To SQL and to NoSQL

CS252

Last week

- We reviewed how to work with SQL
- We saw how we can process SQL queries programmatically using PHP
- We experimented with a sample employees database
  - Wednesday batch had a lot of trouble
  - Monday batch goes Saturday, 8th from 1400-1700
    - Be warned, do your prep reading
- Today
  - Project init
  - Finish up with SQL functionalities
  - SQL security issues
  - Start with NoSQL

Project init

- Worth 20% of the course grade
- Deadlines:
  - Project description: 28th Sept
  - Mid-project review: 21st Oct
  - Completion: 7th Nov
- Ideas
  - No frills transport locator (*****)
  - Crowdsourced train tracking (****+* for integrating micropayments)
  - HTML5/JavaScript game with session management (***+** for RTS)
  - Voice-activated expense tracker (***+* for non-API voice-activation)
  - Resource booking facility (**+* for consistency guarantee)

SQL joins

- Inner joins
  - Return only matching rows
  - Restricted to 256 at the same time
- Outer joins
  - Return all matching rows plus all non-matching rows from at least one of two tables
  - Restricted to only two tables at a time

Inner join

Table One

X	A
1	a
4	d
2	b

Table Two

X	B
2	x
3	y
5	v

```
select *
   from one, two
   where one.x=two.x;
```

X	A	X	B
2	b	2	x

General inner join syntax

SELECT *column-1*<, …*column-n*>
FROM *table-1*|*view-1*<, … *table-n*|*view-n*>
WHERE *join-condition(s)*
<**AND** *other subsetting conditions*>
<*other clauses*>**;**

Left (outer) join

Table One

X	A
1	a
4	d
2	b

Table Two

X	B
2	x
3	y
5	v

```
select *
   from one left join two
      on one.x = two.x;
```

X	A	X	B
1	a	.	
2	b	2	x
4	d	.	

General outer join syntax

SELECT *column-1* <, …*column-n*>
FROM *table-1*
LEFT|RIGHT|FULL JOIN
table-2
ON *join-condition(s)*
<*other clauses*>**;**

SQL security

- Inner security
  - Password protection
  - Access privilege system
- Outer security
  - SQL injection attacks
  - Prevention

Inner security

- The database is the critical vulnerability in your web service
  - Lower elements in the stack are patched by large organizations
  - Higher elements in the stack are static webpages which can't be compromised
    - Wrong!
    - We'll see more about this when we talk about XSS for JavaScript
- First rule of database security
  - For God's sake, password protect your SQL server
  - Does *mysql -u root* work for you?

Access privileges in SQL

- Root account has all privileges
- Other accounts can be granted privileges selectively
- Three types of privileges
  - Admin
  - Database
  - Object
- Use SHOW GRANTS to see which accounts have which privileges on your SQL server
- Grant privileges defensively

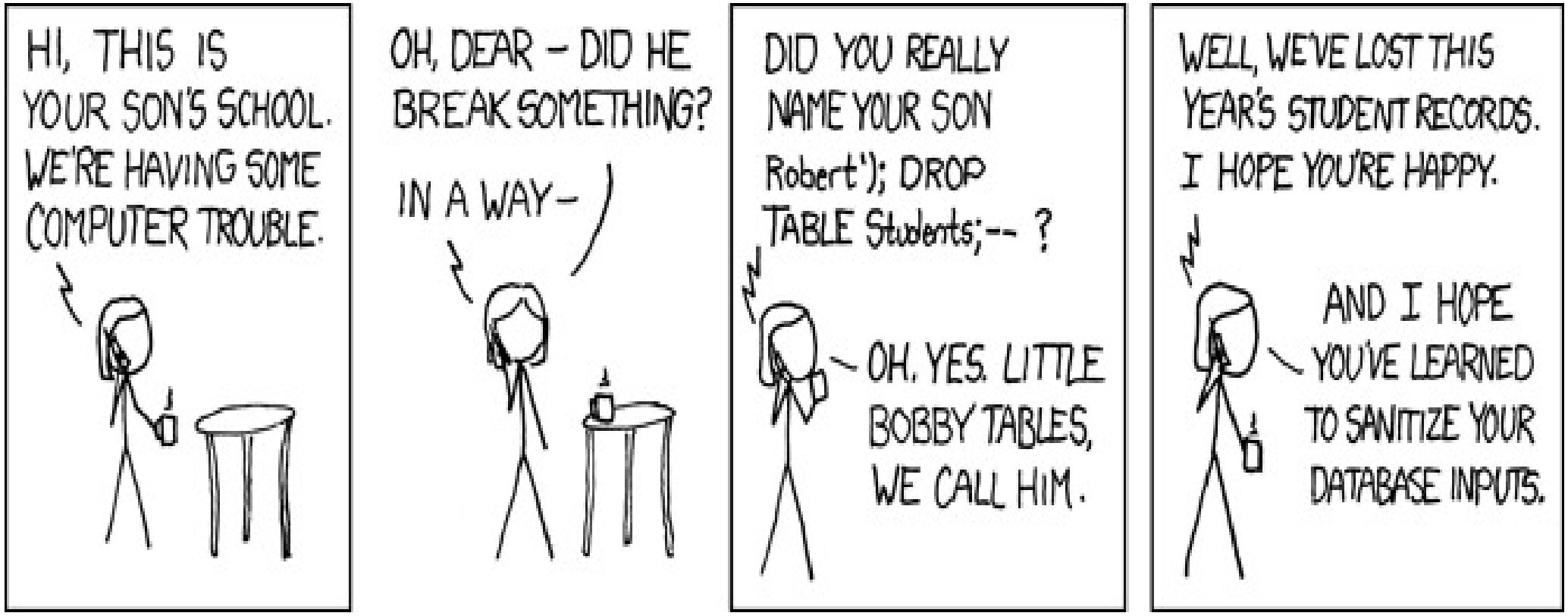
Who should have these privileges?

