Ian	Malcolm	Austin, TX	2/11/2024
2	Ellen	Ripley	Nostromo	11/11/2024
3	Peregrin	Took	Gondor	14/11/2024
4	Rey	Kenobi	D'Qar	23/11/2024
5	April	Ludgate	Awnee, IN	8/12/2024
6	John	Ousterhout	Stanford, CA	21/12/2024

Multiple Unconnected Tables

id	given_name	middle_name	family_name	date_of_birth	grade_point_average	start_date
1	Giles	Prentiss	Boschwick	3/31/1989	3.92	9/12/2006
2	Milletta	Zorgos	Stim	2/2/1989	3.94	9/12/2006
3	Jules	Bloss	Miller	11/20/1988	2.76	9/12/2006
4	Greva	Sortingo	James	7/14/1989	3.24	9/12/2006
...

id	username	password_hash	role
763	Demetrius	ASVUQP8AZV8	administrator
845	Sharon	8WEROCPA387	class_admin
973	Wilmer	S3D03VP3A8AS	class_admin
1021	Nicolai	SDF83NC9A2F2J	data_analyst


Connected Tables

- The problem with having multiple unconnected tables is that it's difficult to tell if the same record is present in both tables
 - **Solution 1 (Ugly):** Duplicate the relevant data in each table. Complicates data management, updates and need to anticipate queries in advance
 - **Solution 2 (Preferred):** Keep a pointer (foreign key) to the other table so that you can access the data by following the pointer. No need to anticipate queries in advance, easy to modify

Connected Tables

id	Award	Year	Student_id
1493	Best Handwriting	2007	1
1657	Nicest Smile	2007	3
1831	Cleanest Desk	2007	3
1892	Most likely to win the lottery	2008	4

id	given_name	middle_name	family_name	date_of_birth	grade_point_average	start_date
1	Giles	Prentiss	Boschwick	3/31/1989	3.92	9/12/2006
2	Milletta	Zorgos	Stim	2/2/1989	3.94	9/12/2006
3	Jules	Bloss	Miller	11/20/1988	2.76	9/12/2006
4	Greva	Sortingo	James	7/14/1989	3.24	9/12/2006
...



Each table has what is known as **primary key** to uniquely identify records in it.

Tables keep **foreign keys** to link to records in other tables. A foreign key is the primary key of the table being linked to.

Structured Query Language (SQL)

- Standard language for accessing relational data
 - Sweet theory behind it: **relational algebra**
- Queries: the strength of relational databases
 - Lots of ways to extract information
 - You specify what you want
 - The database system figures out how to get it efficiently
 - Refer to data by contents, not just name

SQL Example Commands

```
CREATE TABLE Users (  
    id INT AUTO_INCREMENT,  
    first_name VARCHAR(20),  
    last_name VARCHAR(20),  
    location VARCHAR(100));
```

```
INSERT INTO Users (  
    first_name,  
    last_name,  
    location)  
VALUES  
('Ian',  
'Malcolm',  
'Austin, TX');
```

```
DELETE FROM Users WHERE  
    last_name='Malcolm';
```

```
UPDATE Users  
    SET location = 'New York, NY'  
    WHERE id = 2;
```

```
SELECT * FROM Users;
```

```
SELECT * from Users WHERE id = 2;
```

Keys and Indexes

Consider a model fetch: `SELECT * FROM Users WHERE id = 2`

Database could implement this by:

1. **Scan** the Users table and return all rows with id=2
2. Have built an **index** that maps id numbers to table rows. Lookup result from index.

