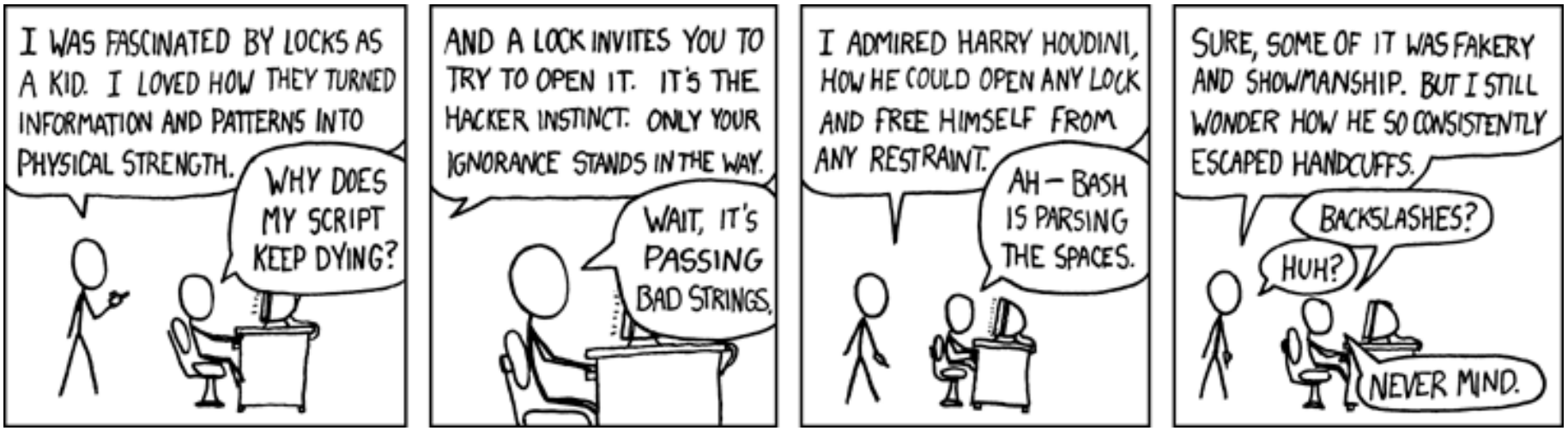
Locks


I WAS FASCINATED BY LOCKS AS A KID. I LOVED HOW THEY TURNED INFORMATION AND PATTERNS INTO PHYSICAL STRENGTH.
WHY DOES MY SCRIPT KEEP DYING?
AND A LOCK INVITES YOU TO TRY TO OPEN IT. IT'S THE HACKER INSTINCT: ONLY YOUR IGNORANCE STANDS IN THE WAY.
WAIT, IT'S PASSING BAD STRINGS.
I ADMIRED HARRY HOUDINI, HOW HE COULD OPEN ANY LOCK AND FREE HIMSELF FROM ANY RESTRAINT.
AH — BASH IS PARSING THE SPACES.
SURE, SOME OF IT WAS FAKERY AND SHOWMANSHIP. BUT I STILL WONDER HOW HE SO CONSISTENTLY ESCAPED HANDCUFFS.
BACKSLASHES?
HUH?
NEVER MIND.

SQL in applications; Transactions

Rasmus Pagh

Hand-in 2, general feedback

- Generally very good answers on the SQL part.
- Sensible choice of indexes.
- However, most groups neglected a real discussions of why/how the indexes work:

A transcript of a database terminal session running all queries in MySQL on the loaded data set, and also showing query plans (using EXPLAIN). Discuss how the observed behavior fits with your understanding of indexes.

SQL in applications

- Most SQL databases are at the back end of applications
  - important to know how this works.
- On the surface, a very boring subject
  - How to move data from A to B doing suitable translation, etc.
- Also a very interesting topic!
  - Focus of a lot of research and development.

- In this course we will stay pretty much at the surface…

JDBC connection

```
String url = "jdbc:mysql://localhost/";
String dbName = "imdb";
String driver = "com.mysql.jdbc.Driver";
String userName = "root";
String password = "";

try {
    Class.forName(driver);
    Connection conn = DriverManager.getConnection(url+dbName,userName,password);
    System.out.println("Connected to MySQL");
    
    Database operations

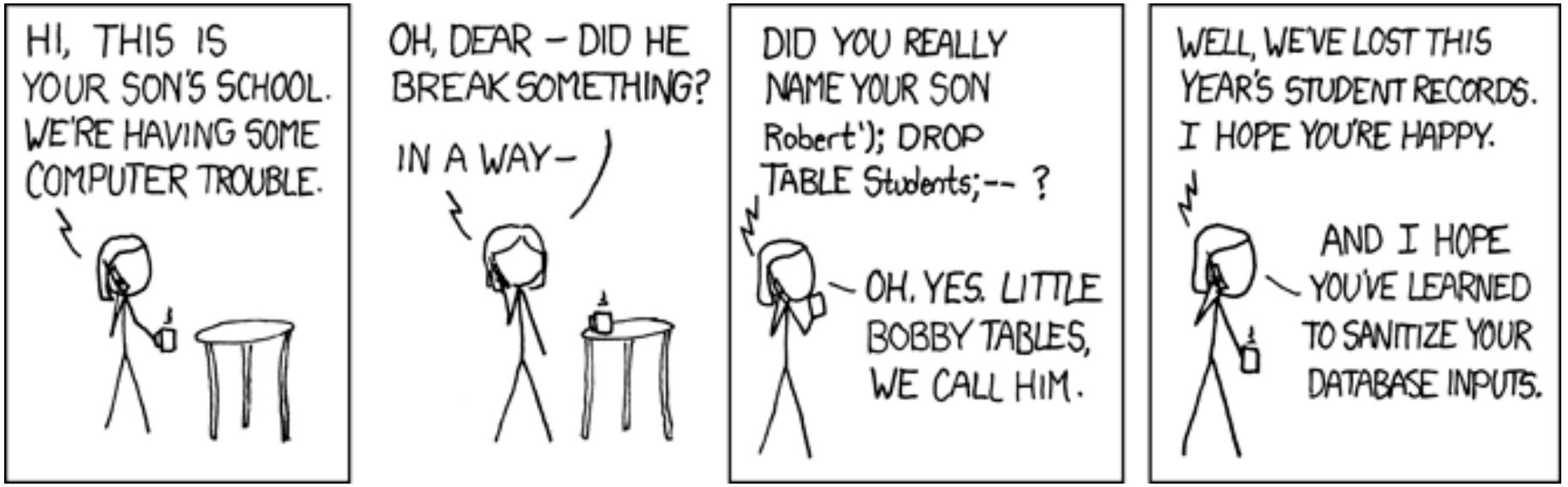
    conn.close();
    System.out.println("Disconnected from MySQL");
} catch (Exception e) {
    e.printStackTrace();
}
```

JDBC dynamic SQL

```
try {
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery("SELECT gender, count(*) FROM person GROUP BY gender");
    while (rs.next()) {
        System.out.println(rs.getString("gender")+": "+rs.getInt(2));
    }

    st.executeUpdate("DROP TABLE IF EXISTS JDBCtest");
    st.executeUpdate("CREATE TABLE JDBCtest(id int, string varchar(10))");
    st.executeUpdate("INSERT INTO JDBCtest VALUES (1,\"Tada!\")");
}
catch(SQLException s){
    System.out.println(s.toString());
}
```

Use caution when creating SQL based on user input!


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.

xkcd.com

JDBC static SQL

```
PreparedStatement insertPerson =
conn.prepareStatement("INSERT INTO person VALUES (?,?,?,?,?,?)"); // Create preparec
insertPerson.setInt(1, 123456);
insertPerson.setString(2, "John Doe");
insertPerson.setString(3, "M");
insertPerson.setDate(4, new java.sql.Date(160617600000)); // Set date, given in mili
insertPerson.setNull(6,java.sql.Types.INTEGER); // Set to NULL
insertPerson.executeUpdate(); // Execute prepared statement with current parameters
```

Efficiency issues

- Connection takes time to establish – use 1 connection for many operations.
- It takes time to parse dynamic SQL – prepared statements start executing faster.
- `ORDER BY` may force creation of full result within the DBMS before any output reaches the application.
  - Why is this not just usual? (Answer in next slide.)

Cursors

- Common to not generate full results of queries, but provide a ”cursor” that allows the result to be traversed.
- JDBC examples:
  - `Statement s = con.createStatement (ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)`
  - `Statement s = con.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)`

Four examples

1. Movies by year – imperative way
2. Movies by year – SQL centric way
3. Iterating through a large result set
4. Iterating through a filtered result set

Automatic code generation

- Instead of dealing directly with JDBC, one can automatically generate code to make objects "persistent" in a database.
  - E.g. Nhibernate
- Advantage: Tedious code made with very little effort.
- Disadvantage: Little and indirect control over efficiency issues.

Language integration

- ”Little languages” with tight database integration.
  - E.g. ”Ruby on Rails”, `http://en.wikipedia.org/wiki/Ruby_on_Rails`
- New query sublanguages for mainstream languages such as C#.
  - E.g. LINQ, `http://en.wikipedia.org/wiki/Language_Integrated_Query`
  - If used with conventional DBMS: Automatically translated to SQL.

Part II: Transactions

Why transactions?

Consider the following three transactions on the relation `accounts(no,balance,type)`:

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance<0;
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance>0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

Transactions in JDBC

- ```
  conn.setAutoCommit(false);
  // Disable automatic commit after
  each statement
  ```

- ```
  conn.commit();
  // Commit all pending updates
  ```

- ```
  conn.rollback();
  // Abort all pending updates
  ```
  ```

Course goals

After the course the students should be able to:

- identify possible problems in transaction handling, related to consistency, atomicity, and isolation.

ACID Properties

Atomicity: Each transaction runs to completion or has no effect at all.

Consistency: After a transaction completes, the integrity constraints are satisfied.

Isolation: Transactions executed in parallel have the same effect as if they were executed sequentially.

Durability: The effect of a committed transaction remains in the database even if the computer crashes.

Durability in a nutshell

- There exist disk systems that are highly reliable (e.g. still functions if one or two disks fail).
  - Trade-off: Redundancy vs reliability
- A database transaction is only really committed when the actions made by the transaction have *all* been written to the *log* on disk.
  - In case of crash, the log is used to reverse the state to the one implied by committed transactions.

Today: Atomicity and isolation

- This lecture is mainly concerned with atomicity and isolation.
- Consistency is a consequence of atomicity and isolation + maintaining any declared DB constraint (not discussed in this course).

Isolation and serializability

- Want transactions to satisfy *serializability*:
  - The state of the database should always look as if the committed transactions ran in a *serial schedule*.
- The scheduler of the DBMS is allowed to choose the order of transactions:
  - It is not necessarily the transaction that is started first, which is first in the serial schedule.
  - The order may even look different from the viewpoint of different users.

A simple scheduler

- A simple scheduler would maintain a queue of transactions, and carry them out in order.
- Problems:
  - Transactions have to *wait* for each other, even if unrelated (e.g. requesting data on different disks).
  - Possibly smaller throughput. (Why?)
  - Some transactions may take very long, e.g. when external input or remote data is needed during the transaction.

A simple scheduler

- A simple scheduler would maintain a queue of transactions, and carry them out in order.
- Some believe this is fine for *transaction processing*:

The End of an Architectural Era
(It's Time for a Complete Rewrite)

Michael Stonebraker
Samuel Madden
Daniel J. Abadi
Stavros Harizopoulos

Nabil Hachem
AvantGarde Consulting, LLC
nhachem@agdba.com

Pat Helland
Microsoft Corporation
phelland@microsoft.com

Interleaving schedulers

- Most DBMSs have schedulers that allow the actions of transactions to interleave.
- However, the result should be **as if** some serial schedule was used.
- We will now study a mechanism that *enforces "*serializability": **Locking**.
- Other methods exist: Time stamping / optimistic concurrency control.
  - Out of scope for this course.

Locks

- In its simplest form, a lock is a right to perform operations on a database element.
- Only one transaction may hold a lock on an element at any time.
- Locks must be requested by transactions and granted by the locking scheduler.

Two-phase locking

- Commercial DBMSs widely use two-phase locking, satisfying the condition:
  - In a transaction, all requests for locks precede all unlock requests.
- 2PL ensures serializability:
  From the point of view of other transactions, each transaction will appear to have run at the time of the first unlock request.

Strict two-phase locking

- In **strict** 2PL all locks are released when the transaction completes.
- This is commonly implemented in commercial systems, since:
  - it makes transaction **rollback** easier to implement, and
  - avoids so-called **cascading aborts** (this happens if another transaction reads a value by a transaction that is later rolled back)

Lock modes

- The simple locking scheme we saw is too restrictive, e.g., it does not allow different transactions to read the same DB element concurrently.
- **Idea**: Have several kinds of locks, depending on what you want to do. Several locks on the same DB element may be ok (e.g. two read locks).

Shared and exclusive locks

- Locks for reading can be shared (S).
- Locks needed for writing must be exclusive (X).
- Compatibility matrix says which locks are granted:

	Lock requested		
Lock held		S	X
	S	Yes	No
	X	No	No

Phantom tuples

- Suppose we lock tuples where A=42 in a relation, and subsequently another tuple with A=42 is inserted.
- For some transactions this may result in unserializable behaviour, i.e., it will be clear that the tuple was inserted during the course of a transaction.
- Such tuples are called phantoms.

Avoiding phantoms

- Phantoms can be avoided by putting an exclusive lock on a relation before adding tuples.
  - However, this gives poor concurrency.
- A technique called "index locking" can be used to prevent other transactions from inserting phantom tuples, but allow most non-phantom insertions.
- In SQL, the programmer may choose to either allow phantoms in a transaction or insist they should not occur.

SQL isolation levels

- A transaction in SQL may be chosen to have one of four isolation levels:
  - `READ UNCOMMITTED`: *Dirty reads* are possible.
  - `READ COMMITTED`: Dirty reads are not permitted (but nonrepeatable reads and phantoms are possible).
  - `REPEATABLE READ`: Nonrepeatable and dirty reads are not permitted (but phantoms are possible).
  - `SERIALIZABLE`: Transaction execution must be serializable (above anomalies not allowed).

SQL isolation levels

- Possible implementations:
  - `READ UNCOMMITTED`:
    ”No locks are obtained.”
  - `READ COMMITTED`:
    ”Read locks are immediately released - read values may change during the transaction.”
  - `REPEATABLE READ`:
    ”2PL but no lock when adding new tuples.”
  - `SERIALIZABLE`:
    ”2PL with lock when adding new tuples.”

Isolation level syntax

- Begin transaction with:

