Web Databases (SQL and NoSQL)

Lecture 11 (CPEN 400A)

Karthik Pattabiraman

Based on CS498RK at UIUC (used with permission),
and the MongoDB tutorial (docs.mongodb.)

Outline

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

What's a Database ?

- In its simplest form, it's a collection of data
  - Allows applications to modify/access data through standard interfaces
  - Separate data storage from logical organization

- Many types of databases
  - Hierarchical
  - Object oriented
  - **Relational**
  - Document-based

→ Most popular (1970s – current)
- Modelled mathematically
- Many optimizations modelled mathematically
- used by many companies, oracle.

↳ NoSQL

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

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
…	…	…	…	…	…	…

↳ unique key

- Rigid col^n types, exact format & types

Source:
http://archive.oreilly.com/pub/a/ruby/excerpts/ruby-learning-rails/intro-ruby-relational-db.html

Database schema

- A logical representation of the tables' structure listing each column name and type

Column Name	Type
id	Integer
given_name	String
middle_name	String
family_name	String
date_of_birth	Date
grade_point_average	Floating Point
start_date	Date

- No SQL do not necessarily have schema.

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
…	…	…	…	…	…	…

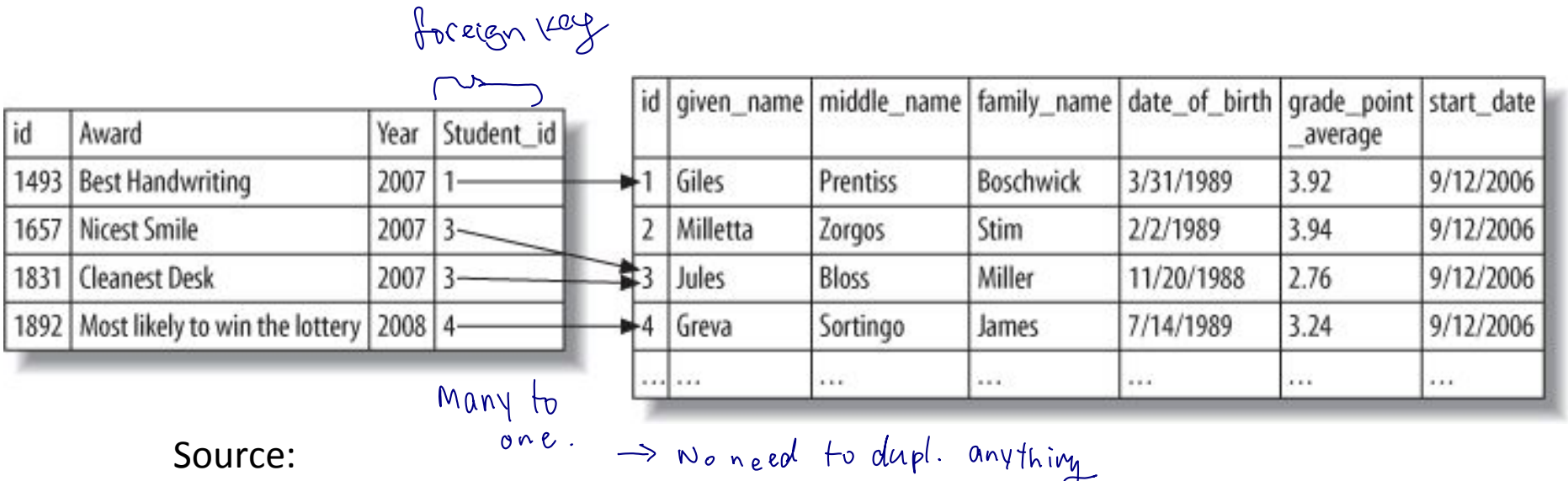
id	username	password_hash	role
763	Demetrius	ASVUQP8AZV8	administrator
845	Sharon	8WEROCPA387	class_admin
973	Wilmer	S3DO3VP3A8AS	class_admin
1021	Nicolai	SDF83NC9A2F2J	data_analyst

Source:
http://archive.oreilly.com/pub/a/ruby/excerpts/ruby-learning-rails/intro-ruby-relational-db.html

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
…	…	…	…	…	…	…

Source:
http://archive.oreilly.com/pub/a/ruby/excerpts/ruby-learning-rails/intro-ruby-relational-db.html

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.