Uses **keys** to tell database that building an index would be a good idea

Primary key: Organize data around accesses

`PRIMARY KEY(id) on CREATE table`

Secondary key: Other indexes (UNIQUE)

```
CREATE TABLE Persons (  
  ID INT PRIMARY KEY AUTO_INCREMENT,  
  first_name VARCHAR(50) NOT NULL,  
  last_name VARCHAR(50) NOT NULL,  
  location VARCHAR(100),  
  UNIQUE (first_name, last_name)  
);
```

Table Joins

- Can be used to combine information from multiple tables together (e.g., through SQL)
 - Produces a single table containing the information in both tables, without **duplication**
 - Joins can involve more than one table
- For example, we can produce a single join table having the award name and the student details

id	Award	Year	Student_id
1493	Best Handwriting	2007	1
1657	Nicest Smile	2007	3
1831	Cleanest Desk	2007	3
1892	Most likely to win the lottery	2008	4

id	given_name	middle_name	family_name	date_of_birth	grade_point_average	start_date
1	Giles	Prentiss	Boschwick	3/31/1989	3.92	9/12/2006
2	Milletta	Zorgos	Stim	2/2/1989	3.94	9/12/2006
3	Jules	Bloss	Miller	11/20/1988	2.76	9/12/2006
4	Greva	Sortingo	James	7/14/1989	3.24	9/12/2006
...

Example of a Join in SQL

- `SELECT * from Employees, Departments
where employee.deptID=department.deptID`

Employee table		Department table	
LastName	DepartmentID	DepartmentID	DepartmentName
Rafferty	31	31	Sales
Jones	33	33	Engineering
Heisenberg	33	34	Clerical
Robinson	34	35	Marketing
Smith	34		
Williams	NULL		

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Robinson	34	Clerical	34
Jones	33	Engineering	33
Smith	34	Clerical	34
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31

The problem with Joins

- Joins are expensive as they need to analyze multiple tables (potentially stored elsewhere)
- Join performance is poor for large tables, though databases are very good at optimizing
- Requires all the tables to be available during join - otherwise join will fail

Outline

- Relational Databases (SQL-based)
- **ACID semantics**
- Non-traditional Databases (NoSQL)
- MongoDB Primer

SQL Databases have ACID Semantics

What does ACID stand for?

SQL Databases have ACID Semantics

ATOMICITY *all or nothing*

CONSISTENCY *written data follows rules and constraints*

ISOLATION *uncommitted transactions are isolated from each other*

DURABILITY *committed transactions are permanent*

Atomicity of Transactions

- Transaction is a sequence of operations executed all at once or not at all (**Atomicity**)
- If failure occurs, roll-back to the beginning
- **Example:** Transfer \$1000 from Acct. A to B
 - Step 1: Locate Account A and check balance
 - Step 2: Subtract 1000 dollars from Acct A
 - Step 3: Credit 1000 dollars to Acct B
 - ERROR -> ROLL BACK steps 3, 2 (and 1 if needed)

Consistency

- Can check one or more constraints on the resulting data, and abort if not satisfied

Example:

- Check status of bank account of A, **Before** transaction and **After**
- Does A have enough funds before transaction?
- Is the funds reduced after the transaction?

Isolation

- Transactions are executed **independently** from one another, preventing concurrent transactions from interfering.

T1 subtracts 10 from A

T2 subtracts 10 from B

T2 adds 10 to A

T1 adds 10 to B

Isolation

- Transactions are executed **independently** from one another, preventing concurrent transactions from interfering.

T1 subtracts 10 from A

T1 adds 10 to B

T2 subtracts 10 from B

T2 adds 10 to A

Durability

- Transactions are permanent when committed

T1 subtracts 10 from A

T1 adds 10 to B

T2 subtracts 10 from B

T2 adds 10 to A

ACID: Pros and Cons

- **Pros**

- Simplifies reasoning about actions of the system
- Guarantees correctness in presence of failures

- **Cons**

- Guarantees come with huge performance cost
- Cannot guarantee availability when network fails
 - This is due to something called the CAP theorem

Class Activity

- Consider the following transactions T1 and T2 which execute on a bank account database. Which of the four ACID rules, if any, (Atomicity, Consistency, Isolation, Durability) are violated ?
- Assume initial balance is \$100. T1 attempts to deposit \$900 to the account. At the same time, T2 checks if the account balance ≥ 500 and returns true. However, T1 aborts and the account balance becomes \$100 again.

