Database Authorization and NoSQL

Databases

Ira Assent

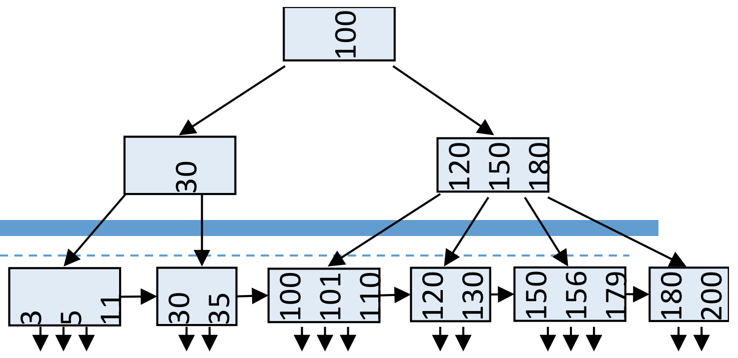
ira@cs.au.dk

Data-Intensive Systems group, Department of Computer Science, Aarhus University, DK

Intended learning outcomes

- Be able to
  - Avoid SQL injection attacks
  - Grant and revoke access to (part of) a database
  - Describe and discuss NoSQL approaches

Recap: Indexes and B+-trees



- Multilevel indexes: index file on top of index file
  - Leaves (lowest level) have data pointers
  - Keep root or top levels in main memory
- B+-tree indexes
  - Like search trees: left subtree contains smaller values, right subtree larger values
  - Optimized for I/O: each node is the size of one disk block
    - Each node has many search keys (attribute value that the index is built upon)
    - The pointer between two search keys leads to the subtree that contains attribute values in the range of values between these two keys
  - All leaves are at the same level, i.e., have the same number of pointers on the path from the root
  - Search from root to leaf by following child pointers
  - Insert by search for where entry should be, add to node
    - If full, need to split leaf and propagate search key to parent
    - If full, split again,… in the worst case root is split and tree grows by one level
  - Delete handled similarly
    - Merge of nodes often ignored in practice, as "empty" space typically filled again quickly

Recap: DB applications

- Using in software program
  - Typical usage scenario
- In your favorite programming language
  - Load the appropriate driver
  - Create a connection
  - Execute query or submit modification
  - May prepare a statement in advance
  - May use variables
  - May loop through input or output rows
- Impedance mismatch
  - Relational database uses relations and rows
  - Programming languages use native types, objects
  - Use cursors to step through query results

```
import java.sql.*;
public class Test {
 public static void main(String args[]) {
  Connection con;
  try {
   String server = "localhost";
   String port = "50000";
   String url =
 "jdbc:mysql://"+server+":"+port+"/sample";
   String userid = ”Mimmi";
   String pwd = ”YourGuess";
 Class.forName("com.mysql.jdbc.Driver").newInstance();
   con = DriverManager.getConnection(url, userid, pwd);
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT * FROM
 Rooms");
    while (rs.next())
    System.out.println(rs.getString(1)+”
 "+rs.getString(2));
   stmt.close();
   con.close();
  } catch(Exception e) { e.printStackTrace(); }}}
```

```
import mysql.connector
cnx = mysql.connector.
connect(user=‘Me’,
database=‘NGO’)
cursor = cnx.cursor()
query = ("SELECT * FROM
ROOMS)
cursor.execute(query)
result = cursor.fetchall()
cursor.close()
for row in result:
    print(row)
    print("\n")
cnx.close()
```

```
ResultSet rs =
stmt.executeQuery("...");
while (rs.next()) {
   ...
   }
rs.close();
```

room	capacity
Turing-216	4
Ada-333	26
Aud-E	286

rs

```
import sqlite3
connection =
sqlite3.connect(‘exampl
e.sqlite')
c = connection.cursor()
for row in
c.execute('SELECT *
FROM Rooms’):
    print(row)
connection.close()
```

Prepared Statements

- SQL statements may be prepared
  - checked and compiled once
  - executed multiple times

```
PreparedStatement pstmt =
    con.prepareStatement("SELECT * FROM Rooms");
ResultSet rs = pstmt.executeQuery();
```

Arguments to Prepared Statements

- Use ? symbols for variables
- Insert values using absolute position

```
PreparedStatement pstmt =
   con.prepareStatement(
        "INSERT INTO Meetings VALUES(?,?,?,'dDB',?)"
   );
pstmt.setInt(1,34716);
pstmt.setDate(2,new java.sql.Date(2014,10,6));
pstmt.setInt(3,14);
pstmt.setString(4,“ira");
pstmt.executeUpdate();
```