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 from the previous slide

Example of a Join in SQL

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

Employee table

LastName	DepartmentID
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

↓

list everything together.

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 straddle multiple tables (potentially stored elsewhere)
- Combination of fields from different tables can result in losing cache locality
- 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 (more later)

Outline

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

SQL Databases have ACID Semantics

↳ satisfy all properties

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 and Transactions

- Transaction is a sequence of operations which are executed all at once or not at all **(Atomicity)** → cannot have partial completion
- If failures occur, roll-back to the beginning
- **Example**: Transfer $1000 from Accts. 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

wrap in 1 transaction.

- All parts needed to complete
- roll transaction back if fail, commit if complete

Consistency

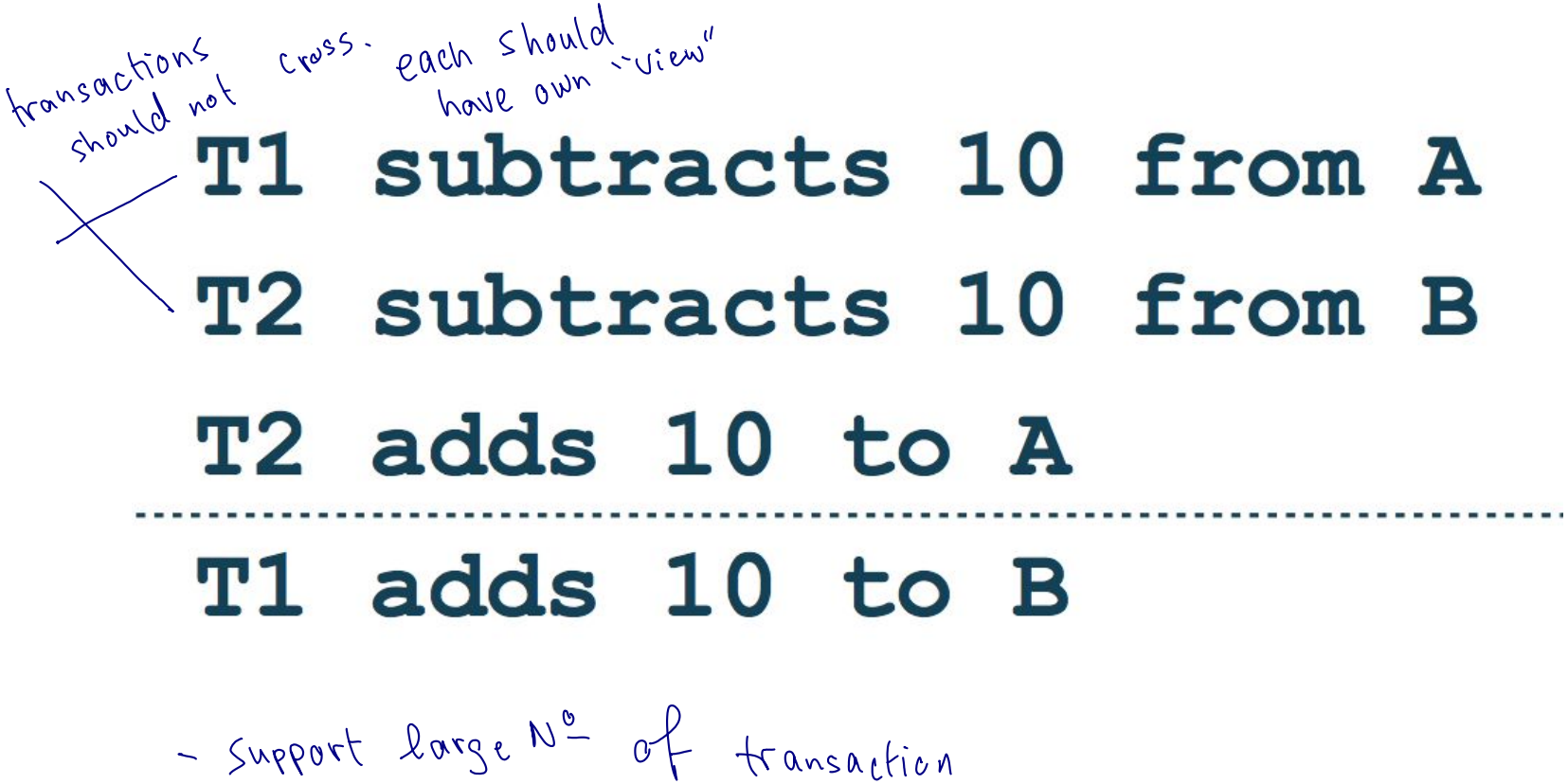
- Can check one or more constraints on the resulting data, and abort if not satisfied

```
CREATE TABLE acidtest (
A INTEGER, B INTEGER,
CHECK (A + B = 100));
```

- invariants should be maintained
- checks written by programmer, rollback if fail.

Isolation

- Transactions are isolated from one another



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
```

- from commit, cannot change record -
- if non permanent, cannot make progress

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.

Outline

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

→ "Not SQL" original
↓↓ now
Not only SQL
Can support OO model2

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

if you wait long enough, it might be consistent

- Failure tolerant, but

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"* - Cannot have two req. return inconsistent val.

AVAILABILITY *"… every request received by a non-failing node in the system must result in a response"* - reasonable response time.

Pick 1 of 2

PARTITION TOLERANCE *"… the network will be allowed to lose arbitrarily many messages sent from one node to another"*

↳ Network partition ⇒ nodes cannot comm. → 1. either respond
2. wait.

Not option.
⇒ Reality

CAP theorem continued..

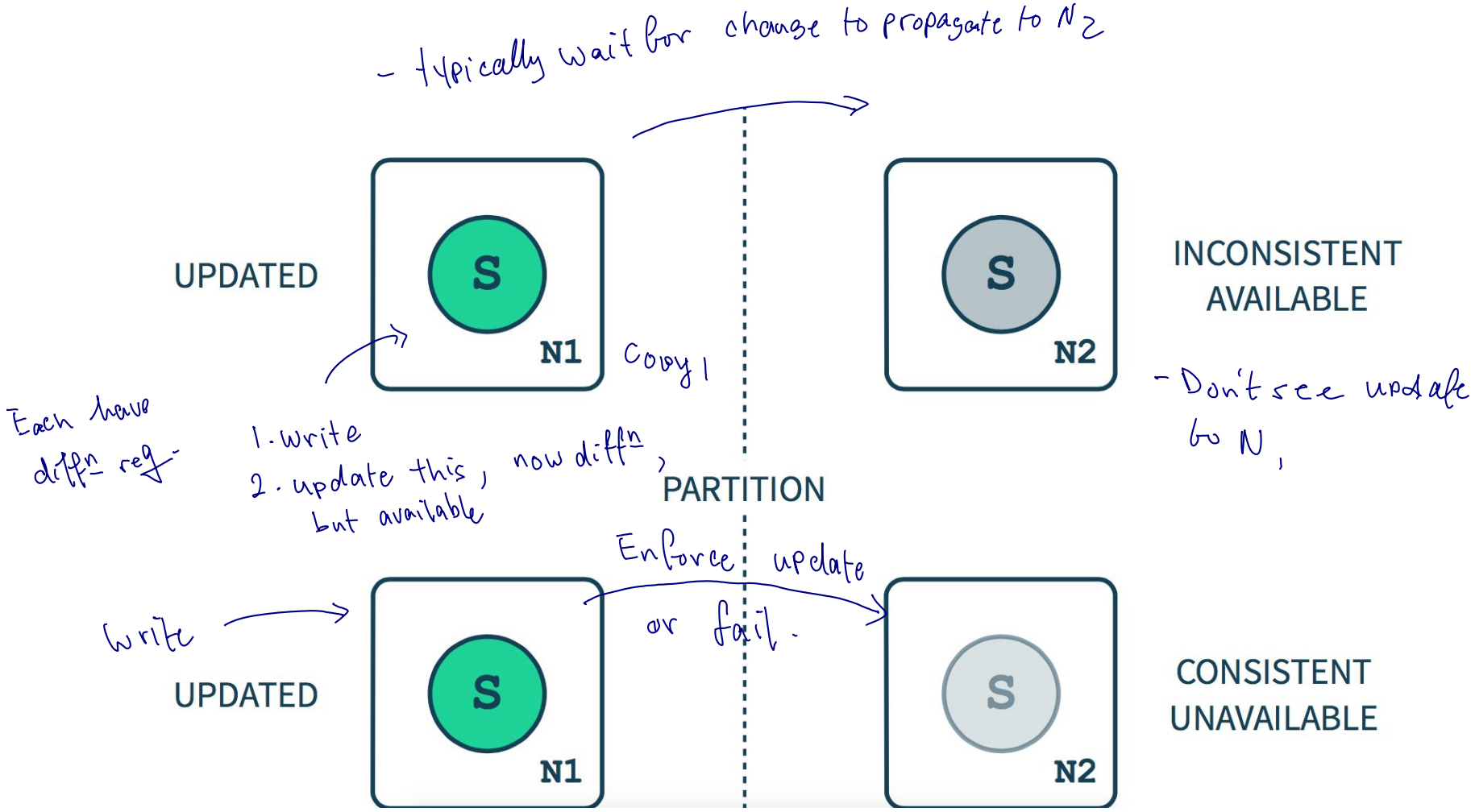
- During a network partition, a system must choose either consistency or availability for it to work through the partition
  - Traditional SQL-based databases choose consistency and may hence not be available
  - NoSQL databases choose availability and hence may not be consistent
  - In web applications,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

↳ e.g. facebook feed.
- doesn't matter if it is not the same

bank
- consistency
- do not care if ATM out of serv.

- only applies to write + read. w/ read, never inconsistent.