Solution

Isolation

The scenario suggests a violation of the **isolation** property. T2 observes the intermediate state of the database (the account balance as \$1000 due to the uncommitted deposit by T1) and acts upon this information. When T1 aborts, the decision made by T2 (that the account balance was \geq \$500) is based on a transient state that no longer exists.

This behavior indicates that the transactions are not properly isolated from each other, as T2 should not be able to see the uncommitted changes made by T1.

Outline

- Relational Databases (SQL-based)
- ACID semantics
- **Non-traditional Databases (NoSQL)**
- MongoDB Primer

What are NoSQL Databases?

What are NoSQL Databases?

- designed to handle **unstructured or semi-structured data**.
- Offer flexible schemas, scalability, and are optimized for large-scale data processing
- Often used in real-time web applications
- include types like document, key-value, and graph databases.

NoSQL Databases

- Do not natively support Table joins
 - Are much more scalable and failure tolerant
 - Must do joins explicitly using program code
- Do not typically support ACID semantics
 - So data may be inconsistent or out of sync (provide what is known as eventual consistency)
 - When failures occur, data may be lost or incorrect

CAP Theorem [Brewer'99]

- You can achieve only two of the following three properties in any database system

CONSISTENCY “...requiring requests of the distributed shared memory to act as if they were executing on a single node, responding one at a time”

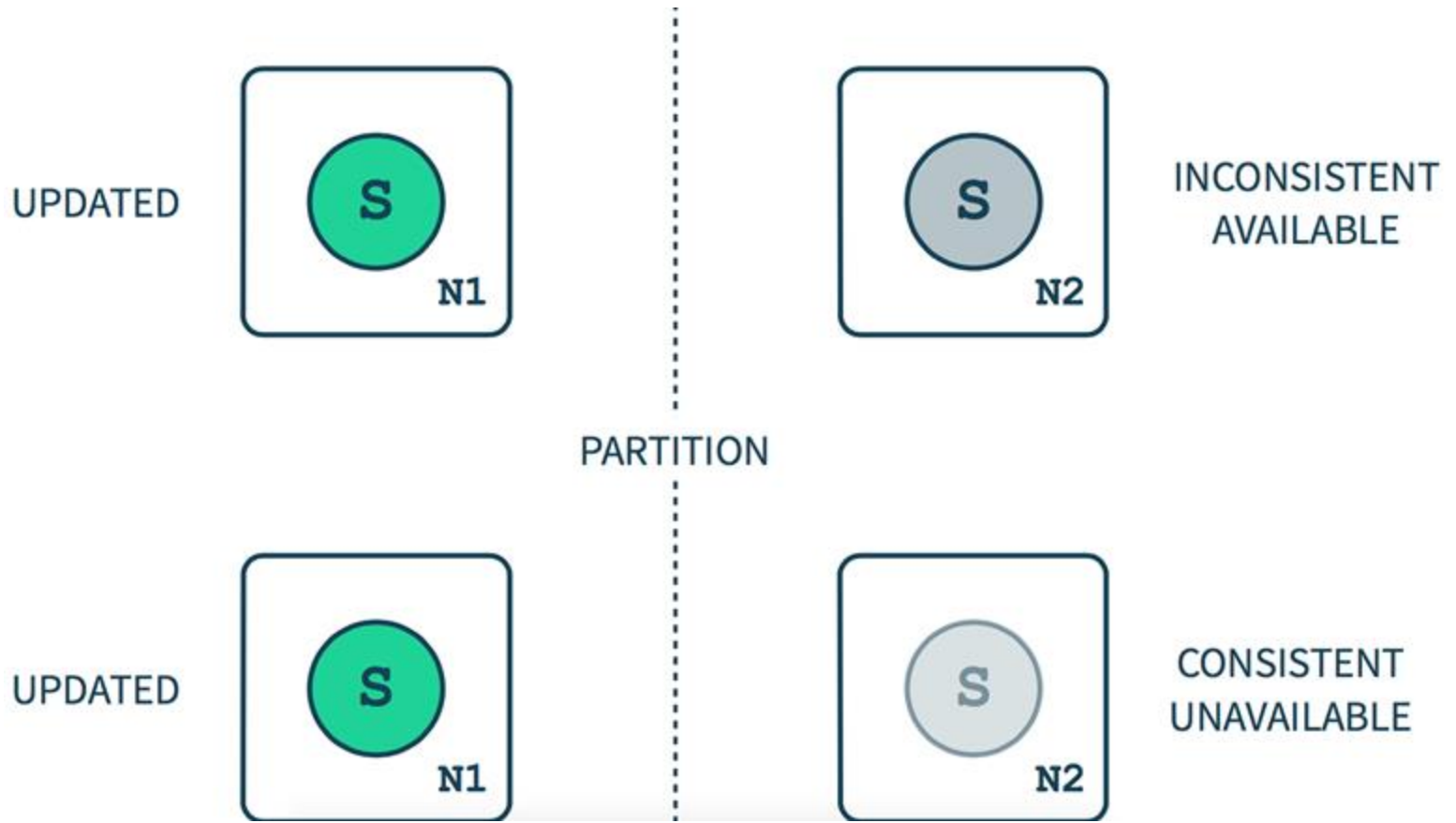
AVAILABILITY “... every request received by a non-failing node in the system must result in a response”

