OVERVIEW OF TRANSACTION MANAGEMENT

Tópicos Avançados da Base de Dados

Arley Pinto – up110370423
(andrapt@gmail.com)

Gabriel de Jesus – up110370572
(thejesusgaby@gmail.com)
• A **transaction** (“xact”) is the DBMS’s abstract view of a user program (or activity):
  
  - A sequence of **reads** and **writes** of database objects.
  - Unit of work that must **commit** or **abort** as an atomic unit.

• A user’s program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.

• Transaction Manager controls the execution of transactions.
**Transaction Management**

**ATOMICITY:** All actions in the Xact happen, or none happen.

**CONSISTENCY:** If each Xact is consistent, and the DB starts consistent, it ends up consistent.

**ISOLATION:** Execution of one Xact is isolated from that of other Xacts.

**DURABILITY:** If a Xact commits, its effects persist.
A very important property guaranteed by the DBMS for all transactions is that they are *atomic*. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.

- A transaction ends in one of two ways:
  - *commit* after completing all actions
  - *abort* after executing some actions
  - *system crash* while the xact is in progress; treat as abort.
DBMS ensures the above by logging all actions:

- **Undo** the actions of aborted/failed transactions
- **Redo** actions of committed transactions not yet propagated to disk when system crashes.
Transaction Management

- Transactions preserve DB *consistency*
  - Given a consistent DB state, produce another consistent DB state

- DB Consistency expressed as a set of declarative *Integrity Constraints*
  - *CREATE TABLE/ASSERTION* statements
    - E.g. Each student TABD can only register in one project group. Each group must have 2 students.

- Transactions that violate ICs are aborted
  - That’s all the DBMS can automatically check!
Transaction Management

CONCURRENCY IN DBMS

• DBMS interleaves actions of many xacts concurrently
  - Actions = reads/writes of DB objects

• DBMS ensures xacts do not “step onto” one another

• Each xact executes as if it were running by itself
  - Concurrent accesses have no effect on a Transaction’s behavior
Consider two transactions (*Xacts*):

- **T1:** BEGIN \( A = A + 100, \ B = B - 100 \) END
- **T2:** BEGIN \( A = 1.06 \times A, \ B = 1.06 \times B \) END

1st xact transfers $100 from B’s account to A’s

2nd credits both accounts with 6% interest.

Assume at first \( A \) and \( B \) each have $1000. What are the *legal outcomes* of running T1 and T2?

- **T1 ; T2** \( A = 1166, B = 954 \)
- **T2 ; T1** \( A = 1160, B = 960 \)

In either case, \( A + B = 2000 \times 1.06 = 2120 \)
Consider a possible interleaved schedule:

<table>
<thead>
<tr>
<th>T1: A=A+100, B=B-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>

This is OK (same as T1;T2). But what about:

<table>
<thead>
<tr>
<th>T1: A=A+100, B=B-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>

Result: A=1166, B=960; A+B = 2126, bank loses $6!

The DBMS’s view of the second schedule:

<table>
<thead>
<tr>
<th>T1: R(A), W(A), R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>
Transaction Management

**SCHEDULING**

- **Serial schedule:** no concurrency
  - Does not interleave the actions of different transactions.

- **Equivalent schedules:** same result on any DB state
  - For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

- **Serializable schedule:** equivalent to a serial schedule
  - A schedule that is equivalent to some serial execution of the transactions.
Some examples of *Xacts*:

- The CDR files reach the Mediation server, the Mediation System does the following actions on the CDRs:
  - Collection, Filtering, Conversion, Aggregation and Dumping
- Apply postpaid CDRs tariff plan for billing
- Customer Care (NGIN–Next Generation Intelligence Network)
- Credit Transfer between two Prepaid mobile clients
Transaction Management

CONCURRENCY IN DBMS (TELECOMMUNICATION)

Some examples:

• Running Tariff CDRs (or others services or jobs) when Sales transactions are in progress

• Read database using SELECT* FROM table (is prohibited in prepaid mobile’s database server enviroment) and not recommended for billing and mediation.

• Etc…
Transaction Management

SCHEDULING (TELECOMMUNICATION)

Some examples:

- Integrating Services for traffic Analyst, Billing and Fraud Management files.
- SIGC service
- Web Services for Simcard Creating, Activation, modification...
- Etc...
Two actions on the same data object conflict if at last one of them is a write. Three anomalous situations can be described:

1. **Reading Uncommitted Data (WR Conflicts)**
   A transaction $T_2$ read a database object $A$ that has been modified by another transaction $T_1$, which has not yet committed.

   \[
   \begin{array}{l}
   T_1: \quad R(A), W(A), \quad R(B), W(B), \text{Abort} \\
   T_2: \quad R(A), W(A), C
   \end{array}
   \]
2. Unrepeatable Reads (RW Conflicts)

A transaction T2 change the value of an object A that has been read by transaction T1, while T1 is still in progress.

