Part VIII

Transactions, Integrity and Triggers
Transactions, Integrity and Triggers

1. Basic Terms
2. Term Transaction
3. Transactions in SQL
4. Integrity Constraints in SQL
5. Trigger
Learning goals for today . . .

- Understanding of fundamentals of integrity control in databases
- Knowledge to formalize and implement integrity constraints
- Knowledge of the transaction concept in databases
Integrity

- **Integrity constraint** (*also: assertion*): Condition for the "permissibility" or "correctness"

  with respect to databases:
  - (single) database states,
  - state transitions from an old to a new database state,
  - long term database evolution
## Classification of Integrity

<table>
<thead>
<tr>
<th>Constants Class</th>
<th>Temporal Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>database state</td>
</tr>
<tr>
<td>dynamic</td>
<td>transitional</td>
</tr>
<tr>
<td></td>
<td>temporal state</td>
</tr>
<tr>
<td></td>
<td>sequence</td>
</tr>
</tbody>
</table>
Inherent Integrity Constraints in the RM

1. **Type Integrity:**
   - SQL allows domain definitions for a range of values for attributes
   - Permission or forbidding of null values

2. **Key Integrity:**
   - Specification of a key for a relation

3. **Referential Integrity:**
   - Specification of foreign keys
Example Scenarios

- Seat reservation for flights simultaneously from multiple travel agencies
  → Seat could be sold multiple times when multiple travel agencies identify the seat as available
- Overlapping account operations of a bank
- Statistics database operations
  → results are corrupted when data is changed during the calculation
Transaction

A transaction is a sequence of operations (actions) that transforms the database from a consistent state into a consistent, possibly changed, state, while the ACID-principle must be hold.

Aspects:
- Semantic Integrity: Correct (consistent) DB-state after a transaction has finished
- Operational Integrity: Prevent fault caused by "simultaneous" access of multiple users on the same data
ACID-Properties

- **Atomicity:**
  Transaction is executed completely or not at all

- **Consistency:**
  Database is before the start and after the end of a transaction in a consistent state

- **Isolation:**
  User, who is working on a database, should have the impression that she works alone on the database

- **Durability (Persistence):**
  The result of transaction has to be saved "permanently" in a database after the transaction competed successfully
Commands of a Transaction Language

- **Begin of a transaction**: Begin-of-Transaction-Command `BOT` (implicit in SQL!)
- **commit**: the transaction should try to finish successfully
  - success is not guaranteed!
- **abort**: the transaction has to be aborted
Transaction: Integrity Violation

- Example:
  - Transfer of an amount $A$ from a household post $K1$ to another post $K2$
  - Condition: Sum of the account balances stays constant

- Simplified notation
  \[
  \text{Transfer} = < K1:=K1-A; K2:=K2+A >;
  \]

- Realization in SQL: as sequence of two elementary changes
  Condition is not necessarily fulfilled between single changing steps!
Transaction: Behavior at System Crash
Transaction: Behavior at System Crash /2

**Consequences:**
- Contents of the volatile memory at the time $t_f$ is unusable → transactions in different ways affected by this

**Transaction states:**
- Still active transactions at the time of the failure ($T_2$ and $T_4$)
- Already finished transactions before the time of the failure ($T_1$, $T_3$ and $T_5$)
Simplified Model for Transactions

- Representation of database changes of a transaction
  - \texttt{read}(A,x): assign the value of the DB-object $A$ to the variable $x$
  - \texttt{write}(x, A): save the value of the variable $x$ in the DB-object $A$

- Example of a transaction $T$:

  \begin{verbatim}
  read(A, x); x := x - 200; write(x, A);
  read(B, y); y := y + 100; write(y, B);
  \end{verbatim}

- Execution variants for two transactions $T_1$, $T_2$:
  - serially, e.g. $T_1$ before $T_2$
  - "mixed", e.g. alternating steps of $T_1$ and $T_2"
Problems with Multi-User Operation

- Nonrepeatable Read
- Dependencies on not released data: Dirty Read
- The Phantom-Problem
- Lost Update
Nonrepeatable Read

Example:

- Assurance $x = A + B + C$ at the end of transaction $T_1$
- $x, y, z$ are local variables
- $T_i$ is the transaction $i$
- Integrity conditions $A + B + C = 0$
Example for Nonrepeatable Read