What happens?

```
"SELECT *
FROM Users
WHERE userid ='" + userid + "'"
```

Variable userid is “`x' OR 'y'='y`”

1. Only users with userid x returned
2. Empty result
3. All data is returned
4. Error message

SQL Injection Attacks

- Be careful with dynamic SQL:

```
"SELECT * FROM Users WHERE userid ='" + userid + "'"
```

- Fine if userid is ”`ira`"
- Bad if userid is "`x' OR 'y'='y`"
  - all data is revealed
- Worse if userid is ”`x';DROP TABLE Users;`"
  - all data is deleted

Injection Attack in Python

```
import sqlite3
connection = sqlite3.connect('users.sqlite')
c = connection.cursor()
c.execute('CREATE TABLE users (name)')
while True:
    user = input('New user: ')
    c.executescript('INSERT INTO users
                     VALUES ("%s")' % user)
```

can execute a string containing several SQL statements

```
New user: evil"); DROP TABLE users;
```

INSERT INTO users VALUES ("evil"); DROP TABLE users; ")

Insecure: NEVER use % on user input

```
connection.commit()
    print(list(c.execute('SELECT * FROM
                          users')))
```

Protection against SQL injection

- Bind Variables
  - Instead of just passing inputs directly to execution, bind inputs to variables first
  - JDBC example (using prepared statement):

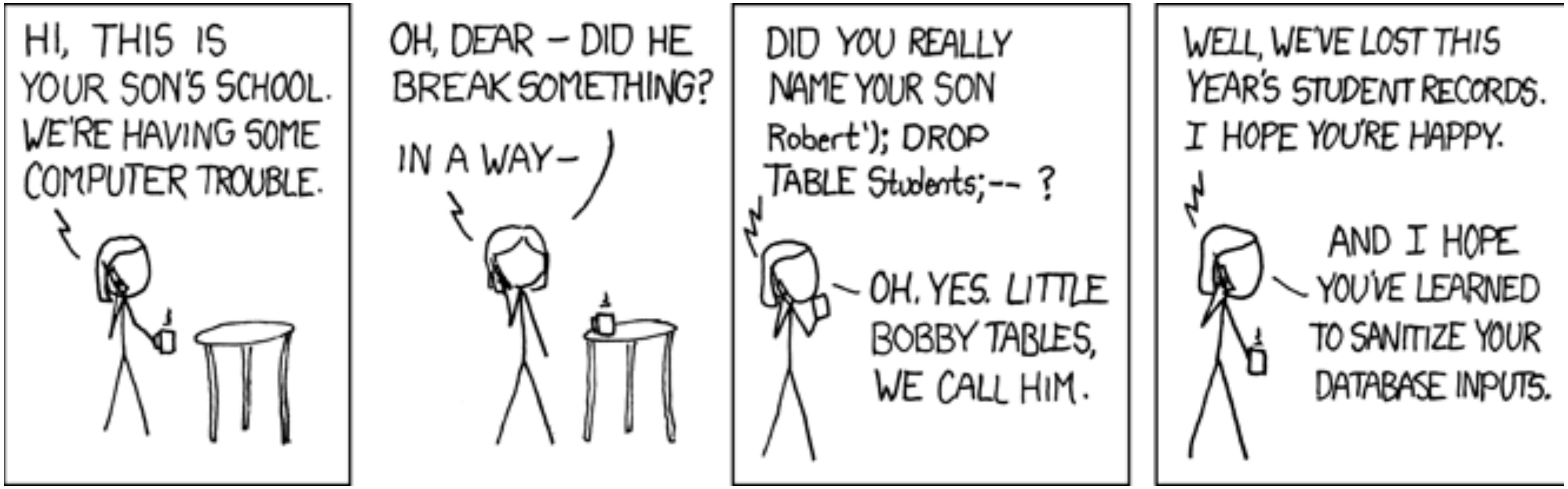
```
PreparedStatement stmt =
conn.prepareStatement("SELECT * FROM Users WHERE
userid = ?");
Stmt.setString(1, userid);
```