Example of Network Partitioning



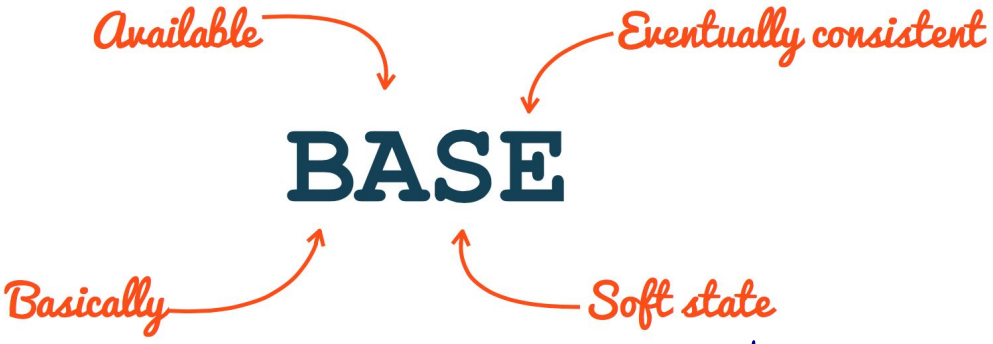
Eventual Consistency

- 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….

→ when partition heals.

→ no guarantee

- things happen in between compromising correctness.



- cannot guarantee state, best effort.

SQL Vs NoSQL - 1

	SQL	NoSQL
TYPES	one type - relational, typically	key-value, document, graph
EXAMPLES	MySQL, SQLite, Oracle Database	MongoDB, Cassandra, HBase, Neo4j

SQL Vs. NoSQL - 2

Glorified hashtable

	SQL	NoSQL
DATA STORAGE MODEL	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	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 - skip schema, since it is only for table join.
SCALING	Vertically: single server must be made increasingly powerful - run multiple close, lockstep	Horizontally: distribute data over several machines - data center model.

SQL Vs. NoSQL - 4

	SQL	NoSQL
SUPPORTS TRANSACTIONS	Yes	In certain circumstances and at certain levels (document-level) → transaction is state of all. single record consistent.
CONSISTENCY	Strong consistency	Tunable consistency (MongoDB), Eventual consistency (Cassandra)

You can set consistency level

Class Activity

- For each of the following scenarios, will you use a traditional database or non-SQL database. Justify your answer using CAP thrm.
  - 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

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
  - other weird restrictions
- **No apriori schema needed, or rather schema can be modified dynamically**
  - Can store dissimilar objects in same document