PARTITION TOLERANCE “... the network will be allowed to lose arbitrarily many messages sent from one node to another”

CAP theorem

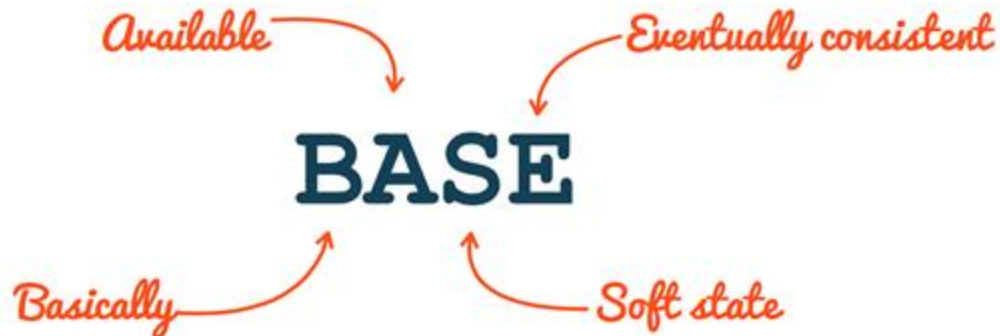
- During a network partition (failure), a system must choose either **consistency** or **availability** for it to work through the partition
 - Traditional **SQL-based** databases choose **consistency** and may hence not be available
 - **NoSQL** databases choose **availability** and hence may not be consistent
 - In web applications, availability often trumps consistency

Example of Network Partitioning



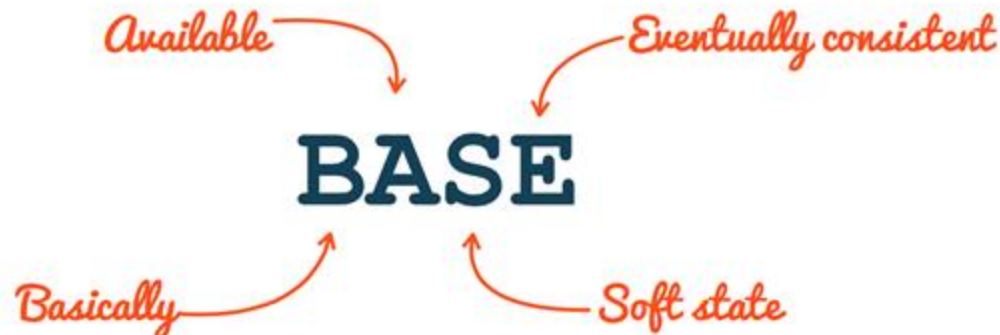
BASE principle

- **B**asically **A**vailable
- **S**oft state
- **E**ventually consistent



Eventual Consistency

- NoSQL databases provide a guarantee that they will eventually be consistent (e.g., when the network partition heals)
 - Eventually can be a very long time
 - Consistent does not mean correct....