  - Python example:

```
c.execute('INSERT INTO users VALUES (?)',(user,))
```

  - **SQL statement is compiled before user input is added**
  - **Input does not become part of SQL execution code**
- Sanitize inputs
  - Upon receiving user input, pass it through a filter
    - Either match against valid pattern (preferred)
    - Strip it from invalid patterns (such as escape characters)

Be Prepared…


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Database Security

- Securing database against threats
  - Part of overall systems security, network security
- Focus here on **discretionary security** mechanisms:
  - **Grant privileges** to users / user **authorization**
    - Capability to access
      - specific data files, tuples or attribute values
      - In a specified **mode** (read, insert, delete,…)
- Database administrator (DBA)
  - Creates accounts for (groups of) users and assigns appropriate security clearance level
  - Grants and revokes privileges

Authorization

- Controlling access to the database
- Authorization Identities
  - The owner of a database object: unlimited privileges to read and modify
  - Users: identified by their login identities, explicitly assigned privileges
  - Roles: hierarchical groups of users
  - Public: corresponds to the group of all users and roles
- Objects
  - tables or views (or triggers, types,....)
- Privileges
  - a range of different actions

Granting Privileges

- `GRANT` *`privileges`* `ON` *`object`* `TO` *`identity`*

```
GRANT SELECT, UPDATE(office)
      ON People
      TO amoeller;
GRANT INSERT, DELETE
      ON Exams
      TO engberg;
```

Privileges

- `SELECT` (retrieval or read): the right to query a table
- **MODIFY** privileges:
  - `INSERT`: the right to insert rows (or individual attributes)
  - `DELETE`: the right to delete rows
  - `UPDATE`: the right to update rows (or individual attributes)
- `REFERENCES`: the right to include an attribute in a foreign key

https://dev.mysql.com/doc/refman/8.0/en/grant.html

Transitive Granting

```
GRANT privileges ON object TO identity
      WITH GRANT OPTION;
```

- Now the grantee may **propagate** / pass on these (or weaker) privileges to other identities
  - Weaker means e.g. same privilege on subset of attributes:

```
GRANT SELECT, UPDATE
 ON Exams TO amoeller
 WITH GRANT OPTION;
```

} Issued by ira

```
GRANT UPDATE(date, time)
 ON Exams TO engberg;
```

} Issued by amoeller

Textbook example

- DBA creates accounts A1, A2, A3, A4
- only A1 should be able to create tables (CREATETAB privilege)
- DBA issues `GRANT CREATETAB TO A1;`
- **Account privilege**
  - Not restricted to any relations or attributes
- A1 creates relations EMPLOYEE and DEPARTMENT
  - A1 is then **owner** of these two relations and has all relation privileges on each of them

EMPLOYEE

Name	Ssn	Bdate	Address	Sex	Salary	Dno

DEPARTMENT

Dnumber	Dname	Mgr_ssn

- A1 gives A2 right to insert and delete tuples in both relations, but without the capability of passing them on by issuing

```
GRANT INSERT, DELETE ON EMPLOYEE, DEPARTMENT TO A2;
```

Revoking Privileges

```
GRANT SELECT, UPDATE
  ON Exams TO amoeller
  WITH GRANT OPTION;
GRANT UPDATE(date, time)
  ON Exams TO engberg;
```

```
REVOKE privileges ON object
  FROM identities CASCADE
```

- Cancels / **revoke**s privileges, also removes transitive privileges
- E.g. `REVOKE UPDATE ON Exams FROM amoeller CASCADE` also removes date, time update privilege on Exams from engberg who received it from amoeller (unless engberg also got it elsewhere!)

```
REVOKE privileges ON object
  FROM identities RESTRICT
```

- revoke privileges, fail if transitive privileges are affected
- E.g. `REVOKE UPDATE ON Exams FROM amoeller RESTRICT` cannot be executed

- REVOKE GRANT OPTION FOR *privileges* ...
  - revoke the right to grant future transitive privileges
  - `REVOKE GRANT OPTION FOR UPDATE ON Exams FROM amoeller` means that amoeller can no longer pass on UPDATE privileges on Exams

EMPLOYEE

Name	Ssn	Bdate	Address	Sex	Salary	Dno

DEPARTMENT

Dnumber	Dname	Mgr_ssn

Textbook example

- A1 allows A3 to retrieve information from either of the two tables and also to be able to propagate by issuing

```
GRANT SELECT ON EMPLOYEE, DEPARTMENT
      TO A3 WITH GRANT OPTION;
```