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{read}(A, x)$;</td>
<td>$\text{read}(A, y)$;</td>
</tr>
<tr>
<td>$\text{read}(B, y)$;</td>
<td>$y := y/2;$</td>
</tr>
<tr>
<td>$x := x + y;$</td>
<td>$\text{write}(y, A);$</td>
</tr>
<tr>
<td>$\text{read}(C, z);$</td>
<td>$\text{read}(C, z);$</td>
</tr>
<tr>
<td>$x := x + z;$</td>
<td>$z := z + y;$</td>
</tr>
<tr>
<td>commit;</td>
<td>$\text{write}(z, C);$</td>
</tr>
<tr>
<td></td>
<td>commit;</td>
</tr>
</tbody>
</table>
Dirty Read

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A, x$);</td>
<td>read($A, x$);</td>
</tr>
<tr>
<td>$x := x + 100$;</td>
<td>read($B, y$);</td>
</tr>
<tr>
<td>write($x, A$);</td>
<td>$y := y + x$;</td>
</tr>
<tr>
<td>abort;</td>
<td>write($y, B$);</td>
</tr>
<tr>
<td></td>
<td>commit;</td>
</tr>
</tbody>
</table>
## The Phantom-Problem

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
</table>
| **select count 
(*) into X
from Customer;**                         | **insert
into Customer
values ('Meier', 0,...);**                   |
| **update Customer
set Bonus = Bonus +10000/X;**         | **commit;**                                |
| **commit;**                               |                                           |
# Lost Update

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read(A, x);</code></td>
<td><code>read(A, x);</code></td>
<td>10</td>
</tr>
<tr>
<td>$x := x + 1;$</td>
<td>$x := x + 1;$</td>
<td>10</td>
</tr>
<tr>
<td><code>write(x, A);</code></td>
<td><code>write(x, A);</code></td>
<td>11</td>
</tr>
<tr>
<td><code>read(A, x);</code></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>$x := x + 1;$</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td><code>write(x, A);</code></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td><code>read(A, x);</code></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>$x := x + 1;$</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td><code>write(x, A);</code></td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>
Serializability

An interleaved execution of multiple transactions is called **serializable**, if its effect is identical to the effect of a (arbitrarily chosen) serial execution of these transactions.

- **Problem for checking serializability:**
  - there are $n!$ different serial execution orders for $n$ transactions...

- **Schedule:** Plan of execution for transactions (ordered list of transaction operations)
Transactions in SQL-DBS

Weakening of ACID in SQL: Isolation levels

```sql
set transaction
  [ { read only | read write }, ]
[isolation level
  { read uncommitted |
    read committed |
    repeatable read |
    serializable }, ]
[ diagnostics size ...]
```

Default settings:

```sql
set transaction read write,
  isolation level serializable
```
Meaning of Isolation Levels

- **read uncommitted**
  - weakest level: access to not committed data, only for read only transactions
  - statistic and similar transactions (approximate overview, incorrect values possible)
  - no locks → efficient executable, other transactions are not hindered

- **read committed**
  - only read finally written values, but nonrepeatable read possible

- **repeatable read**
  - no nonrepeatable read, but phantom-problem can occur

- **serializable**
  - guarantees serializability
### Isolation Levels: read committed

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>set transaction isolation level read committed</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>select Name from WINES where WineID = 1014</code></td>
<td><code>update WINES set Name = 'Riesling Superiore'</code></td>
</tr>
<tr>
<td></td>
<td>→ <em>Riesling</em></td>
<td><code>where WineID = 1014</code></td>
</tr>
<tr>
<td>3</td>
<td><code>select Name from WINES where WineID = 1014</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ <em>Riesling</em></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><code>commit</code></td>
</tr>
<tr>
<td>5</td>
<td><code>select Name from WINES where WineID = 1014</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ <em>Riesling Superiore</em></td>
<td></td>
</tr>
</tbody>
</table>
## Isolation Levels: read committed /2

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>set transaction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>isolation level</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read committed</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>select Name from WINES</td>
<td>update WINES</td>
</tr>
<tr>
<td></td>
<td>where WineID = 1014</td>
<td>set Name = 'Riesling Superiore'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where WineID = 1014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>blocked</td>
</tr>
<tr>
<td>3</td>
<td>update WINES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set Name = 'Superiore Riesling'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>where WineID = 1014</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-&gt; blocked</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>
### Isolation Levels: `Serializable`

