Databases

dr. Jörg Verstraete

Polish Academy of Sciences
Systems Research Institute

jorg.verstraete@ibspan.waw.pl
http://www.mini.pw.edu.pl/~verstraetej/
Today

- Lecture
  - Transactional processing
    - what?
    - why?
    - how?
- Labs
  - Continuation of database design
Transactional processing

• Concept
  – Multiple DML statements (CRUD) should be considered atomic
    – e.g. transfer between accounts
      
      UPDATE Accounts set balance=balance-100 where account_id=456
      UPDATE Accounts set balance=balance+100 where account_id=8899

    • either both updates should happen, or no update should happen
    – Results of commands should be consistent
    – One batch should not affect another
ACID properties

• Data processing is referred to as transactional when ACID conditions are satisfied:
  – Atomicity
  – Consistency
  – Isolation
  – Durability

• Concepts
  – commit: accept all statements in the transaction
  – rollback: ignore all statements in the transaction
ACID properties

• Atomicity
  – Transactions, defined as a sequence of SQL statements, must be treated as a single unit
    • all statements committed
    • all statements rolled back

• Consistency
  – No matter the result of a transaction, the database must remain consistent
    • all constraints must remain valid
ACID properties

• Isolation
  – Unfinished transactions should not affect the outcome of other ongoing transactions
    • e.g. as long a student record is begin modified, no list of students can be created
    • DBMS offer different isolation levels to increase concurrency

• Durability
  – The result of a committed transaction must be persistent
    • changes to the data cannot be revoked or lost
Transactions

• Explicit
  – specified by
    • BEGIN TRANSACTION
    • COMMIT or ROLLBACK

• Implicit
  – specified by a single statement that modifies many records
    • e.g. UPDATE … WHERE ...
  – guaranteed to be performed as a transaction
Transactions

- **BEGIN TRANSACTION**
  - starts a transaction
- **COMMIT**
  - writes all the changes to the database
- **ROLLBACK**
  - undoes all the changes made in this transaction since the beginning of the transaction
- Additionally, in some RDBMS
  - nested transactions: transaction in a transaction
  - save points: can be used to undo part of a transaction
Example 1

• Explicit

BEGIN TRANSACTION
UPDATE Accounts set balance=balance-100 where account_id=456
UPDATE Accounts set balance=balance+100 where account_id=8899
COMMIT

• Implicit

UPDATE Students set faculty = 1 where studentID <4
Transactions

• ROLLBACK
  – automatically called if an error occurs
  – can be called explicitly to cancel the statements

• Affect DML
  – insert, update, delete

• DDL statements can be not part of a transaction
  – SQL Server: rollback removes a table created
  – Oracle: rollback does not remove a table
Transactions and DDL

- DDL statements should not be mixed with changes to the data (DML) in the same transaction
  - DDL should be executed as part of a planned development of the model
  - DML relates to data manipulations
Implementation of transactions

• No straightforward solutions
  – ACID properties must be strictly followed, in the event of a power failure / device failure
    • the result of a committed transaction should not be lost
      – durability
    • partial update is not allowed
      – atomicity
    • partial results cannot be used by other running transactions (they should be rolled back if not fully completed)
      – isolation
Implementation of transactions

- Transaction is submitted
  - Information on transaction is saved
    - Transaction log
      - helps to identify affected records
      - resolve pending transactions
        - in case of interruption
  - Database is updated
    - Database
      - may contain inconsistent data
        - in case of interruption
      - resolved using transaction log, before dbms service resumes
Implementation of transactions

- **Justification**
  - On average >99% of transactions are committed
  - For efficiency
    - perform actions
    - restore original data if necessary
- **Multi-user environment**
  - Multiple simultaneous transactions
  - Transaction isolation conflicts with throughput
DBMS and transactions

- Transactional processing
  - significant overhead
  - requires sophisticated solutions from DBMS
- Most commercial DBMS are transactional
  - Oracle, SQL Server, DB2, …
- Many open source DBMS do not support transactions
  - limited reliability for mission-critical applications
Today

• Lecture
  – Transactional isolation problems
  – concurrency
    • what? why?
Problem

- Concurrent transactions

Account 456, 1000 PLN

1500 PLN

1400 PLN

? 

BEGIN TRANSACTION
UPDATE Accounts
set balance = balance +500
where account_id = 456

ROLLBACK

BEGIN TRANSACTION
UPDATE Accounts
set balance = balance -100
where account_id = 456

COMMIT

Wrong value in database, first transaction should not have had a result saved in the database.
Transactions and concurrency

• Multiple transactions run in parallel
  – different users
  – different applications

• Transaction isolation is easy if there would be no concurrency
  – everything serialised
    • one ongoing transaction, rest waits
  – severely limits DBMS performance
    • a balance is needed
Transaction anomalies

- **Dirty reads**
  - occurs when data is read that is not committed
    - transaction 1 updates a value
    - transaction 2 reads this value
    - transaction 1 rolls back the update

- **Lost updates**
  - occurs when multiple transactions update
    - transaction 1 updates a value
    - transaction 2 updates the same value
      - before transaction 1 is committed
Transaction anomalies