- A3 passes on retrieval privilege on Employee to A4 by issuing:

```
GRANT SELECT ON EMPLOYEE TO A4;
```

  - Notice that A4 cannot pass on this privilege because no GRANT OPTION used
- A1 decides to take away the retrieval privilege on Employee from A3 by issuing:

```
REVOKE SELECT ON EMPLOYEE FROM A3;
```

- DBMS automatically revokes SELECT privilege on EMPLOYEE from A4, too, because A3 granted that privilege to A4 and A3 does not have the privilege anymore

Which privileges needed?

```
INSERT INTO Beers(name)
  SELECT beer FROM Sells
  WHERE NOT EXISTS
       (SELECT * FROM Beers
        WHERE name = beer);
```

1. SELECT on Sells and Beers, and INSERT on Beers
2. UPDATE on Sells and Beers
3. SELECT and INSERT on Sells and Beers
4. SELECT on Sells and Beers, and UPDATE on Beers
5. Don't know

The References Privilege

```
CREATE TABLE Rooms (
   room        VARCHAR(15) PRIMARY KEY,
   capacity    INT     );
CREATE TABLE People (
   name        VARCHAR(40) NOT NULL,
   office      VARCHAR(15) REFERENCES Rooms(room),
   userid      VARCHAR(15) PRIMARY KEY,
   `group`     CHAR(3) );
GRANT INSERT ON People TO amoeller;
GRANT REFERENCES(room) ON Rooms TO amoeller;
INSERT INTO People VALUE(...);
```

- REFERENCES privilege on R gives capability to reference relation R when specifying integrity constraints
- can also be restricted to specific attributes of R (typically the key attributes)
- Because the validity of a foreign key yields information!

Roles

- Combines groups of users with set of privileges

```
CREATE ROLE Paymaster;
GRANT UPDATE(salary) ON Payroll TO Paymaster;
GRANT Paymaster TO amoeller WITH GRANT OPTION;
```

owner

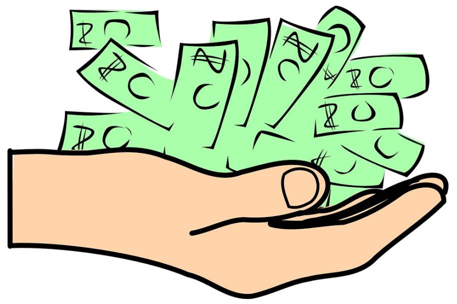
```
SET ROLE Paymaster;
GRANT Paymaster TO engberg;
```

amoeller

- All users and roles have the role public
- Role Hierarchies
  - All privileges are inherited

```
CREATE ROLE Employee;
CREATE ROLE Paymaster;
CREATE ROLE Boss;
GRANT Employee TO Paymaster;
GRANT Paymaster TO Boss;
```

$$\begin{tikzcd}
\boxed{\text{Boss}} \\
\boxed{\text{Paymaster}} \arrow[u] \\
\boxed{\text{Employee}} \arrow[u]
\end{tikzcd}$$



Views

```
CREATE VIEW Vips AS
      SELECT * FROM People
      WHERE group = 'vip';

GRANT SELECT ON Vips TO amoeller;
```

- Privileges on the base tables are not considered
  - they are neither implicitly granted nor required
- INSERT, DELETE, and UPDATE privileges are meaningful for modifiable views
- SELECT privileges are for the entire table
  - Views allows finer granularity
  - E.g. some administrator can see some employee information but not salaries

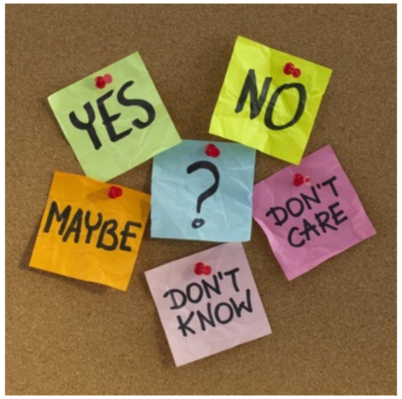
Textbook example

- A1 wants to give back to A3 limited capability to retrieve name, birthday and address information of employees from Department 5
- A1 creates view:

```
CREATE VIEW A3EMPLOYEE AS
  SELECT NAME, BDATE, ADDRESS
  FROM EMPLOYEE
  WHERE DNO = 5;
```

- And issues:

```
GRANT SELECT ON A3EMPLOYEE TO A3
     WITH GRANT OPTION;
```



- A1 allows A4 to update only employee salaries by issuing

```
GRANT UPDATE ON EMPLOYEE (SALARY) TO A4;
```

  - UPDATE or INSERT privilege can specify attributes
  - SELECT, DELETE not attribute specific: use views

EMPLOYEE

Name	Ssn	Bdate	Address	Sex	Salary	Dno

DEPARTMENT

Dnumber	Dname	Mgr_ssn

NoSQL Motivation

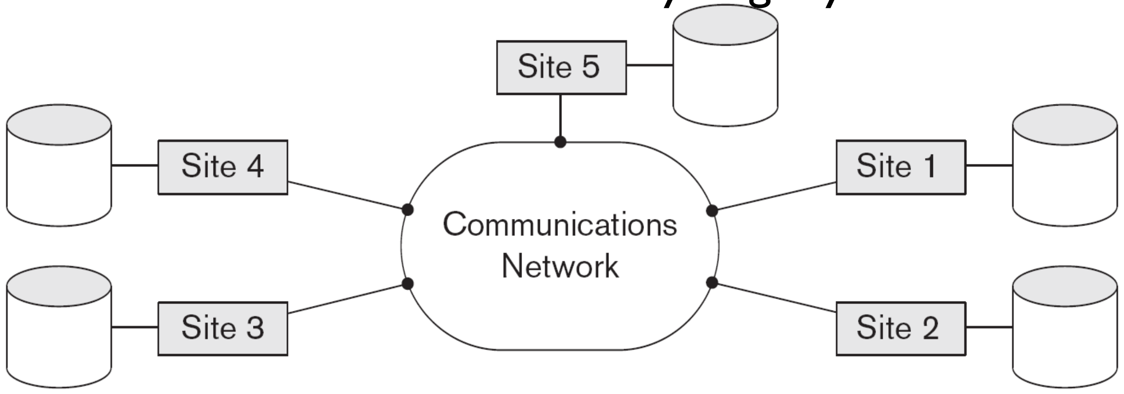
- Huge data volumes in companies like Google, Meta, X
  - Social media, web, spatial data,
  - Applications require more than traditional relation SQL based systems
- Focus
  - High performance: efficient response time
  - Availability: always able to respond (no / very little downtime)
  - Scalability: can handle huge and continuously increasing data volumes
    - Challenges for traditional systems
      - Powerful query language, concurrency control etc. not important
      - Relational model too restrictive: schema models required, effort in maintaining constraints etc.
- **NOSQL: Not Only SQL** (not NO to SQL)
- Consider e.g. Google Mail
  - Millions of users
  - Each can have thousands of messages, possibly with attachments

NoSQL systems

- In the interest of performance, availability, scalability, willing to sacrifice
  - some (immediate) data consistency
  - powerful query languages
  - structured data storage
- Google developed BigTable
  - Used in Gmail, Google Maps
  - Similar concepts in Apache Hbase (open source)
  - Column-based / wide column store / column family store
- Other examples
  - Amazon DynamoDB (key-value)
  - Facebook Cassandra (now open source; key-value, column-based)
  - MongoDB, CouchDB (document store)
  - Neo4J, GraphBase (graph based)
  - OrientDB
  - …

NoSQL Approach: distribution

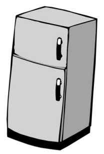
- Most NOSQL use **distributed databases** / distributed storage
  - Database across several computers
    - Called **sites**, nodes, local databases
  - Defined as collection of multiple logically related databases distributed over computer network (e.g. internet), managed by distributed DBMS (DDBMS)
- NoSQL emphasizes high availability
  - Typically full replication: each data item stored more than once in several copies
    - Different models for site responsible for **master copy**
      - Updates are propagated to **slave copies**
  - physical placement of data (files, relations, etc.) not known to the user / application program
    - You do not know from which site you get your Gmail data



Bandwidth and network latency

- In a NoSQL database, we want to minimize the amount of data that needs to be transmitted over the network
  - Generally, the network is the internet
    - New measure of complexity: e.g. transferred data volume (or communication rounds) instead of disk I/O
      - network latency and bandwidth dominate response time

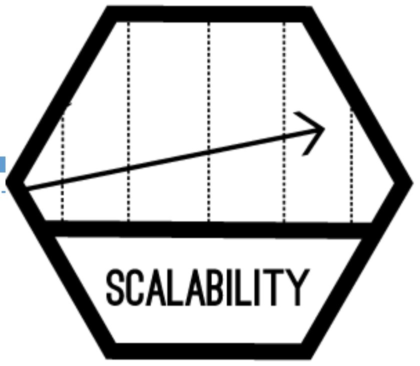
- Just as you thought having to go to the fridge was bad, turns out going to the shop is worse…



Data replication has?

- Improved read and write performance
- Improved write, poorer read performance
- Improved read, poorer write performance
- Poorer read and write performance

Trade-offs in NoSQL



- Scalability
  - Data typically keeps growing in volume
    - Add nodes to distributed system for data processing and storage
    - Usually scale while system is under operation: no interruption
- High performance
- Typically consistency less important
  - In relational databases, we are guaranteed consistency, i.e., the data is always in agreement with all integrity constraints
  - In NoSQL, use weaker eventual consistency notion
    - Slaves (sites with copies) eventually hold same data as master (main site responsible for a particular data item), but there may be temporal inconsistencies in the meantime

CAP properties

- Goals when replicating data
  - Consistency
    - Same values among replicas of any item
  - Availability
    - System successfully processes read/writes
  - Partition tolerance
    - System continues operation even if partitioned by network fault

- **Important difference in ”consistency”**
  - In relational DBMS: do not violate integrity constraints on schema
  - In NoSQL replication: values of replicas identical

CAP theorem

- Impossible to guarantee all three: C, A, P
- Need to choose any two properties if desired
- Typically, A and P most important in NoSQL systems
- Resort to weaker consistency
  - **Eventual consistency**
    - Values at all copies are updated eventually

NoSQL systems: data model

- Emphasize performance and flexibility over modeling power and complex querying
- **No schema** required
  - Allow **semi-structured**, **self-describing** data
    - JSON (JavaScript Object Notation), XML (Extensible Markup Language),…
    - Idea: description, such as tags, part of the data objects
  - Specify partial schema for improved query efficiency, but not a requirement
  - Since no constraints can be specified, checking must be handled by application programs

XML

- Extensible Markup Language
- Self-describing document
  - **tags <>** to annotate the data with **meta-data** (also human readable)
    - Can be seen as a kind of local schema along with the data, only for this data item
  - Hierarchically nested tags possible
  - Tags **well-formed** (opening and closing tags in the correct order)

```
<?xml version="1.0" encoding="UTF-8" ?>
<document>
    <type>Student Record</type>
    <name>Anne Christensen</name>
    <study>
        <studyname>Computer Science</studyname>
        <level>Bachelor</level>
    </study>
</document>
```

https://www.w3.org/XML/

JSON

- JavaScript Object Notation
- Similar to XML self-describing, only opening tag (bracket notation)

```
{
        "document" : {
                 "type" :  "Student Record",
                "name": "Anne Christensen",
                "study":  {
                          "studyname": "Computer Science",
                          "level": "Bachelor"
                }
        }
}
```

https://www.json.org/json-en.html

VOTE

Why self-describing formats in NoSQL?

A. They are easier to read.
B. They are more flexible.
C. They are easier to normalize.
D. They allow for better compression.

NoSQL systems: query languages

- Less powerful query languages
- Many NoSQL based systems rely on simple search/read access
  - Find particular object using id
  - No need for complex query languages that express powerful conditions and combinations across tables
- Many NoSQL systems provide functions and operations via an API (application programming interface)
  - Read/write done via function/operation calls
  - CRUD (Create, Read, Update, Delete) operations
  - SCRUD (plus Search)
  - Some provide query language that corresponds to subset of SQL capabilities
    - Typically join not available
    - If needed: must be handled by application program
- Versioning
  - Some NoSQL systems provide storage of multiple versions
  - Timestamps of version creation

Summary

- Intended learning outcomes
- Be able to
  - Avoid SQL injection attacks
  - Grant and revoke access to (part of) a database
  - Describe and discuss NoSQL approaches

Where to go from here?

- We'll see some more concrete examples of NoSQL database such as writing queries in a graph NoSQL database

What was this all about?

Guidelines for your own review of today's session

- SQL injection attacks are…
  - We should avoid them by…
- Authorization allows to…
  - We can use identities… on objects…
  - To define privileges…
  - Transitive granting is…
  - We can revoke…
  - Roles and views are useful to…
- NoSQL databases are motivated by…
  - The core idea is to…
  - In particular, they focus on…
  - They sacrifice…