SQL Vs NoSQL - 1

SQL

NoSQL

TYPES

one type

key-value,
document, graph

EXAMPLES

MySQL, SQLite,
Oracle Database

MongoDB,
Cassandra, HBase,
Neo4j

SQL Vs. NoSQL - 2

DATA STORAGE MODEL

SQL

Individual records are stored as rows; columns store a specific piece of data about record

Separate data types are stored in separate tables and joined together when complex queries are executed

NoSQL

Key-value stores are similar to SQL, but have only two columns

Document DBs store all relevant data together in a single document in a hierarchically nested format (JSON, XML)

www.mongodb.com/nosql-explained

SQL Vs. NoSQL - 3

SQL

NoSQL

SCHEMAS

Structure and data types are fixed in advance

Unlike SQL rows, dissimilar data can be stored together as necessary

SCALING

Vertically: single server must be made increasingly powerful

Horizontally: distribute data over several machines

SQL Vs. NoSQL - 4

	SQL	NoSQL
SUPPORTS TRANSACTIONS	Yes	In certain circumstances and at certain levels (document-level)
CONSISTENCY	Strong consistency	Tunable consistency (MongoDB), Eventual consistency (Cassandra)

Class Activity

- For each of the following scenarios, will you use a traditional database or no-SQL database. Justify your answer using CAP theorem:
 - Online photo gallery to browse photos and upload photos occasionally from multiple locations
 - Large ecommerce store in which the inventory needs to reflect any purchases made instantly in all locations
 - Shopping cart of customers in an online store in which users can login from different locations

Class Activity

- Online photo gallery to browse photos and upload photos occasionally from multiple locations
 - No-SQL
 - **Availability** and **Partition Tolerance** are important
- Large ecommerce store in which the inventory needs to reflect any purchases made instantly in all locations
 - SQL
 - **Consistency** and **Partition Tolerance** are essential
- Shopping cart of customers in an online store in which users can login from different locations
 - No-SQL?
 - **Availability** and **Partition Tolerance? Or Consistency?**

Outline

- Relational Databases (SQL-based)
- ACID semantics
- Non-traditional Databases (NoSQL)
- **MongoDB Primer**

MongoDB

- **Document-oriented NoSQL database**
 - Documents are the equivalent of tables
 - Stored in JSON format (technically BSON, or binary JSON)
 - Must be smaller than 16 MB in size
- **No apriori schema needed, or rather schema can be modified dynamically**
 - Can store dissimilar objects in same document
 - Documents can be embedded/nested in other documents

MongoDB: Data types

JSON: null, boolean, number, string, array, and object

MongoDB: null, boolean, number, string, array, **date, regex, embedded document, object id, binary data, code**

MongoDB: Example Dataset

```
{
  "address": {
    "building": "1007",
    "coord": [ -73.856077, 40.848447 ],
    "street": "Morris Park Ave",
    "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
    { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
    { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
    { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
    { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
    { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

Databases and Collections

- A MongoDB database consists of multiple databases. Specify db to use by “use test”
- A database can have multiple collections. Specify collection as `db.collectionName.op`
- A collection can have one or more documents
 - Each record is called a document

Insert into a Database

- `db.collectName.insert(document in JSON)`