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>set transaction isolation level serializable</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>select Name into N from WINES where WineID = 1014</code></td>
<td><code>update WINES</code></td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$ N := <code>Riesling</code></td>
<td><code>set Name = 'Riesling Superior'</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>where WineID = 1014</code></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><code>commit</code></td>
</tr>
<tr>
<td>5</td>
<td><code>update WINES</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>`set Name = 'Superior'</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>where WineID = 1014</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$ [Abort]</td>
<td></td>
</tr>
</tbody>
</table>
Integrity Constraints in SQL-DDL

- **not null**: Null values prohibited
- **default**: Specification of default values
- **check** (search-condition): Attribute specific constraint (usually One-Tuple-Integrity-Condition)
- **primary key**: Specification of a primary key
- **foreign key** (Attribute(e))
  - **references** Table( Attribute(e) ): Specification of the referential integrity
Integrity Constraints: Range of Values

- **create domain**: Establishing of a user defined range of values

  **Example**

  ```sql
  create domain WineColor varchar(4)
  default 'Red'
  check (value in ('Red', 'White', 'Rose'))
  ```

- **Application**

  ```sql
  create table WINES (  
    WineID int primary key,
    Name varchar(20) not null,
    Color WineColor,
    ...)
  ```
Integrity Constraints: check-Clause

- **check**: Establishing of further local integrity constraints within the defined range of values, attributes and relational scheme

- Example: Restriction of permitted values

- Example

```sql
create table WINES (  
    WineID int primary key,  
    Name varchar(20) not null,  
    Year int check(Year between 1980 and 2010),  
    ...  
)
```
Preservation of Referential Integrity

- Checking of foreign keys after database changes
  - for $\pi_A(r_1) \subseteq \pi_K(r_2)$,
    - e.g. $\pi_{\text{Vineyard}}(\text{WINES}) \subseteq \pi_{\text{Vineyard}}(\text{PRODUCER})$
      - Tuple $t$ is inserted into $r_1 \Rightarrow$ check, whether $t' \in r_2$ exists with:
        $t'(K) = t(A)$, d.h. $t(A) \in \pi_K(r_2)$
        if not $\Rightarrow$ reject
      - Tuple $t'$ is removed from $r_2 \Rightarrow$ check, whether $\sigma_{A=t'(K)}(r_1) = \emptyset$, i.e.
        no tuple from $r_1$ references $t'$
        if not empty $\Rightarrow$ reject or remove tuple from $r_1$, that reference $t'$ (at cascading deletion)
Checking Modes of Constraints

- **on update | delete**
  Specification of a triggering event that starts the checking of the condition

- **cascade | set null | set default | no action**
  **Cascading**: Handling of some integrity violations propagates over multiple levels, e.g. deletion as reaction on a violation of the referential integrity

- **deferred | immediate** sets the checking time for a condition
  - **deferred**: put back to the end of the transaction
  - **immediate**: immediate verification at any relevant database change
Checking Modes: Example

- Cascading deletion

```sql
create table WINES (  
    WineID int primary key,  
    Name varchar(50) not null,  
    Price float not null,  
    Jahr int not null,  
    Vineyard varchar(30),  
    foreign key (Vineyard) references PRODUCER (Vineyard)  
    on delete cascade)
```
The assertion-Clause

- **Assertion**: Predicate expressed by a condition that always has to be fulfilled by a database
- **Syntax (SQL:2003)**
  
  ```sql
  create assertion name check ( predicate )
  ```

- **Example**:

  ```sql
  create assertion Prices check
  ( ( select sum (Price) 
      from WINES ) < 10000 )
  ```

  ```sql
  create assertion Prices2 check
  ( not exists ( 
    select * from WINES where Price > 200 ) )
  ```
Trigger

- Trigger: Statement/Procedure that is executed automatically by the DBMS at the occurrence of a specific event

- Application:
  - Enforcement of integrity conditions ("implementation" of integrity rules)
  - Auditing of DB-actions
  - Propagation of DB-changes

- Definition:

```sql
create trigger ... 
after <Operation>
<Procedure>
```
Example for Triggers