```
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED |
  READ COMMITTED |
  REPEATABLE READ |
  SERIALIZABLE }
```

ACID testing MySQL

- Most storage engines are made with only very simple concurrency control in mind.
- InnoDB (default engine) supports the standard SQL isolation concurrency control features, and more.
- Beware: Using SQL isolation levels with another storage engine may have no effect (except perhaps a warning).

Be careful with SERIALIZABLE

- Some common implementations of ”SERIALIZABLE” allows, e.g., the following:
  - Suppose we have a relation R with a tuple for each reserved seat in a plane.
  - Transactions A and B simultaneously read R and find that seat 13A is free.
  - Transaction A and B both insert a tuple indicating that seat 13A has been booked.
- Such conflicts can be stopped by a lock or by a database constraint.

Explicit row locking

- Many DBMSs allow transactions to explicitly lock a set of tuples.
- Example:

```
SELECT * FROM seats
WHERE seat = '13A'
FOR UPDATE;
```

- Can be used to control a resource, e.g. the right to insert a reservation tuple for seat 13A in another table.

Snapshot isolation

- Some DBMSs implement **snapshot isolation**, an isolation level that gives a stronger guarantee than `READ COMMITTED`.
- MySQL/InnoDB:
  `START TRANSACTION WITH CONSISTENT SNAPSHOT;`
- Each transaction T executes against the version of the data items that was committed "when the T started".
- Possible implementation:
  - No locks for read, locks for writes.
  - Store old versions of data (costs space).

Lock granularity in MySQL

- So far we did not discuss what DB elements to lock: Atomic values, tuples, blocks, relations?
- InnoDB uses row-level locking by default.
  - No ”lock escalation”.
- Table locking can be done manually:
  - LOCK TABLES T1 READ, T2 WRITE,…
  - UNLOCK TABLES

Locks and deadlocks

- The DBMS sometimes must make a transaction wait for another transaction to release a lock.
- This can lead to deadlock if e.g. A waits for B, and B waits for A.
- In general, we have a deadlock exactly when there is a cycle in the waits-for graph.
- Deadlocks are resolved by aborting some transaction involved in the cycle.

Simple deadlock prevention

- 2001 MySQL manual:

  All locking in MySQL is deadlock-free.

  This is managed by always requesting all needed locks at once at the beginning of a query and always locking the tables in the same order.

- 2008 MySQL manual:

  All locking in MySQL is deadlock-free, except for *InnoDB* and *BDB* type tables.

  – Explanation? Why use InnoDB and BDB?

- **Question**: Why does ”always locking tables in the same order” never lead to deadlock?

Summary

- Concurrency control mechanisms give various trade-offs between isolation and performance.
  - Safe choice is SERIALIZABLE (well…)
  - Sometimes lower SQL isolation levels suffice – difficult to analyze in general
  - Manual efforts may sometimes be better: Table locking, explicit row locking,…
  - Deadlocks happen. A simple (but brutal) cure is lock acquisition in fixed order.
- Better summary: A song!

Next steps

- Exercises on JDBC, and torturing your DBMS, at 12.30.

- You now have all the background to start working on hand-in 3, due 11/11.
- It is allowed to discuss the hand-in, but you should write your own answer **individually** without help!

- Next Friday: Lecture break.
- Ninh is available for questions 10-11.30 in room 4D27.