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), R(A), W(A), C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A), C</td>
</tr>
</tbody>
</table>

3. Overwriting Uncommitted Data (WW Conflict)

A transaction T2 overwrite the value of an object A, which as already been modified by transaction T1, while T1 is still in progress.

<table>
<thead>
<tr>
<th>T1:</th>
<th>W(A), W(B), C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>W(A), W(B), C</td>
</tr>
</tbody>
</table>
A locking protocol is a set of rules to be followed by each transaction to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in serial order.

**Strict Two-phase Locking (Strict 2PL) Protocol**

- Each transaction must obtain an S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.

- Lock rules:
  - If an T holds an X lock on an object, no other T can acquire a lock (S or X) on that object;
  - If an T holds an S lock, no other T can get an X lock on that object.

- Deadlocks:
  - A simple way to identify deadlocks is to use a timeout mechanism.

<table>
<thead>
<tr>
<th>Lock type</th>
<th>read-lock</th>
<th>write-lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>read-lock</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>write-lock</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Log consists of “records” that are written sequentially.
- Ti writes an object: the old value and the new value.
- Ti commits/aborts: a log record indicating this action.

Log records are chained together by Transaction id.

Log is often duplexed and archived on stable storage.

Need for UNDO and/or REDO
- UNDO required if: uncommitted data can overwrite stable version of committed data;
- REDO required if: T can commit before all its updates are on disk.
There are 3 phases in the Aries recovery algorithm:

- **Analysis**: Scan the log forward to identify all Transactions that were active, and all dirty pages in the buffer pool.

- **Redo**: Repeats all actions and restores the database state to what it was at the time of the crash.

- **Undo**: Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.
Transaction Management

PostgreSQL Transactions

- BEGIN
- SAVEPOINT
- COMMIT
- ROLLBACK

Atomic (all-or-nothing)

- No explicit transaction:
  UPDATE utentes SET endereco = 'rua tal e qual';

- Explicit transaction:
  BEGIN WORK;
  UPDATE utentes SET endereco = 'Campo Alegre';
  COMMIT WORK;

Review with case study (Postgresql)
Transaction Management

POSTGRESQL's default isolation level, READ COMMITTED, allows you to see other transaction commits while your transaction is open.

<table>
<thead>
<tr>
<th>User 1</th>
<th>User 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM trans_test</td>
<td>Return 0</td>
<td></td>
</tr>
<tr>
<td>INSERT INTO trans_test VALUES(1)</td>
<td>Add row to trans_test</td>
<td></td>
</tr>
<tr>
<td>SELECT * FROM trans_test</td>
<td>Return 1</td>
<td></td>
</tr>
<tr>
<td>SELECT * FROM trans_test</td>
<td>Return 1</td>
<td></td>
</tr>
</tbody>
</table>

POSTGRESQL's default isolation level, READ COMMITTED, allows you to see other transaction commits while your transaction is open.

```
BEGIN WORK;
    SELECT COUNT(*) FROM trans_test;
--
    -- someone commits INSERT INTO trans_test
    --
    SELECT COUNT(*) FROM trans_test;
COMMIT WORK;
```

Result = 5

Result = 6
Visibility of transactions
You can prevent your transaction from seeing changes made to the database.

BEGIN WORK;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT COUNT(*) FROM trans_test;
--
-- someone commits INSERT INTO trans_test
--
SELECT COUNT(*) FROM trans_test;
COMMIT WORK;

SERIALIZABLE isolation provides a stable view of the database for SELECT transactions. For transactions containing UPDATE and DELETE queries, SERIALIZABLE mode is more complicated. If two concurrent transactions attempt to update the same row, serializability is impossible. In such a case, POSTGRESQL forces one transaction to roll back.
Transaction Management

The log

In PostgreSQL, when a row is updated, a new version (called a tuple) of the row is created and inserted into the table.

Each tuple has additional data recorded with it xmin and xmax:
The log

To track the status of transactions, a special table called PG_LOG is maintained.

Contains two bits of status information for each transaction: the possible states are in progress, committed, or aborted.

A Vacuum cleaner process is provided to garbage collect expired/aborted versions of a row.
Database Management System
http://en.wikipedia.org/wiki/Database_management_system

Transaction Management Overview – Presentation Timothy Leary

A Quick Survey of MultiVersion Concurrency Algorithms
Dibyendu Majumdar, 2006

Postgresql – Transaction Manager
http://issuu.com/gshguru/docs/postgre_transaction_management

PostgreSQL 8.4.11 Documentation
http://www.postgresql.org/docs/8.4/static/tutorial-transactions.html

PostgreSQL Concurrency With MVCC
http://devcenter.heroku.com/articles/postgresql-concurrency