- ALL
- ALTER *
- CREATE
- DROP
- EVENT
- FILE *
- INSERT
- SELECT
- SHOW DATABASES

Outer security

- Basics
  - Put the server behind a firewall
  - Block port to external access
- Transmit data using SSL/SSH
- Don't trust user inputs
  - The story of young Bobby Tables

Bobby Tables



From xkcd

A typical SQL/PHP script

```
$sql = "SELECT id, firstname, lastname FROM someTable";
$result = $conn->query($sql);

if ($result->num_rows > 0) {
    // output data of each row
    while($row = $result->fetch_assoc()) {
        echo "id: " . $row["id"]. " - Name: " . $row["firstname"]. " " . $row["lastname"]. "<br>";
    }
```

Misplaced trust – a case study

- SELECT * FROM Students WHERE Name = 'Bobby Tables';

Misplaced trust – a case study

- SELECT * FROM Students WHERE Name = 'Bobby Tables';
- $sql = “SELECT * FROM Students WHERE Name =”. <user_input>.”;”

Misplaced trust – a case study

- SELECT * FROM Students WHERE Name = 'Bobby Tables';
- $sql = “SELECT * FROM Students WHERE Name =”. <user_input>.”;”
- SELECT * FROM Students WHERE Name = 'Robert'; DROP TABLE Students;--

Misplaced trust – a case study

- SELECT * FROM Students WHERE Name = 'Bobby Tables';
- $sql = “SELECT * FROM Students WHERE Name =”. <user_input>.”;”
- SELECT * FROM Students WHERE Name = 'Robert'; DROP TABLE Students;--
- Yikes!

Prevention

- Escaping special characters
  - Make a list of special characters for your RDBMS
  - Escape (prepend with \) them by parsing user inputs
- Using prepared statements
  - Modern scripting languages allow the use of prepared statements with placeholders for user inputs
  - Placeholders are typed and don't allow arbitrary string inputs

The evolution of data storage

- Explosion of social media sites (Facebook, Twitter) with large data needs
- Explosion of storage needs in large web sites such as Google, Yahoo
  - Much of the data is not files
- Rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Shift to dynamically-typed data with frequent schema changes

Parallel Data Storage

- Web-based applications have huge demands on data storage volume and transaction rate
- Scalability of application servers is easy, but what about the database?
- Approach 1: memcache or other caching mechanisms to reduce database access
  - Limited in scalability
- Approach 2: Use existing parallel databases
  - Expensive, and most parallel databases were designed for decision support not OLTP
- Approach 3: Build parallel stores with databases underneath

Scaling RDBMS - Partitioning

- “Sharding”
  - Divide data amongst many cheap databases (MySQL/PostgreSQL)
  - Manage parallel access in the application
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware
- Sharding systems and key-value stores don’t support many relational features
    - No join operations (except within partition)
    - No referential integrity constraints across partitions

From weak SQL to NoSQL

- Stands for No-SQL or **N**ot **O**nly **SQL?**
- Class of non-relational data storage systems
  - E.g. BigTable, Dynamo, PNUTS/Sherpa, ..
- Usually do not require a fixed table schema nor do they use the concept of joins
  - Distributed data storage systems
- All NoSQL offerings relax one or more of the ACID properties