```
db.restaurants.insert(  
  {  
    "address" : {  
      "street" : "2 Avenue",  
      "zipcode" : "10075",  
      "building" : "1480",  
      "coord" : [ -73.9557413, 40.7720266 ],  
    },  
    "borough" : "Manhattan",  
    "cuisine" : "Italian",  
    "grades" : [  
      {  
        "date" : ISODate("2014-10-01T00:00:00Z"),  
        "grade" : "A",  
        "score" : 11  
      },  
      {  
        "date" : ISODate("2014-01-16T00:00:00Z"),  
        "grade" : "B",  
        "score" : 17  
      }  
    ],  
    "name" : "Vella",  
    "restaurant_id" : "41704620"  
  }  
)
```

Insert

```
// Insert a single document
db.myCollection.insertOne({
  name: "John Doe",
  age: 30,
  city: "New York"
});
```

```
// Insert multiple documents
db.myCollection.insertMany([
  { name: "Jane Doe", age: 25, city: "Los Angeles" },
  { name: "Richard Roe", age: 35, city: "Chicago" }
]);
```


Finding objects

- `db.collectName.find()` – shows all documents
- `db.collectName.find(JSON object)` – shows documents satisfying the given JSON object
 - Finds all docs with the fields and values equal to the JSON object passed as an argument
 - Can also specify conditional operations such as `$lt`, `$gt`, or logical combinations (using AND, OR)

Examples of queries

- `db.restaurants.find({"cuisine": "Italian"})`
 - Finds all restaurants with the cuisine==Italian
- `db.restaurants.find({ "grades.score":
{ $gt:30} })`

Object_id

- Every document is given a unique ‘_id’ value – automatically assigned by the MongoDB
- Object IDs must be unique in a document, and should be of type ObjectId
- Can be used to remove or update specific objects

Update

- `db.collectName.update(objects to be matched, object fields to be updated)`

```
db.restaurants.update(  
  { "name" : "Juni" },  
  {  
    $set: { "cuisine": "American (New)" },  
    $currentDate: { "lastModified": true }  
  }  
)
```



Update operator (full list of operators can be found at:
<https://docs.mongodb.org/manual/reference/operator/update/>)

Remove

- Can remove documents from a collection using the remove method

`db.collectName.remove(matching condition)`

example:

```
db.restaurants.remove({ "cuisine":  
                        "Italian" })
```

Operations on each record

Example: Print the grades of all restaurants that have more than one grade associated with them.

```
db.restaurants.find().forEach(  
    function(Object) {  
        if (Object.grades.length > 1)  
            printjson(Object.grades);  
    }  
)
```

Table Joins in MongoDB

- Joins are not natively supported in MongoDB and hence need to be written manually
 - Iterate over each document of the first collection
 - Lookup the corresponding document in the second collection either by key or by name
 - Write JavaScript code to merge the information in the relevant fields from the two documents
 - Return the merged information as the query result

Example: Join Operation

- Assume that you had another collection in the database called “users” which had a list of users who had reviewed each restaurant. Assume this collection is indexed by restaurant name.
- We wish to write a query to list all the restaurants that have at least one review, and the list of users who reviewed that restaurant.

Example Join Operation

```
db.restaurants.find().forEach(  
  function(Object) {  
    if (Object.grades.length > 1) {  
      var user = db.Users.find(Object.name);  
      if (user!=null) {  
        printjson(Object.name);  
        printjson(user);  
      }  
    }  
  }  
)
```

Class Activity

- You have two collections in a MongoDB database. *marks* contains the list of students in a course with their marks and student number, and *students* contains the student number along with details such as first name, last name etc.
- **Compute** the join of these two collections (in JS code) from the Mongodb shell to print each student along with their marks and details.
- You can assume the database is already loaded into the shell.

Solution to the activity

```
db.marks.find().forEach(  
  function(Object) {  
    var st = db.students.find( {"student no":  
      Object.studentNo} );  
    if (st!=null) {  
      printjson(st);  
      printjson(Object.marks);  
    }  
    else {  
      print("No match found for " + Object.studentno);  
    }  
  }  
)
```