• Phantom records
  - occurs when another transaction adds records that should be used in this transaction
    • transaction 1 selects records
    • transaction 2 adds a record that satisfies the query of 1
    • transaction 1 performs actions: it does not have the new record

• Non-repeateable read
  - occurs when the same query during a transaction has different results
Dealing with anomalies

- Sometimes, one is willing to accept partly incorrect data

- Modern DBMS allow to set a transaction isolation level
  - balance performance and isolation
    - set during a session, by a client
    - only for this session

- SQL Server provides several levels
  - syntax: SET TRANSACTION ISOLATION LEVEL ...
## SQL Server isolation levels

### Main levels

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty reads</th>
<th>Non-repeatable reads</th>
<th>Phantoms</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>No synchronisation between transactions</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>Only results of committed transactions are seen. They can be committed during other transactions, causing non-repeatable reads and phantoms</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>Transaction sees the content of the database as it was at the beginning</td>
</tr>
<tr>
<td>SNAPSHOT</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>Corresponds to full serial execution of transactions, data read in a transaction gets blocked.</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td></td>
</tr>
</tbody>
</table>
Mechanism

- Locking
  - record level
  - range of records
  - entire table

- During a query, DBMS checks if the records needed for the query are locked

- Can be consolidated: if e.g. 90% of the records need to be locked, the DBMS may chose to lock the entire table
  - not in Oracle
Locks

• Update locks
  • imposed by DMBS automatically
  • correspond to transaction isolation level
    – e.g. allow read access to a modified record
  - Shared
  - Exclusive
    – prevent all other access

• Schema locks
  - Schema stability
    • ensures that a DB object (table, index, … will not be removed)
  - Schema modification
    • prevents access to DB objects that are being modified
**Locks: example**

- **read committed**

| Account 456, 1000 PLN | BEGIN TRANSACTION  
|-----------------------|---------------------|
| 1500 PLN             | UPDATE Accounts     
|                      | set balance = balance +500 
|                      | where account_id = 456 
| 1500 PLN             | ROLLBACK            
| 1000 PLN             |                     
| 900 PLN              |                     |

The lock is released when the first transaction finishes.

BEGIN TRANSACTION  
UPDATE Accounts  
set balance = balance -100  
where account_id = 456  
COMMIT
Deadlocks

- A combination of locks that prevents more than one transaction from execution
  - cannot be resolved without terminating at least one of the transactions

- Terminate implies rollback
  - cancels all the updates made
  - not an acceptable situation
Deadlock: example

BEGIN TRANSACTION
UPDATE Accounts
set balance = balance+500
where account_id = 456

UPDATE Accounts
set balance = balance+100
where account_id = 789

COMMIT

BEGIN TRANSACTION
UPDATE Accounts
set balance = balance+500
where account_id = 789

UPDATE Accounts
set balance = balance+100
where account_id = 456

WAIT

WAIT

account 456 is locked

account 789 is locked
Deadlocks

- How to avoid?
  - Proper agreements in coding: always lock resources in same order
    - issue insert/update/delete statements in same sequence
    - e.g. lock record with lowest primary key first
    - e.g. lock tables accounts before table customers
How to monitor locks?

- SQL Server
  - Enterprise manager / SQL Server Management Studio
    - current activity subtree
  - stored procedure sp_lock
- As rule
  - Locks are released upon when transaction terminates: commit / rollback
How to monitor locks?

- **SQL Server**
  - Enterprise manager / SQL Server Management Studio
    - current activity subtree
  - stored procedure `sp_lock`
- **As rule**
  - Locks are released upon when transaction terminates: commit / rollback
Execution time

- **Standard short running transactions**
  - Typically less then a second
  - Sets locks on accessed resources
  - Majority of transactions

- **Long-running transactions**
  - e.g. Modification of the design
    - not based on locks, using logical copy of data
Concurrency issues

• Example 1: read uncommitted

BEGIN TRANSACTION

UPDATE customers
set ordercount = 2
where customerId = 'ALFKI'

'ALFKI' is locked

COMMIT / ROLLBACK

SET TRANSACTION ISOLATION
LEVEL READ UNCOMMITTED

BEGIN TRANSACTION

SELECT * from customers
where customerId = 'ALFKI'

NOT SUSPENDED
DIRTY READ POSSIBLE

COMMIT / ROLLBACK
Concurrency issues

• Example 2: read committed

BEGIN TRANSACTION
UPDATE customers
set ordercount = 2
where customerId = 'ALFKI'

'SALFKI' is locked

COMMIT / ROLLBACK

SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRANSACTION
SELECT * from customers
where customerId = 'ALFKI'
SUSPENDED
NO DIRTY READ

COMMIT / ROLLBACK
Concurrency issues

• Example 3: non-repeatable read

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRANSACTION
  UPDATE customers
  set ordercount = 2
  where customerId = 'ALFKI'
  NOT SUSPENDED
COMMIT / ROLLBACK
```

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRANSACTION
  SELECT * from customers
  where customerId = 'ALFKI'
  'ALFKI' is used
END TRANSACTION
SELECT * from customers
  where customerId = 'ALFKI'
  NON REPEATABLE READ POSSIBLE
COMMIT / ROLLBACK
```
Concurrent issues

