Advanced Database Systems

Course introduction

Feliz Gouveia

UFP, January 2013
Introduction

• 6 ECTS course of the 1st year, 2nd cycle of Engenharia Informática (Computer Science)
• The Advanced Database Systems course extends the topics of the introductory Database Management Systems course, and introduces new ones: OODB, Spatial data, columnar databases
# Degree structure (2nd cycle)

<table>
<thead>
<tr>
<th>1st year</th>
<th>2nd year</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS Planning and Development / Mobile Networks and Services I (6)</td>
<td>Final Project I (12)</td>
</tr>
<tr>
<td>Mobile Computing (6)</td>
<td>Knowledge Management / Mobile Networks and Services II (3)</td>
</tr>
<tr>
<td>Professional Ethics (2)</td>
<td>Multimedia and Interactive Systems / Mobile Applications Programming (4)</td>
</tr>
<tr>
<td>Academic Internship (8)</td>
<td>Programming Paradigms / Mobile Applications Project (4)</td>
</tr>
<tr>
<td><strong>Advanced Database Systems (6)</strong></td>
<td>Elective course (4)</td>
</tr>
<tr>
<td>Artificial Intelligence (8)</td>
<td>Research Methods (3)</td>
</tr>
<tr>
<td>Distributed Systems (8)</td>
<td>Final Project II (3)</td>
</tr>
<tr>
<td>Man-Machine Interaction (6)</td>
<td>Dissertation (27)</td>
</tr>
<tr>
<td>Computer Security and Auditing (6)</td>
<td></td>
</tr>
<tr>
<td>Elective course (4)</td>
<td></td>
</tr>
</tbody>
</table>
Course goals

• To provide the students with a better understanding of the essential techniques used in a Database Management System, either by revisiting them or by studying new approaches.

• To provide students with knowledge to choose, design, and implement a dbms in a complex domain, making the best use of the available tools and techniques.

• To provide students with knowledge to analyze and tune a given dbms, given a workload and usage patterns.
Course goals

• To allow the students to learn and experiment advanced database techniques, models and products, and to provide them with the knowledge to take decisions concerning implementation issues.

• To provide students with knowledge to analyze, modify if necessary and experiment algorithms that make up the database internals.

• To expose students to advanced topics and techniques that appear promising research directions.
Course outcomes

• Describe dbms internals. Understand and describe internal algorithms in detail.

• Identify and be able to use recent and advanced database techniques.

• Decide on configuration issues related to database operation and performance. Identify which parameters are tunable and what are the implications.

• Analyze and optimize transactional code, identifying causes of possible anomalies and correct them.
Course outcomes

• Decide on optimization issues given a known database workload, by manipulating indexes, choosing more adequate data types, and modifying queries.

• Identify limitations of the standard Relational databases in certain application domains, e.g. for multidimensional data, or unstructured data.

• Analyze, describe and use other models than the Relational.
Course outcomes

• Identify opportunities for the use of the object model, and design and code client code to manipulate an object database.

• Analyze, compare and evaluate alternative database architectures and models in different application contexts.
# Course Syllabus

<table>
<thead>
<tr>
<th>Module</th>
<th>Topics</th>
<th>Contact Hours / ECTS</th>
</tr>
</thead>
</table>
| Database internals and advanced algorithms | - Transaction Management  
  - Concurrency control  
  - Storage organization  
  - Recovery | 24 hours / 2.5 ECTS |
| Object-oriented databases | - The Object-Relational approach  
  - Object-oriented databases  
  - Object Query languages  
  - Architecture of a OODB. Transactions. | 12 hours / 1.5 ECTS |
| Other database models | - Representation of Spatial data  
  - Spatial data operators and indexing  
  - Non-SQL databases  
  - Architecture and applications of columnar databases | 18 hours / 2 ECTS |
Concurrency control

• Students learned standard techniques
• Important topic. In practice, can result in:
  – Poor performance
  – Database inconsistencies
• Products have their own implementations
  – Not always standard
• Recent research with real implications
Snapshot Isolation

Feliz Gouveia
Porto, January 2013
Plan

1. The scheduling problem
2. Serializable schedules
3. Scheduling approaches
   - Locking
   - Multi-version
4. Snapshot Isolation (SI)
5. SI anomalies
6. Serializable SI
1. The scheduling problem

- Basic concurrency control question:
  - If several processes or users are modifying and querying a database, how to guarantee correctness?