  - Documents can be embedded in other documents

- MongoDB supports JS. natively, pass JSON back & forth

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”

  → Name of DB.

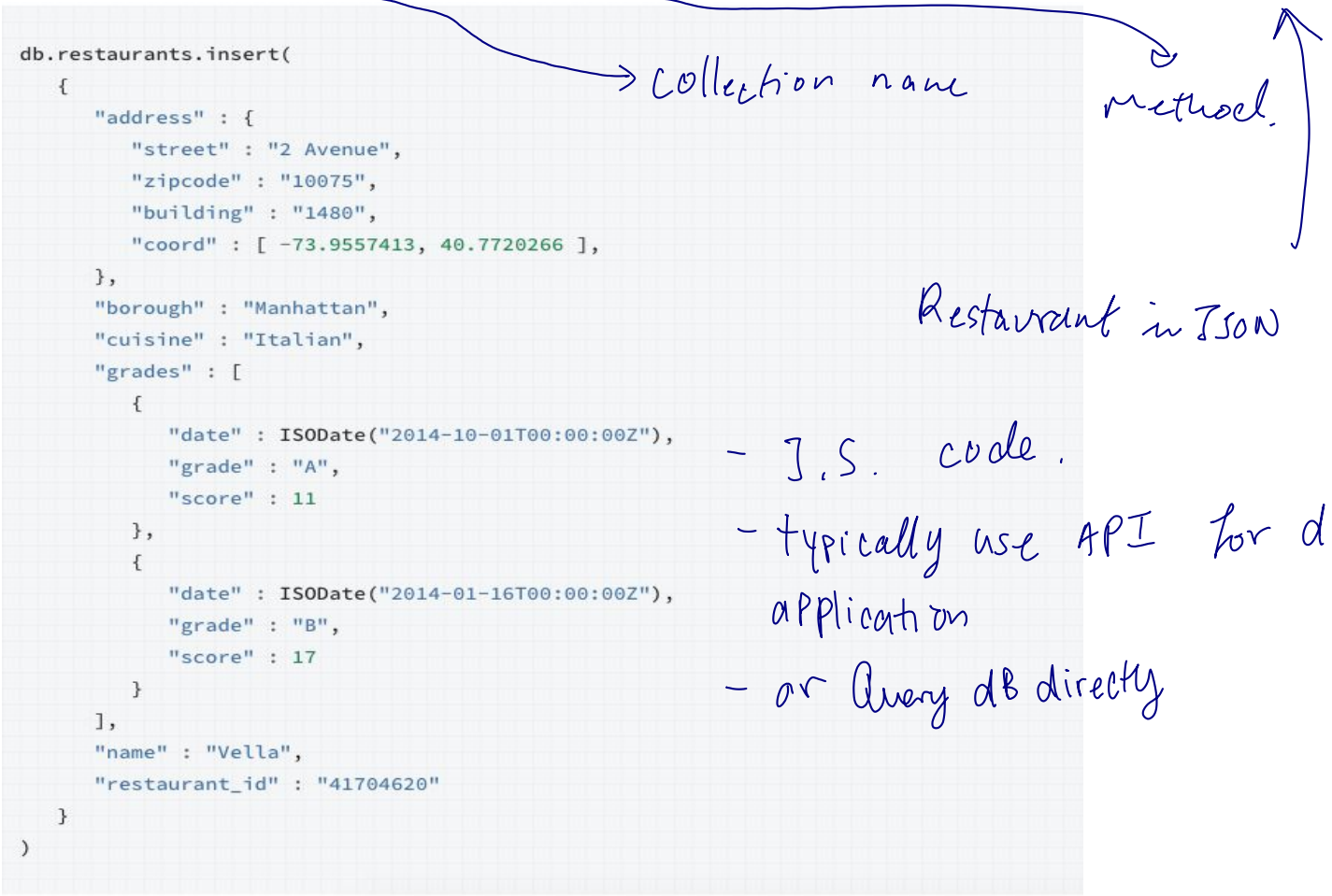
- A database can have multiple collections. Specify collection as db.collectionName.op

  → operator.

- A collection can have one or more documents
  - Each record is called a document

Insert into a Database

- db.collectName.insert(document in JSON)



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({“borough”: “Manhattan”})
  - Finds all restaurants with the borough==manhattan

JSON obj.

- db.restaurants.find({ “grades.score”: { $gt:30} })

find restaurants w/ grade.score greater than 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 }
    }
)
```

match

update operator

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({ “borough”: “Manhattan” })

– remove restaurants in Manhattan

Operations on each record

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

- iter-e thru

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

-JS direct exec on dB.

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

- No free lunch - → you pay cost, you decide how to deal w/ further.

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. How will you compute the join of these two collections (in JS code) from the Mongdb shell to list the student details along with the marks. 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);
            }
    }
)
```

Outline

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