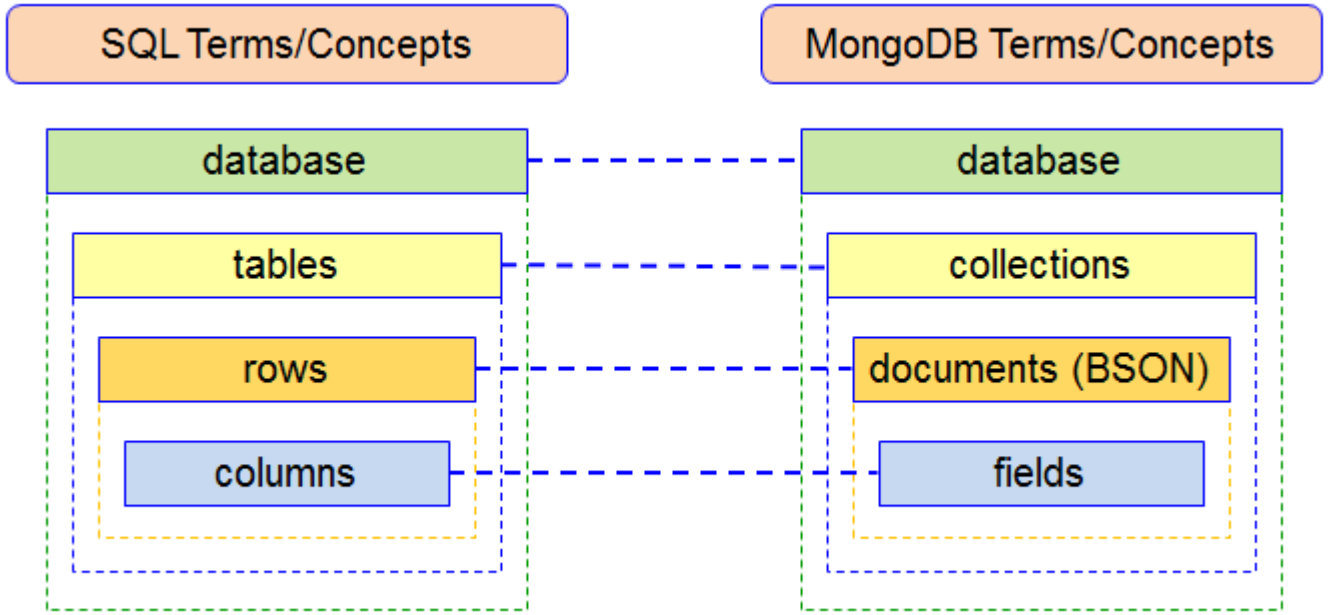
NoSQL Data Storage: Classification

- Uninterpreted key/value or ‘the big hash table’.
  - Amazon S3 (Dynamo)
- Flexible schema
  - BigTable, Cassandra, HBase (ordered keys, semi-structured data),
  - Sherpa/PNuts (unordered keys, JSON)
  - MongoDB (based on JSON)
  - CouchDB (name/value in text)

A typical mongoDB document


```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

Conceptual evolution



CAP Theorem

- Three properties of a system
  - Consistency (all copies have same value)
  - Availability (system can run even if parts have failed)
    - Via replication
  - Partitions (network can break into two or more parts, each with active systems that can't talk to other parts)
- Brewer's CAP "Theorem": You can have at most two of these three properties for any system
- Very large systems will partition at some point
  - Choose one of consistency or availablity
  - Traditional database choose consistency
  - Most Web applications choose availability
    - Except for specific parts such as order processing

Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to ACID
  - Soft state: copies of a data item may be inconsistent
  - Eventually Consistent – copies becomes consistent at some later time if there are no more updates to that data item

Common advantages of NoSQL Systems

- Cheap, easy to implement (open source)
- Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - When data is written, the latest version is on at least one node and then replicated to other nodes
  - No single point of failure
- Easy to distribute
- Don't require a schema

Limitations

- Joins
- ACID transactions
- SQL
- Integration with applications that are based on SQL

Should I be using NoSQL Databases?

- NoSQL Data storage systems makes sense for applications that need to deal with very very large semi-structured data
  - Log Analysis
  - Social Networking Feeds
- Most of the time, we work on organizational databases, which are not that large and have low update/query rates
  - regular relational databases are the correct solution for such applications

In lab this week

- You will work with a sample dataset of FIR data from UP Police
- Download the cases.JSON file from the course website link
- Import it into a mongoDB database
- Figure out how to do basic CRUD operations in the mongo shell
  - Read the mongoDB manual linked on the course website
- Create a web service using PHP/mongo that will tell me
  - Which district has the most crime reported per capita
  - Which police station is the most inefficient in completing investigations
  - Which crime laws are most and least uniquely applied in FIRs