- And,
  - How to achieve that correctness allowing the maximum degree of concurrency?
Concurreny anomalies

• In the following schedule, two transactions manipulate an account:

\[ T1 : R(x = 2)_{t}, \ x := x + 12, \ W(x = 14)_{t+2} \]
\[ T2 : \ R(x = 2)_{t+1}, \ x := x - 5, \ W(x = -3)_{t+3} \]

• The Bank checks the general ledger, and expects to find \( x = 9 \), but in the database finds \( x = -3 \)
How to avoid them?

• Disallowing concurrency is not an option
  – Although a serial schedule is always correct

• Identification of conflicts. A pair of operations $p$ and $q$ is in conflict if:
  – $p$ and $q$ are from different transactions
  – Operate on the same element
  – At least one of them is a write

• A conflict alone is not bad. Just means the order of operations is important
2. Serializable schedules

• A Serializable schedule is correct. It gives the same results as a serial schedule
• Goal is to make sure a schedule is “conflict-equivalent” to a serial schedule
• $S_1$ is conflict equivalent to $S_2$ means:
  – Same set of transactions in $S_1$ and $S_2$
  – Conflicts are ordered the same way in $S_1$ and $S_2$
• So a Serializable schedule orders conflicting operations same way a serial schedule does
Conflict-equivalent schedules

- Is this schedule conflict-equivalent to a serial schedule?

  $T_1 : R(x = 2)_{t}, \ x := x + 12, \ W(x = 14)_{t+2}$
  $T2 : \ R(x = 2)_{t+1}, \ x := x - 5, \ W(x = -3)_{t+3}$

- We can change the order of operations if they are not in conflict, and try to produce $T_1 \ T_2$ or $T_2 \ T_1$

  $T_1 : R(x = 2)_{t}, \ x := x + 12, \ W(x = 14)_{t+2}$
  $T2 : \ R(x = 2)_{t+1}, \ x := x - 5, \ W(x = -3)_{t+3}$
Conflict-equivalent schedules

• Back to the example. Neither $T_i$ can come before the other
  
  $T_1: \text{R}(x = 2)_{t}, \ x := x + 12, \ \text{W}(x = 14)_{t+2}$
  
  $T_2: \quad \text{R}(x = 2)_{t+1}, \ \ x := x - 5, \quad \text{W}(x = -3)_{t+3}$

• There is no legal swapping of operations
  – Schedule is not serializable

• A database scheduler should not allow such schedule
The precedence graph

• Easier way to test for serializability
• Nodes in the graph are transactions
• An edge from $T_i$ to $T_j$ means there is a conflicting pair of operations from both and the operation of $T_i$ comes first
The serializability theorem

• A schedule is serializable if its precedence graphic is acyclic (Gray, Lorie et al, 1976)

• Intuitively, an acyclic precedence graph can always be topologically sorted
  – Can always define an order as in a serial schedule
3. Scheduling approaches

• In a precedence graph with N nodes it takes $O(N^2)$ to test for cycles
• Overhead not acceptable in high-concurrency systems
  – Hundreds or thousands of concurrent requests per second
• Practical approaches to test for cycles were developed
Lock-based approaches

• Serialize access to data by locking it:
  – Read locks
  – Write locks

• An element can only have one lock if it is a write lock (called exclusive)

• An element can have any number of read locks (called shared)

• Transactions are well-behaved: always ask for locks
The lock-based scheduler

- A read (write) operation asks for a read (write) lock; if not granted, the transaction waits

   $T_1: R(x = 2)_{t}, x := x + 12, W(x = 14)_{t+2}$

   $T_2: R(x = 2)_{t+1}, x := x - 5, W(x = -3)_{t+3}$
The lock-based scheduler

• Incidentally, previous situation is a deadlock
  – No progress is possible

• Several options:
  – Kill any transaction
  – The one that came late
  – Use timers....