- Example 4: repeatable read

```sql
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRANSACTION
  UPDATE customers
  set ordercount = 2
  where customerId = 'ALFKI'
COMMIT / ROLLBACK
```

```sql
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
  SELECT * from customers
  where customerId = 'ALFKI'
SUSPENDED
COMMIT / ROLLBACK
```

'ALFKI' is used
Locking mechanism

- Higher transaction isolation level
  - better protection against anomalies
  - lower concurrency (due to more locks)
  - more chances for deadlocks (due to more locks)
    - even in reads (see example of repeatable read)
Multi-versioning

• Aimed at improving concurrent access
• Multiple versions of data are considered
  – user A starts a transaction and modifies a table
  – user B keeps working with original table
• Consequences
  – reads will not be blocked by inserts/updates
  – transactions with only reads will never create a deadlock and will never return a non-existing answer
• Statement level or transaction level
  – depends on selected isolation level
Mechanism

- Each statement/transaction maintains its logical version of data
  - online investigation of rollback segments allows the DBMS to logically recreate the content of the database at any time in the past
    - limited by log size limitations
  - As long as a transaction A did not commit, other transactions see the data as it was before A started.
# Oracle Isolation levels

## Main levels

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty reads</th>
<th>Non-repeatable reads</th>
<th>Phantoms</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>Not needed. Multiversioning prevents dirty reads, blocking reads are never applied</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>Combined with multi-versioning, the data will be reconstructed as it appeared when the query started. <strong>Concurrency control is not based on locks: lost updates can happen!</strong></td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>Not needed, already fulfilled by READ COMMITTED.</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>Transaction sees the content of the database as it was at the beginning</td>
</tr>
<tr>
<td>READ ONLY</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>Entire transaction is read only</td>
</tr>
</tbody>
</table>
Transactions - Oracle

• Started by default
  – no begin transaction

• SET TRANSACTION ISOLATION LEVEL
  – can only be first statement

• RED COMMITTED
  – most frequently used
Concurrency issues

• Example 1: read committed

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

UPDATE customers
  set ordercount = 2
  where customerId = 'ALFKI'

'ALFKI' is locked

COMMIT / ROLLBACK

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

SELECT * from customers
  where customerId = 'ALFKI'

NOT SUSPENDED
  Reads data as if transaction 1 has not started yet.
Concurrency issues

• Example 2: repeatable read

```sql
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
UPDATE customers
set ordercount = 2
where customerId = 'ALFKI'
COMMIT / ROLLBACK

SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT * from customers
where customerId = 'ALFKI'
COMMIT / ROLLBACK

NON REPEATABLE READ IMPOSSIBLE

'ALFKI' is used
```
Concurrency issues

• Example 3: repeatable read

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED

INSERT into employees ....
```

- `new record`

```
COMMIT
```

```
SET TRANSACTION ISOLATION SERIALIZABLE

SELECT * from employees
```

- `NO NEW EMPLOYEE` returns the content committed at the start of the transaction
Multiversioning - Oracle

- **SERIALIZABLE**
  - assumes no changes will be made to that data that is changed in a parallel transaction
  - error message when an attempt is made to modify data changed by other transactions
    - “Can't serialize access for this transaction”
  - select ... for update can be applied to serialize access to some records

- **READ ONLY**
  - error message when rollback segments containing the required past state have been overwritten because of inadequate size of rollback segments
    - “Snapshot too old error”
    - should not happen on a properly configured production system
Concurrence issues

- **Example 4: write lock**

  ```sql
  SET TRANSACTION ISOLATION LEVEL ...

  UPDATE customers
  set ordercount = 2
  where customerId = 'ALFKI'

  COMMIT / ROLLBACK
  
  'ALFKI' is locked
  
  SET TRANSACTION ISOLATION LEVEL READ ...

  UPDATE customers
  set ordercount = 2
  where customerId = 'ALFKI'

  SUSPENDED
  Record has a write lock, as in other DBMS
  ```
Write consistency

- Modification statement submitted (insert/update/delete)
- Consistent reads are applied to determine scope (set of records satisfying WHERE at the time when modification was submitted)
- Each record in scope is read
  - modified record found?
    - no: changes are applied
    - yes:
      - READ COMMITTED level?
        - no: cannot serialize access error
        - yes: select for update is applied
          - records are selected again, but now records are locked
          - changes are applied
Write consistency

• Consequences
  – Triggers may be fired twice for a single UPDATE
    • before first dirty record was found
    • after rereading in select for update
  – Triggers should not be run in autonomous transactions
  – Significant overhead possible if data is frequently modified by different transactions
Conclusions

• Multi-versioning may negatively affect performance
  – e.g. long running transaction that reads a lot of data, some data may have been modified by a parallel transaction so rollback logs need to be investigated to ensure version consistency

• Locking mechanism still needed, built into records

• Every DBMS works differently
  – don't make assumptions based on other DBMS
  – check implementation details to avoid errors or bad performance
  – do not try to avoid transactional processing