- Realization of a calculated attribute with two triggers:
  - Introduction of new tasks

```sql
create trigger TaskCounter+
on insertion of Task A:
update Customer
set NrTasks = NrTasks + 1
where CName = new A.CName
```

- Analogously for deletion of tasks:

```sql
create trigger TaskCounter-
on deletion ...
update ... - 1 ...
```
Trigger: Design and Implementation

- Specification of
  - Event and condition for activation of the trigger
  - Action(s) for the execution

- Syntax in SQL:2003 defined

- Available in most commercial systems (but with different syntax)
SQL:2003-Trigger

**Syntax:**

```sql
create trigger <Name:>
after | before <Event>
on <Relation>
[ when <Condition> ]
begin atomic < SQL-statements > end
```

**Event:**

- `insert`
- `update [ of <list of attributes> ]`
- `delete`
Further Specifications for Triggers

- **for each row resp. for each statement**: Activation of the trigger for each single change of a set-valued change or just once for the whole change
- **before resp. after**: Activation before or after the change
- **referencing new as resp. referencing old as**: Binding of a tuple variable on the new introduced resp. just removed ("old") tuple of a relation
  \( \rightarrow \) tuple of the difference relation
Example for Triggers

- *No customer account can fall below 0:*

```sql
create trigger bad_account
after update of Acc on CUSTOMER
referencing new as INSERTED
when (exists
    (select * from INSERTED where Acc < 0)
)
begin atomic
    rollback;
end
```

⇒ similar trigger for insert
Example for triggers /2

*Producers must be removed, if they do not offer any wine:*

```sql
create trigger useless_Vineyard
after delete on WINES
referencing old as o
for each row
when (not exists
  (select * from WINES W
    where W.Vineyard = o.Vineyard))
begin atomic
  delete from PRODUCER where Vineyard = o.Vineyard;
end
```
Integrity Enforcement with Triggers

1. Specify object $o_i$, for which the condition $\phi$ should be monitored
   - Usually monitor multiple $o_i$ when condition is across relations
   - Candidates for $o_i$ are tuples of the relation names that occur in $\phi$

2. Specify the elemental database changes $u_{ij}$ on objects $o_i$ that can violate $\phi$
   - Rules: e.g., check existence requirements on deletion and updates, but not on insertion etc.
Integrity Enforcement with Triggers /2

3. Specify, depending on the application, the reaction $r_i$ on the integrity violation
   - Reset the transaction (rollback)
   - Correcting database changes

4. Formulate following triggers:

   ```
   create trigger t-phi-ij after u_{ij} on o_i
   when \neg \phi
   begin r_i end
   ```

5. If possible, simplify the created trigger
Trigger in Oracle

- Implementation in PL/SQL
- Notation

```
create [ or replace ] trigger trigger-name
   before | after
   insert or update [ of columns ]
   or delete on table
   [ for each row
   [ when ( predicate ) ] ]
PL/SQL-Block
```
Trigger in Oracle: Types

- **Statement level trigger**: Trigger is triggered before resp. after the DML-statement.
- **Row level trigger**: Trigger is triggered before resp. after each single modification (*one tuple at a time*)

**Trigger on row level:**

- **Predicate for restriction** (*when*)
- **Access on old** (:old.col) resp. **new** (:new.col) tuple
  - for **delete**: only (:old.col)
  - for **insert**: only (:new.col)
  - in **when**-clause only (:new.col) resp. (:old.col)
Trigger in Oracle /2

- Transaction abortion with \texttt{raise\_application\_error}(\textit{code, message})
- Distinction of the type of the DML-statement

\begin{verbatim}
if deleting then ... end if;
if updating then ... end if;
if inserting then ... end if;
\end{verbatim}
Trigger in Oracle: Example

No customer account can fall below 0:

create or replace trigger bad_account
after insert or update of Acc on Customer
for each row
when (:new.Acc < 0)
begin
    raise_application_error(-20221,
        'Not below 0');
end;
Summary

- Enforcement of correctness resp. integrity of the data
- Inherent integrity constraints of the relational model
- Additional SQL-integrity constraints: check-clause, assertion-statement
- Trigger for "implementation" of integrity constraints resp. rules
Control Questions

- What is the purpose of integrity enforcement? Which types of integrity constraints are there?
- How can integrity constraints and rules be formulated in SQL systems?
- What requirements result from the ACID-principle? How are these achieved in database systems?