• Otherwise, a locking scheduler that keeps all locks to the end produces serializable schedules
2PL Theorem

• Locking in 2 phases (2PL):
  – A transaction never asks for more locks after it releases the first one

• The 2PL Theorem (Eswaran, 1976):
  – If all transactions obey the 2PL rule, then the schedule is serializable
2PL in practice

• DB2, MS SQL Server, MySQL use 2PL
• Performance is considered good
  – Depending on the transactions, the rate of interruptions or blocking can be high
• Read-only transactions are blocked
  – In OLAP settings this is waste of time
  – Web patterns reveal there are more readers than writers
Web applications

- Google Megastore: 23 billion transactions daily, more than 86% are read-only
- Yahoo: 95% of transactions to the cloud are read-only
- In general web stores, portals, exhibit similar numbers
- Can the blocking of readers be eliminated?
Multi-version schedulers

• Basic idea: use several versions of data, allowing readers to get appropriate versions
  – Avoids blocking of readers, always succeed
• Adopted in PostgreSQL, Oracle, SQL Server 2005, MySQL
• Simplest implementation uses Timestamp Ordering
Multi-version Timestamp Ordering

• Each transaction gets a start timestamp ($TS$)
• Each data has the $TS$ of the transaction that creates it ($W-TS$), and the $TS$ of the transaction that last read it ($R-TS$)
• Result looks like a mono-version schedule serialized by the $TS$ of the transactions
  \[- T_i T_j \text{ if } TS(T_i) < TS(T_j) \]
MVTO: reads

- A read operation $R(x)$ by $T$ is translated into $R(x_k)$ where $x_k$ is the last version such that $T_k$ started before $T$ started: $W-TS(x_k) < TS(T)$
- Sets $R-TS(x_k)$ to $TS(T)$
A write operation $W(x)$ by $T$ can have two outcomes:

- A transaction $T_i$ with $TS(T_i) > TS(T)$ read $x$. $T$ is interrupted
- Otherwise writes a new version of $T$ and sets $W-TS(x)$ to $TS(T)$

So, $T$ is interrupted if it would invalidate a read by an younger transaction
MVTO: example

- T₂ must be interrupted, invalidates T₃’s read

Should have read from T₂
MVTO: issues

• An element has potentially many versions
  – Some committed, some uncommitted
  – Some versions become obsolete and must be discarded

• Hard to get unique timestamps

• A read is a write (updates R-TS)

• There are also MVCC proposals with 2PL
  – Uses locks for update transactions, but locks were the problem....
Snapshot Isolation

- Proposed in Berenson et al (1995) as a variant of MVCC
- Not serializable, but acceptable behavior
- A transaction reads a snapshot of the data (committed versions only)
  - Concurrent updates are invisible
- Before committing, a transaction checks if any concurrent transaction wrote the same data
Snapshot Isolation

• Readers do not block writers and writers do not block readers
• Only checks for write-write conflicts
  – Writeset of concurrent transactions must be disjoint
  – Can be implemented without locks
Snapshot Isolation

• How to prevent lost updates?

• First committer Wins (FCW)
  – When committing \( W(x) \), check:
    • If there is a \( x \) in the writeset of a committed concurrent transaction, interrupt

• Slight variation is First Updater Wins (FUW):
  – Perform the previous test when wanting to write \( x \)
  – Avoid loosing work, wasting resources
SI: practical implementation

• Should allow for versioned reads and an easy way to check writesets
• Use versions from the recovery logs (MySQL, Oracle), tempdb (MS SQL Server) or keep multiple versions of tuples (non-overwriting storage, as in PostgreSQL)
• Check for write-write conflicts at update time (FUW rule) using tuple locks
Is SI anomaly-free?

- Dirty-read: no, the snapshot is committed
- Lost-update: no, uses FUW rule
- Fuzzy-read: if using the same snapshot, reads are repeatable
- Phantoms: no, if using the same snapshot

- Does snapshot isolation add new anomalies?
Read skew

• A transaction transfers an amount from account A to B
  – \( R_1(A) \) \( R_1(B) \) \( W_1(A - 5) \) \( W_1(B + 5) \)
• A transaction reads the total amount \( A + B \)
  – \( R_2(A) \) \( R_2(B) \) display \( (A + B) \)
• The following can happen
  – \( R_1(A) \) \( R_2(A) \) \( R_1(B) \) \( W_1(A - 5) \) \( W_1(B + 5) \) \( c_1 \) \( R_2(B) \)
Write skew

• Account $A = 70$, $B = 80$, if total < 0 withdrawal from any account rejected
• A transaction checks total and subtracts 100
  $- R_1(A) R_1(B) W_1(A - 100)$
• A transaction checks total and subtracts 100
  $- R_2(A) R_2(B) W_2(B - 100)$
• The total can be -50:
  $- R_1(A) R_1(B) R_2(A) R_2(B) W_1(A - 100) W_2(B - 100)$
Predicate write skew

- List of doctors on duty: there must be at least one on a given day
- A doctor checks and there are 2 on duty: updates to be off duty, but before committing
- Another doctor checks and there are 2 on duty: updates himself to be off duty
- Result: both doctors off duty
  - Because both read from the same snapshot and there are no write conflicts
Three transactions, one RO

- Accounts X = 0, Y = 0
- T1 deposits 20 in account Y
- T2 subtracts 10 from X. If X + Y < 0 suffers a penalty of 1
- T2 reads and prints X and Y

\[
\begin{align*}
R_2(X) \& R_2(Y) \& R_1(X) \& W_1(Y, 20) \& C_1 \& R_3(X) \& R_3(Y) \& C_3 \\
W_2(X, -11) \& C_2
\end{align*}
\]
What happened?

- Final: $X = -11, Y = 20$
- T3 printed $X = 0, Y = 20$, so must have followed T2
- So should have T2 T3 T1, but then the penalty in T1 was undue
- Conclusion: not serializable, should not happen
- Note that removing T3 it is serializable
Dependencies

• Two adjacent rw-dependencies around T2
  – Dangerous structure
• T2 (the pivot) should be interrupted
What are the implications?

• At least Oracle and PostgreSQL run Snapshot Isolation as their strict level of execution
• Even if anomalies are rare, there is concern about textbook definitions (e.g. serializability) and their implementations
  – Terminology gets confusing
• Queries must be carefully designed to avoid anomalies
Serializable SI

- Several attempts were made to ensure SI is serializable, thus anomaly-free
- Some approaches were static (design time)
- Most appealing were dynamic (run time)
  - SSI uses a Multiversion Serialization Graph (MVSG)
  - Similar to the monoverion SG, with the dependencies next slide
Types of dependencies

• An edge from $T_1$ to $T_2$ represents a:
  – WW dependency: $T_1$ writes $x$ and $T_2$ writes later $x$
  – WR dependency: $T_1$ writes $x$ and $T_2$ reads this or later version of $x$
  – RW dependency: $T_1$ reads $x$ and $T_2$ writes later $x$
    • Called an anti-dependency

• SSI tracks adjacent rw-dependencies: form a
  dangerous structure
The write skew example

- $R_1(A)$ $R_1(B)$ $R_2(A)$ $R_2(B)$ $W_1(A - 100)$ $W_2(B - 100)$
- There is a cycle in the graph with two RW edges
The general case to avoid

- There is a cycle, and $T_{\text{out}}$ is the first to commit
- $T_{\text{in}}$ and $T_{\text{out}}$ may be the same
Practical application

- D. Ports & K. Gritner (2012) implemented the first production ready SSI scheduler
  - on PostgreSQL 9.1
Practical application

• Minor changes to existing code (a couple of thousand lines)
• SI serialization anomalies lead to a “cannot serialize” error, detected by applications
• Standard benchmarks (TPC-C, SSBM) under testing
Practical implications

• Making the “serializable” level serializable, reduces errors and frees applications from checking for anomalies.

• A theoretical approach lead to a (probably) huge step forward in understanding SI.

• Revilak (2011) proposed in his PhD thesis *Precisely Serializable Snapshot Isolation*. 
  – Less false positives than SSI, but greater overhead.
References


References