Active Databases

General principles of Conventional Database Systems

SQL Schema Definitions

SQL Queries and Updates → DBMS → Query Results
**Conventional (Passive) Databases**

- Data model, usually relational
- Transaction model
  - *Passive* update principle - client controls DBMS updates

Example of real world problem not well suited for passive update principle:

- Inventory control
  - reordering items when quantity in stock falls below threshold.
- Travel waiting list
  - book ticket as soon as right kind is available
- Stock market
  - Buy/sell stocks when price below/above threshold
Conventional Databases

Passive DBMS

- Periodical polling of database by application
  - Frequent polling => expensive
  - Infrequent polling => might miss the right time to react
- The polling has to be done for all items in stock and can be expensive.
- Problem is that DBMS does not know that application is polling.
Active Databases

Active DBMS

- Recognize predefined *situations* in database
- *Trigger* predefined actions when situations occur

Actions are usually database updates, not calls to external programs to e.g. order items.
Active Databases

General idea
- ADBMS provides:
  - Regular DBMS primitives
  - definition of application-defined situations
  - triggering of application-defined reactions
Active Databases

Can be used for computation of derived data

*View materialization* of derived data

  e.g. incremental recomputation of view of sum of salaries per department, `salsum(dno,total)`, computed from `employee(ssn,dno,salary)`

  or invalidation of materialized view when relevant update (e.g. salary) occurs

  => Rematerialize view when accessed next time if materialized view invalid

In Oracle materialized views can be specified declaratively as

```sql
create materialized view salsum
as select dno, sum(salary) as total
from employee
```

However, this is not standard SQL! Triggers provide an alternative.
Active Databases

Semantics of ECA rules

• Most common model presently
• Event Condition Action:
  
  **WHEN** event occurs
  Usually update of single row in database table)

  **IF** condition holds
  Usually SQL query joining the triggered row with database table.
  Condition is considered true if query returns non-empty result

  **DO** execute action
  Usually SQL update statements or call to stored procedure
  referencing the updated row
Active Databases

- Example, no condition part (EA-rule), SQL-99

\[
\text{EMPLOYEE(\text{SSN, DNO, SALARY})}
\]

\[
\text{SALSUM(DNO, TOTAL) <= Materialized from EMPLOYEE}
\]

\[
\text{CREATE TRIGGER EMPLOYEE\_SALARY\_MATERIALIZATION}
\]

\[
\text{AFTER UPDATE OF SALARY ON EMPLOYEE<--- Event}
\]

\[
\text{REFERENCING NEW ROW AS NROW, OLD ROW AS OROW}
\]

\[
\text{FOR EACH ROW <--- per single updated row}
\]

\[
\text{BEGIN <--- Action}
\]

\[
\text{UPDATE SALSUM S}
\]

\[
\text{SET TOTAL = TOTAL - OROW.SALARY FROM OROW WHERE S.DNO = OROW.DNO}
\]

\[
\text{UPDATE SALSUM S}
\]

\[
\text{SET TOTAL = TOTAL + NROW.SALARY FROM NROW WHERE S.DNO = NROW.DNO}
\]

\[
\text{END}
\]

This does not cover changes to DNO! More triggers may be needed.
Active Databases (ECA)

- **Event**: Update of a single database record
  Parameterized using pseudo tables with a single row (added, updated, or deleted) specified by REFERENCING clause.

- **Condition**: Query on database state,
  e.g. a database query
  - empty result => condition is FALSE
  - non-empty result => condition is TRUE

- **Action**: Database update statement(s)
  Call stored procedure

- Unconditioned (EA) rules, as in example:
  ON ... DO

- Condition/Action (CA) rules
  Not used in databases
  Difficult to identify situation when rule triggered both for user and DBMS.
Active Databases

- Example of triggers (ECA) for maintaining constraints, SQL:99:
  Department table with number and manager’s SSN:
  \[
  \text{DEPARTMENT (DNO, MGRSSN)}
  \]

  \[
  \text{CREATE TRIGGER SALARY\_SITUATION1}
  \]

  \[
  \text{AFTER UPDATE OF SALARY ON EMPLOYEE--- Event}
  \]

  \[
  \text{REFERENCING NEW ROW AS NROW, OLD ROW AS OROW}
  \]

  \[
  \text{FOR EACH ROW --- C and A per updated row}
  \]

  \[
  \text{IF NROW.SALARY > (SELECT M.SALARY --- Condition}
  \]

  \[
  \text{FROM EMPLOYEE M, DEPARTMENT D, NROW}
  \]

  \[
  \text{WHERE}
  \]

  \[
  \text{NROW.DNO = D.DNO AND}
  \]

  \[
  \text{D.MGRSSN = M.SSN)}
  \]

  \[
  \text{THEN BEGIN --- Action}
  \]

  \[
  \text{UPDATE EMPLOYEE E}
  \]

  \[
  \text{SET SALARY = OROW.SALARY*0.9 FROM OROW}
  \]

  \[
  \text{END}
  \]
Active Databases

- NOTICE! `SALARY_CONSTRAINT` needed for managers:

```sql
CREATE TRIGGER SALARY_SITUATION2
AFTER UPDATE OF SALARY ON EMPLOYEE
REFERENCING NEW ROW AS NROW, OLD ROW AS OROW
FOR EACH ROW
IF NROW.SALARY < (SELECT E.SALARY
    FROM EMPLOYEE E, DEPARTMENT D, NROW
    WHERE E.DNO = D.DNO AND
    D.MGRSSN = NROW.SSN )
THEN
ROLLBACK
```

- NOTICE! `SALARY_SITUATION3` needed for departments too in case employee promoted to manager!

- Possible catch-all solution: Integrity constraints, *assertions*. 
Active Databases

• Advanced level SQL:99 has assertions too:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK(NOT EXISTS
    (SELECT *
    FROM EMPLOYEE E, EMPLOYEE M,
        DEPARTMENT D
    WHERE E.SALARY > M.SALARY AND
        E.DNO = D.DNO AND
        D.MGRSSN = M.SSN))
```

**NOTICE:** Advanced assertions may not be supported by the DBMS or may be implemented very inefficiently! Check manual for when they are efficient.

Naive implementation would check above constraint after each update to any of the tables EMPLOYEE or DEPARTMENT, which is very inefficient (does not scale).

Assertions cannot make different compensating actions depending on situation, as triggers can!
Active Databases

Cautions:

- Very powerful mechanism:
  Small statement => massive behavior changes.
  Rope for programmer.
  Requires careful design
- Trace consequences of rule specification/changes.
  Make sure indefinite triggering or undesired cascading triggering cannot happen.
- Avoid using triggers unless really needed.
  Use queries, view materialization statements, referential integrity constraints, or stored procedures instead if possible.
Active Databases

SUMMARY

- Active DBMSs embed situation-action rules in database
- Support many functionalities:
  - E.g. Integrity control, derived data, change notification
  - ADBMS functionality commercially available in SQL:99 as *triggers*: