“Capítol 5: SQL Avançat”

Fitxers i bases de dades

Capítol 5: SQL Avançat

- Accedint SQL des d'un Llenguatge de Programació
  - SQL dinàmic
    - JDBC i ODBC
  - SQL incrustat
- SQL Data Types ans Schemas
- Functions and Procedural Constructs
- Triggers
- Advanced Aggregation Features
- OLAP

JDBC i ODBC

- API (application-program interface) d'un programa per interactuar amb el servidor de la base de dades
- L'aplicació pot cridar a
  - Connectar-se amb el servidor de la base de dades
  - Enviar comandes SQL al servidor
  - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) treballa amb C, C++, C# i Visual Basic
  - Altres API's com DO.NET sit on top of ODBC
- JDBC (Java Database Connectivity) treballa amb Java

JDBC

- JDBC és una API Java per communicar amb els suports SQL dels sistemes de bases de dades.
- JDBC suporta una varietat de característiques per cercar i actualitzar dades, i per recuperar resultats de consultes.
- JDBC també suporta el rescat de metadades, tlas com consultes sobre relacions presents a la base de dades i els noms i tipus de relacions d'atributs.
- Model de comunicació amb la base de dades:
  - Obrir una connexió
  - Crear un objecte "statement"
  - Executar consultes fent servir l'objecte Statement per enviar-les i portar resultats
  - Mecanismes d'excepcions per suportar errors

Codi JDBC

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
          ... Do Actual Work ...
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle){
            System.out.println("SQLException:"+ sqle);
    }
}
```

Codi JDBC (Cont.)

- Actualitzar la base de dades

```
try {
    stmt.executeUpdate(
        "insert into instructor values ('77987', 'Kim', 'Physics', 98000)");
} catch (SQLException sqle)
{
    System.out.println("Could not insert tuple." + sqle);
}
```

- Executar, cercar, portar i mostrar resultats

```
ResultsSet rset = stmt.executeQuery(
                    "select dept_name, avg (salary)
                    from instructor
                    grouped bt dept_name");
while (rset.next()){
    System.out.println(rset.getString("dept_name")+" "+
                        rset.getFloat(2));
}
```

Detalls del Codi JDBC

- Obtenció dels camps de resultat:
  - rs.getString("dept_name") **and** rs.getString(1)

    equivalent si dept_name és el primer argument del resultat seleccionat.
- Tractant amb valors nuls
  - int a = rs.getInt("a");
  - if (rs.wasNull()) Systems.out.println("Got null value");

Declaració preparada

- PreparedStatement pStmt = conn.prepareStatement(
  "insert into instructor values (?,?,?,?)");
  pStmt.setString(1,"88877"); pStmt.setString(2,"Perry");
  pStmt.setString(3,"Finance"); pStmt.setString(4,125000);
  pStmt.executeUpdate();
  pStmt.setString(1,"88878");
  pStmt.executeUpdate();

- Per consultes, utilitzar pStmt.executeQuery(), el qual retorna un ResultSet

- Atenció: Fer servir sempre declaracions preparades when taking an input from the user and adding it to a query
  - MAI crear una consulta a partir de concatenar caracters aconseguits com a inputs
  - "insert into instructor values('"+ ID +"', '"+ name +"', "+" '+ dept name +"','"balance+")"
  - Què passa si el nom és "D'Souza"?

SQL Injection

- Suposem una consulta construïda fent servir
  - "select * from instructor where name = '"+ name +"'"
- Suposem que l'usuari, en comptes d'entrar un nom, escriu:
  - X' o 'Y' = 'Y
- Aleshores el resultat de la instrucció és:
  - "select * from instructor where name = '"+ "X' o 'Y' = 'Y" +"'"
  - Que és:
    - select * from instructor where name = 'X' o 'Y' = 'Y'
  - L'usuari podria haver fet servir
    - X'; update instructor set salary = salary + 10000; –
- Internament la declaració preparada fa servir:

  "select * from instructor where name = 'X\' o \'Y\' = \'Y'
  - Fer servir sempre declaracions preparades, amb els inputs d'usuaris com a paràmetres

Metadata Features

- ResultSet metadata
- Ex.: deprés d'executar una consulta per aconseguir un ResultSet rs:
  - ResultSetMetaData rsmd = rs.getMetaData();
    for(int i = 1; i <= rsmd.getColumnCount(); i++){
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumntypeName(i));
    }
- Com és això d'útil?

Metadata (Cont.)

- Database metadata
- DatabaseMetaData dbmd = conn.getMetaData();
  ResultsSet rs = dbmd.getColumns(null, "univdb", "department", "%");
  \\ hola

  Arguments to getColumns: Catalog, Schema-pattern, Table-pattern, and Column-Pattern

  Returns: One row for each column; row has a number of attributes such as COLUMN_NAME, TYPE_NAME

  ```
  while (rs.next()){
      System.out.print(rs.getString("COLUMN_NAME"),
                                   rs.getString("TYPE_NAME");
  }
  ```
- I on és això útil?

Control Transaccional en JDBC

- Per defecte, cada declaració d'SQL és tractada com una transacció separada que és encarregada automàticament
  - Mala idea per transaccions amb actualitzacions múltiples
- En una connecció es pot apagar l'automatic commit
  - conn.setAutoCommit(false);
- Transactions must then be committed or rolled back explicity
  - conn.commit(); o
  - conn.rollback();
- conn.setAutoCommit(true) engega automatic commit.

Other JDBC Features

- Cridant funcions i procediments
  - CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");
  - CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)}");
- Manipulant tipus d'objectes grans
  - getBlob() i getClob() són similars al mètode getString(), però retornen objectes de tipus Blob i Clob, respectivament
  - S'aconsegueixen dades d'aquests objectes mitjançant getBytes()
  - associate an open stream with Java Blob or Clob object to update large objects
    - blob.setBlob(int parameterIndex, InputStream inputStream).

SQLJ

- El JDBC és excessivament dinàmic, els errors no es podren trobar compilant
- SQLJ: Assigna SQL a Java
  - #sql iterator deptInfolter (String dept name, int avgSal);

    deptInfolter iter = null;

    #sql iter = {select dept_name, avg(salary) from instructor

    group by dept name};

    while (iter.next())}

    String deptName = iter.dept_name();

    int avgSal = iter.avgSal();

    System.out.printIn(deptName + " " + avgSal);

    }

    iter.close();

ODBC

- Obertura de connectivitat de base de dades estàndard (ODBC)
  - estàndard per aplicacions a l'hora de comunicar amb el servidor de la base de dades
  - application program interface (API) to
    - obrir una connexió amb la base de dades
    - enviar consultes i actualitzacions
    - conseguir resultats
- Aplicacions tals com GUI, fulls de càlcul, etc. poden fer servir ODBC
- Originàriament va ser pensat per Basic i C, hi ha versions disponibles per a molts llenguatges

ODBC (Cont.)

- Cada sistema de base de dades que suporta ODBC proporciona un "driver" que ha de ser enllaçat amb el programa client.
- Quan el programa client fa una crida al ODBC API, el codi de la llibreria comunica amb el servidor per realitzar l'acció sol·licitada, i portar els resultats.
- El programa ODBC primer assigna un entorn SQL, després tracta una connexió amb la base de dades.
- Obrir la connexió de la base de dades fent servir SQLConnect(). Paràmetres per l'SQLConnect:
  - tractar la connexió
  - el servidor el qual connectar-se
  - l'identificador de l'uruari
  - password
- També ha d'especificar tipus d'arguments:
  - SQL_NTS denota arguments previs com a una cadena de caracters nul·la.

Codi ODBC

- int ODBCexample()
  {
  RETCODE error:
  HENV env; * environment *\
  HDBC conn; * database connection *\
  SQLAllocEnv(&env);
  SQLAllocConnect(env, &conn);
  SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
  "avipasswd", SQL_NTS);
  { ... Do actual work ... }

  SQLDisconnect(conn);
  SQLFreeConnect(conn);
  SQLFreeEnv(env);
  }

Codi ODBC (Cont.)

- El programa envia comandes SQL a la base de dades fent servir SQLExecDirect
- Els resultats de les tuples són portats fent servir SQLFetch()
- SQLBindCol() combina variables del llenguatge C a atributs del resultat de la consulta
  - Quan una tuple és portada, els seus valors atributs són automàticament emmagatzemats a les corresponents variables de C.
  - Arguments a SQLBindCol()
    - ODBC stmt variable, attribute position in query result
    - El tipus de conversió de SQL a C.
    - L'adreça de la variable.
    - Per tipus de longitud-variables com arrays de caracters,
      - La màxima longitud de la variable
      - Ubicació per emmagatzemar longitud real quan una tupla és desgavellada
      - Nota: Un valor negatiu retornat pel camp de longitud indica valor nul
- Una bona programació requereix comprovar els resultats de cada funció i mirar si hi ha errors; s'han omès moltes comprovacions per brevetat.

Codi ODBC (Cont.)

- Cos principal del programa

```
char deptname[80];
float salary;
int lenOut1, lenOut2;
HSTMT stmt;
char * sqlquery = "select dept_name, sum (salary)
                   from instructor
                   group by dept_name";
SQLAllocStmt(conn, &stmt);
error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
if (error == SQL SUCCESS) {
        SQLBindCol(stmt,1,SQL_C_CHAR, deptname, 80, &lenOut1);
        SQLBindCol(stmt,2,SQL_C_FLOAT, &salary, 0, &lenOut2);
        while(SQLFetch(stmt) == SQL_SUCCESS) {
            printf(" %s %g\n", deptname, salary);
        }
}
SQLFreeStmt(stmt, SQL_DROP);
```

ODBC declaracions preparades

- Prepared Statement
  - SQL statement prepared: compilada a la base de dades
  - Pot tenir marcadors de posició: Ex.: afegir dins d'un compte valors (?,?,?)
  - Executat repetidament amb el valor actual del marcadors de posició
- Per preparar la declaració

  SQLPrepare(stmt, <SQL String>);
- Per combinar paràmetres

  SQLBindParameter(stmt, <parameter#>,
  ... tipus d'informació i valors omesos oer simplificar...
- Per executar la declaració

  retcode = SQLExecute(stmt);
- Per evitar risc en la seguretat d'SQL, no crear cadenes de caràcters SQL directament fent servir l'input de l'usuari; en comptes d'això utilitzar prepared statements per combinar inputs d'usuari

More ODBC Features

- Metadata features
  - Trobar totes les relacions a la base de dades i
  - Trobar els noms i tipus de columnes d'un resultat d'una consulta o una relació de la base de dades
- Per defecte, cada declaració d'SQL és tractada com una transacció separada que és entregada automàticament
  - Es pot apagar l'entrega automàtica en una connexió
    - SQLSetConnectOption(conn,SQL_AUTOCOMMIT, 0)}
  - Les transaccions aleshores han de ser entregades o rolled back explícitament per
    - SQLTransact(conn, SQL_COMMIT) o
    - SQLTransact(conn, SQL_ROLLBACK)

ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
  - Core
  - Level 1 requereix el suport per consultes de metadades
  - Level 2 requereix habilitat per enviar i recuperar arrays de dades de valors i més informació detallada.
- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.

ADO.NET

- API dissenyat per Visual Basic .NET i C#, proveint capacitat a l'accés a les bases de dades similar a JDBC/ODBC
  - Exempe parcial de codi ADO.NET en C#

```
using System, System.Data, System.Data.SqlClient;
SqlConnection conn = new SqlConnection(
                "Data Source =<IPaddr>, Initial Catalog =<Catalog>");
conn.Open();
SqlCommand cmd = new SqlCommand("select * from students",
                                                conn);
SqlDataReader rdr = cmd.ExecuteReader();
while(rdr.Read()){
    Console.WriteLine(rdr[0],rdr[1]); /* Prints results attributes 1 & 2 */
}
rdr.Close(); con.Close();
```

Embedded SQL

- Els estàndards SQL defineixen assignacions d'SQL en una varietat de llenguatges de programació tals com C, Java i Cobol.
- Un llenguatge el qual les cerques SQL són assignades fa referència al host language, i les estructures SQL permeten al language host constar d'*assignacions* SQL.
- La forma bàsica d'aquests llenguatges segueixen que el Systema R assigni SQL a PL/I.
- La sentència EXEC SQL és utilizada per identificar assignacions SQL sol·licitades al preprocessador

  EXEC SQL <sentència SQL assignada> END_EXEC

  Nota: Això varia en funció del llenguatge (per exemple, en Java l'assignació fa servir #SQL{...};)

Exemple consulta

- Des de dins del llenguatge hoste, trobar l'ID i el nom dels estudiants que hagin completat més dels crèdits emmagatzemats a la variable credit_amount.

- Especificar la consulta en SQL i declarar-li un *cursor*.

```
EXEC SQL
  declare c cursor for
  select ID, name
  from student
  where tot_cred > :credit_amount
END_EXEC
```

Embedded SQL (Cont.)

- La sentència open causa la consulta a ser evaluada

  EXEC SQL **open** c END_EXEC

- La sentència fetch causa que el resultat del valor d'una tuple en una consulta sigui posada en variables de llenguatge hosta.

  EXEC SQL **fetch** c **into** $:si, : sn$ END_EXEC

  Repeated calls to **fetch** get successive tuples in the query result

- Una variable anomenada SQLSTATE a l'àrea de comunicació d'SQL (SQLCA) aconsegueix fixar a '02000' per indicar que no hi ha disponible més dades

- La sentència clause causa que el sistema de base de dades elimini la relació temporal que manté el resultat de la consulta.

  EXEC SQL **close** c END_EXEC

  Nota: Els detalls anteriors canvien en funció del llengutage. Per exemple, l'assignació en Java defineix iteradors Java per desplaçar-se pel resultat de les tuples.

Actualitzar a través de cursors

- Can update tuples fetched by cursorby declaring that the cursor is for update

```
declare c cursor for
  select *
  from instructor
  where dept_name = 'Music'
for update
```

- To update tuple at the current location of cursor c

```
update instructor
set salary = salary + 100
where current of c
```

Estructures de procediment en SQL

Extensions de procediment i Procesos d'emmagatzematge

- SQL proporciona un llenguatge **modul**
  - Permet definicions de processos en SQL, amb condicions if-then-else, for i loops while, etc.
- Processos d'emmagatzematge
  - Es poden guardar processos a la base de dades
  - després s'executen fent servir la comanda **call**
  - permet aplicacions externes per operar a la base de dades sense saber sobre detalls interns
- Aspectes d'objecte-orientat d'aquestes característiques són explicades al Capítol 22 (Object Based Databases)

Funcions i Procediments

- SQL:1999 suporta funcions i procediments
  - Funcions/procediments poden ser escrits en SQL, o en un llenguatge extern de programació
  - Les funcions són especialment útils amb tipus de dades específics com les imatges i els objectes geomètrics.
    - Exemple: funcions per comprovar si polígons se sobreposen, o per comparar imatges similars.
  - Alguns sistemes de bases de dades suporten table-valued functions, les quals poden retornar una relació com a resultat.
- SQL:1999 també suporta un gran conjunt de construccions imperatives, incloent
  - Loops, if-then-else, assignacions
- Algunes bases de dades tenen extensions pròpies de procediments a SQL que divergeixen de SQL:1999.

Funcions SQL

- Definir una funció que, donat el nom del departament, retorni el nombre d'instructors d'aquell departament.

```
create function dept_count (dept_name varchar(20))
returns integer
begin
    declare d_count integer;
    select count (*) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
end
```

- Trobar el nom del departament i el pressupost de tots els departaments amb més de 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name) > 1
```

Funcions Taula

- SQL:2003 afegeix funcions que retornen una relació com a resultat
- Exemple: Retornar tots els comptes propietat d'un client donat

```
create function instructors_of(dept_name char(20)
    returns table (ID varchar(5),
                   name varchar(20),
                   dept_name varchar(20),
                   salary numeric(8,2))
    returns table
       (select ID, name, dept_name, salary
       from instructor
       where instructor.dept_name = instructors_of.dept_name)
```

- Usage

```
select *
from table (instructors_of('Music'))
```

SQL Procedures

- La funció *dept_count* podria escriure's com a un procediment:

```
create procedure dept_count_proc (in dept_name varchar(20),
                                  out d_count integer)
begin
    select count (*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
end
```

- Els procediments poden ser invocats per un procediment SQL o bé per assignacions SQL, utilitzant la comanda **call**:

```
declare d_count integer;
call dept_count_proc('Physics', dept_count);
```

  Els procediments i les funcions poden ser invocats també desde SQL dinàmiques.
- SQL:1999 permet més d'una funció/procediment del mateix nom (anomenat name overloading), sempre que el nombre d'arguments siguin diferents, o almenys els tipus d'arguments ho siguin.

Procedural Constructs

- Warning: la majoria dels sistemes de bases de dades implementen la seva pròpia variant de la sintaxi estàndard
  - Llegeix el teu manual del sistema per veure com treballa sobre el teu sistema.
- Sentència composta: **begin ... end**,
  - Pot contenir multiples sentències SQL entre **begin** i **end**.
  - Les variables locals poden ser declarades dins d'una sentència composta.
- Les comandes **while** i **repeat**:

```
declare n integer default 0;
while n < 10 do
    set n = n + 1
end while
repeat
    set n = n - 1
    until n = 0
end repeat
```

Procedural Constructs (Cont.)

- Bucle **for**:
  - Permet fer una iteració sobre tots els resultats d'una consulta.
  - Exemple:

```
declare n integer default 0;
for r as
    select budget from department
    where dept_name = 'Music'
do
     set n = n - r.budget
end for
```

Procedural Constructs (Cont.)

- Sentències condicionals (**if-then-else**):
  SQL:1999 també proporciona una sentència de **cas** semblant a la de C.
- Exemple del procediment: el registre d'un estudiant després d'assegurar que la capacitat de les aules no sobrepassi.
  - Retorna 0 en cas d'èxit i -1 si s'excedeix la capacitat.
  - Vegeu el llibre per obtenir més informació.
- La senyalització de les condicions d'excepció, i la declaració dels manipuladors d'excepcions

  declare *out_of_classroom_seats* **condition**
  declare exit handler for *out_of_classroom_seats*
  begin
  ...
  ... signal *out_of_classroom_seats*
  end

  - El manipulador es **exit** – causes enclosing **begin...end** to be exited.
  - Altres accions són possibles sobre la excepció.

External Language Functions/Procedures

- SQL:1999 permet l'ús de funcions i procediments escrits en altres llenguatges com C o C++.
- Declaració de funcions i procediments amb llenguatge extern:

create procedure *dept_count_proc* (**in** *dep_name* **varchar**(20),
out *count* **integer**)
language *C*
external name *'/usr/avi/bin/dept_count_proc'*

create function *dept_count* (*dept_name* **varchar**(20))
returns *integer*
language *C*
external name *'/usr/avi/bin/dept_count'*

External Language Functions/Procedures (Cont.)

- Beneficis del llenguatge extern en funcions/procediments.
  - Més eficient per a moltes operacions i més poder expressiu.
- Inconvenients:
  - El codi per implementar la funció pot necessitar ser carregat dins del sistema de la base de dades i ser executat en l'espai d'adreces del sistema de base de dades.
    - risc de corrupció accidental d'estructures de bases de dades.
    - risc per a la seguretat, permetent als usuaris l'accés a dades no autoritzades.
  - Hi ha alternatives, que donen una bona seguretat en el cost del pitjor rendiment potencial.
  - L'execució directa en l'espai del sistema de base de dades s'utilitza quan l'eficiència és més important que la seguretat.

Security with External Language Routines

- Per tractar amb problemes de seguretat.
  - Utilitzar tècniques sandbox.
    - és a dir, utilitzar un llenguatge segur com Java, que no es pot utilitzar per a l'accés/dany d'altres parts del codi base de dades.
  - O bé, utilitzar les funcions/procediments de llenguatge extern en un procés separat, que no tenen accés a la memòria de procés de base de dades.
    - Paràmetres i resultats comunicats a través de la comunicació entre processos.
- Tots dos tenen les despeses generals de funcionament.
- Molts dels sistemes de bases de dades són compatibles amb els dos enfocaments anteriors, així com l'execució directa en l'espai d'adreces del sistema de base de dades.

Triggers

Triggers

- Un trigger és una declaració que s'executa automàticament pel sistema com un efecte secundari d'una modificació de la base de dades.
- Per dissenyar un mecanisme trigger, hem de:
  - Especificar les condicions en què el trigger es va a executar.
  - Especificar les accions que s'han de prendre quan s'executa el trigger.
- Els triggers es van introduir a l'estàndard SQL en SQL:1999, però es va proporcionar fins i tot abans d'utilitzar la sintaxi no estàndard per la majoria de les bases de dades.
  - La sintaxi il·lustrada aquí pot no funcionar correctament en el teu sistema de base de dades; revisa els manuals del sistema

Trigger Example

- Ex: *time_slot_id* no és una clau principal de *timeslot*, així que no podem crear una restricció de clau externa desde *section* fins a *timeslot*.
- Alternativa: utilitza triggers sobre *section* i *timeslot* per fer complir les restriccions d'integritat

```
create trigger timeslot_check1 after insert on section;
referencing new row as nrow
for each row
when(nrow.time_slot_id not in (
    select time_slot_id
    from time_slot)) /* time_slot_id not present in time_slot *
begin
    rollback
end;
```

Trigger Example (Cont.)

```
create trigger timeslot_check2 after delete on timeslot;
    referencing old row as orow
    for each row
    when(orow.time_slot_id not in (
        select time_slot_id
        from time_slot)) /* last tuple for time_slot_id deleted from time_slot *
    and(orow.time_slot_id in (
        select time_slot_id
        from section)) /* and time_slot_id still references from time_slot *
    begin
        rollback
    end;
```

Triggering Events and Actions in SQL

- Triggering event pot ser **insert**, **delete** o **update**.
- Triggers del tipus update pot ser restringit a atributs específics
  - **Ex: after update of** *takes* **on** *grade*
- Els valors dels atributs *before* i *after* d'un *update* poden ser referenciats:
  - **referencing old row as**: per deletes i updates
  - **referencing new row as**: per inserts i updates
- Els triggers poden ser activats abans d'un esdeveniment, que pot servir com a limitació addicional. E.g. convert blank grades to null.

```
create trigger setnull_trigger before update of takes;
referencing new row as nrow
for each row
when(nrow.grade = ' ')
begin atomic
    set nrow.grade = null;
end;
```

Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> *'F'* **and** *nrow.grade* **is not null**
  and (*orow.grade* = *'F'* **or** *orow.grade* **is null**)
  begin atomic
  update *student*
  set *tot_cred* = *tot_cred* +
  (**select** *credits*
  from *course*
  where *course.course_id* = *nrow.course_id*)
  where *student.id* = *nrow.id*;
  end;

Statement Level Triggers

- En lloc d'executar una acció separada per a cada fila afectada, una sola acció pot ser executada per a totes les files afectades per una transacció.
  - Utilitza **for each statement** en lloc de **for each row**
  - Utilitza **referencing old table** o **referencing new table** per referir-se a taules temporals (anomenades transition tables) que contenen les files afectades.
  - Pot ser més eficient quan es tracta de sentències SQL que actualitzen un gran nombre de files.

When Not To Use Triggers

- Triggers es van utilitzar anteriorment per a tasques com:
  - Manteniment de summary data (ex: el salari total de cada departament)
  - Replicació de bases de dades mitjançant el registre dels canvis en les relacions especials (anomenats change o delta relation) i que té un procés separat que aplica els canvis al llarg d'una rèplica.
- Hi ha millors maneres de fer això ara:
  - Les bases de dades actuals proporcionen en vista materialitzada per mantenir summary data.
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

When Not To Use Triggers

- Risk of unintended execution of triggers, for example, when
  - loading data from a backup copy
  - replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

Recursive Queries

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id )
select *
from rec_prereq;
```

This example view, *rec_prereq*, is called the *transitive closure* of the *prereq* relation

Note: 1^{st} printing of 6^{th} ed erroneously used *c_prereq* in place of *rec_prereq* in some places

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of prereq with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required

See procedure *findAllPrereqs* in book

The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *rec_prereq*
  - The next slide shows a prereq relation
  - Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
  - The final result is called the fixed point of the recursive view definition.
- Recursive views are required to be monotonic. That is, if we add tuples to prereq the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Iteration Number	Tuples in cl
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

Figure: Example of Fixed-Point Computation.

Advanced Aggregation Features

Ranking

- Ranking is done in conjunction with an order by specification.
- Suppose we are given a relation

  student_grades(ID, GPA)

  giving the grade-point average of each student
- Find the rank of each student.

  select *ID*, **rank**() **over** (**order by** *GPA* **desc**) **as** *s_rank*
  from *student_grades*
- An extra **order by** clause is needed to get them in sorted order

  select *ID*, **rank**() **over** (**order by** *GPA* **desc**) **as** *s_rank*
  from *student_grades*
  order by *s_rank*
- Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
  - **dense_rank** does not leave gaps, so next dense rank would be 2

Ranking

- Ranking can be done using basic SQL aggregation, but resultant query is very inefficient

```
select ID,( 1 + (select count(*)
    from student_grades B
    where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank
```

Ranking (Cont.)

- Ranking can be done within partition of the data.
- "Find the rank of students within each department."

```
select ID, dept_name
    rank() over ( partition by dept_name order by GPA desc)
        as dept_rank
from dept_grades
order by dept_name, dept_rank;
```

- Multiple **rank** clauses can occur in a single **select** clause.
- Ranking is done after applying **group by** clause/aggregation
- Can be used to find top-n results
  - More general than the **limit** n clause supported by many databases, since it allows top-n within each partition

Ranking (Cont.)

- Other ranking functions:
  - **percent_rank** (within partition, if partitioning is done)
  - **cume_dist** (cumulative distribution)
    - fraction of tuples with preceding values
  - **row_number** (non-deterministic in presence of duplicates)
- SQL:1999 permits the user to specify **nulls first** or **nulls last**

select *ID*,
    rank() **over** (**order by** *GPA* **desc nulls last**) **as** *s_rank*
from *student_grades*

Ranking (Cont.)

- For a given constant n, the ranking the function *ntile(n)* takes the tuples in each partition in the specified order, and divides them into n buckets with equal numbers of tuples.
- E.g.,

  select *ID*, **ntile**(4) **over** (**order by** *GPA* **desc**) **as** *quartile*
  from *student_grades*;

Windowing

- Used to smooth out random variations.
- E.g., moving average: "Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day"
- Window specification in SQL:
  - Given relation *sales(date, value)*

    select *date*, **sum**(*value*) **over**
    (**order by** *date* **between rows** 1 **preceding and** 1 **following**)
    from *sales*

Windowing (Cont.)

- Examples of other window specifications:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**
  - **range between** 10 **preceding and current row**
    - All rows with values between current row value –10 to current value
  - **range interval** 10 **day preceding**
    - Not including current row

Windowing (Cont.)

- Can do windowing within partitions
- E.g., Given a relation *transaction (account_number, date_time, value)*, where value is positive for a deposit and negative for a withdrawal
  - "Find total balance of each account after each transaction on the account"

```
select account_number, date_time
    sum(value) over
        (partition by account_number
         order by date_time
         rows unbounded preceding)
    as balance
from transaction
order by account_number, date_time
```

OLAP

Data Analysis and OLAP

- Online Analytical Processing (OLAP)
  - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- Data that can be modeled as dimension attributes and measure attributes are called multidimensional data.
  - Measure attributes
    - measure some value
    - can be aggregated upon
    - e.g., the attribute *number* of the *sales* relation
  - Dimension attributes
    - define the dimensions on which measure attributes (or aggregates thereof) are viewed
    - e.g., attributes *item_name*, *color*, and *size* of the sales relation

Example sales relation

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6

Figure: Example sales relation.

Cross Tabulation of *sales* by *item_name* and *color*

clothes_size **all**

item_name	*color* dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

Figure: Example of a cross-tabulation.

- The table above is an example of a cross-tabulation (cross-tab), also referred to as a pivot-table
  - Values for one of the dimension attributes form the row headers
  - Values for another dimension attribute form the column headers
  - Other dimension attributes are listed on top
  - Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

Data Cube

- A data cube is a multidimensional generalization of a cross-tab
- Can have n dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube

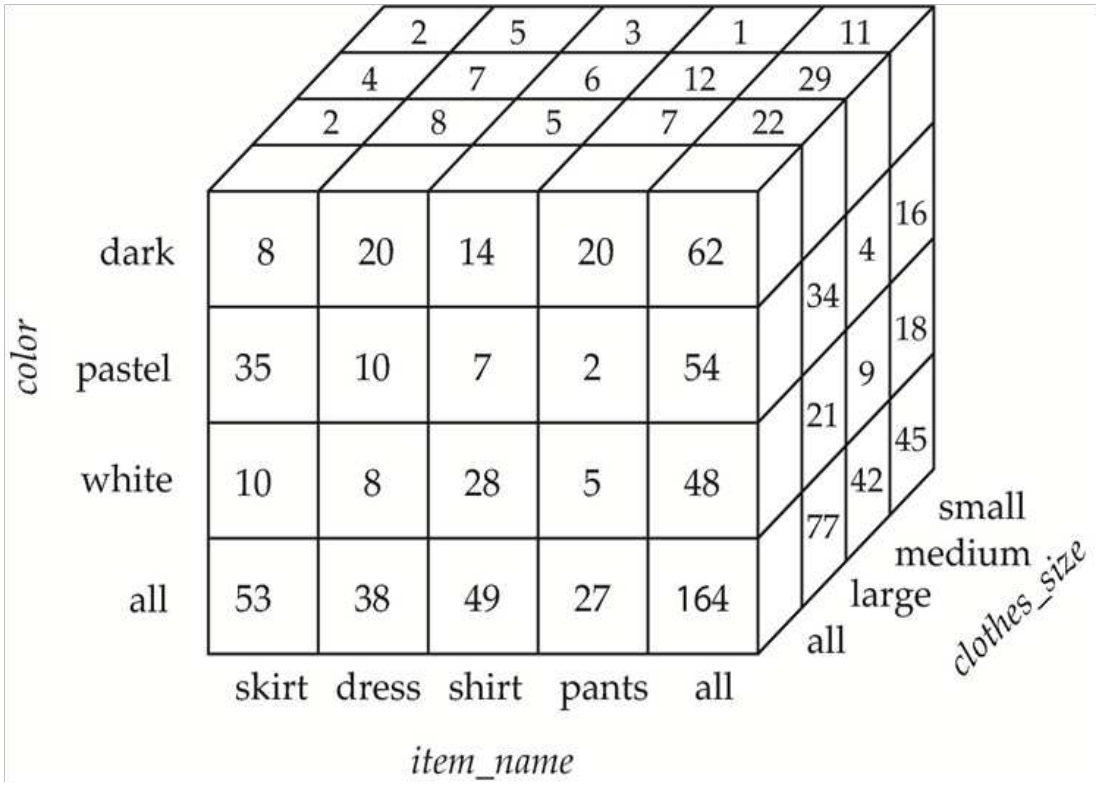

Figure: Data cube.

Hierarchies on Dimensions

- Hierarchy on dimension attributes: lets dimensions to be viewed at different levels of detail
  - E.g., the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year

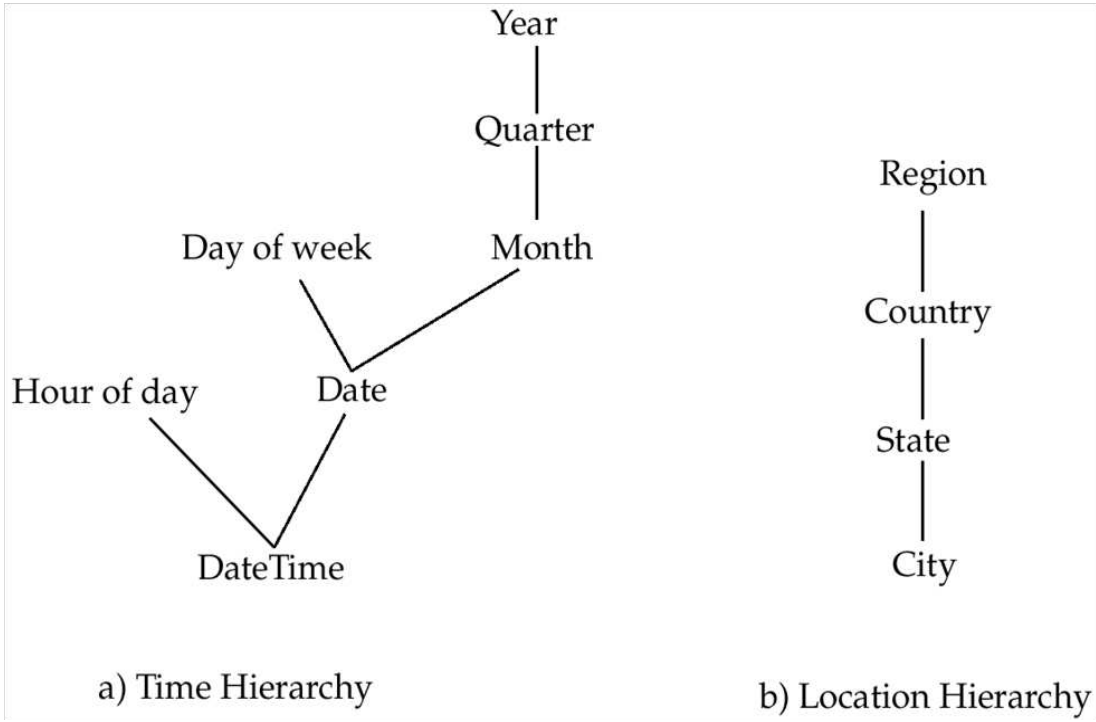

Figure: Examples.

Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
  - Can drill down or roll up on a hierarchy

clothes_size: **all**

category	*item_name*	*color*				
		dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pants	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Figure: Example of cross tabulation with Hierarchy.

Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations
  - We use the value **all** is used to represent aggregates.
  - The SQL standard actually uses null values in place of **all** despite confusion with regular null values.

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	**all**	8
skirt	pastel	**all**	35
skirt	white	**all**	10
skirt	**all**	**all**	53
dress	dark	**all**	20
dress	pastel	**all**	10
dress	white	**all**	5
dress	**all**	**all**	35
shirt	dark	**all**	14
shirt	pastel	**all**	7
shirt	White	**all**	28
shirt	**all**	**all**	49
pant	dark	**all**	20
pant	pastel	**all**	2
pant	white	**all**	5
pant	**all**	**all**	27
all	dark	**all**	62
all	pastel	**all**	54
all	white	**all**	48
all	**all**	**all**	164

Figure: Example of relational representation of cross-tabs.

Extended Aggregation to Support OLAP

- The **cube** operation computes union of **group by**'s on every subset of the specified attributes
- Example relation for this section

  sales(item_name, color, clothes_size, quantity
- E.g. consider the query

  select *item_name, color, size,* **sum***(number)*

  from *sales*

  group by cube*(item_name, color, size)*

  This computes the union of eight different groupings of the *sales* relation:

  { *(item_name, color, size), (item_name, color),*

  (item_name, size), *(color, size),*

  (item_name), *(color),*

  (size), *( ) }*

  where () denotes an empty **group by** list.
- For each grouping, the result contains the null value for attributes not present in the grouping.

Online Analytical Processing Operations

- Relational representation of cross-tab that we saw earlier, but with *null* in place of **all**, can be computed by

  select *item_name*, *color* **sum**(*number*)
  from *sales*
  group by cube(*item_name*, *color*)

- The function **grouping**() can be applied on an attribute
  - Returns 1 if the value is a null value representing all, and returns 0 in all other cases.

    select *item_name*, *color*, *size*, **sum**(*number*)
      grouping(*item_name*) **as** *item_name_flag*
      grouping(*color*) **as** *color_flag*
      grouping(*size*) **as** *size_flag*
    from *sales*
    group by cube(*item_name*, *color*, *size*)

Online Analytical Processing Operations

- Can use the function **decode**() in the **select** clause to replace such nulls by a value such as **all**
  - E.g., replace *item_name* in first query by
    decode(**grouping**(*item_name*), *1*, *"all"*, *item_name*)

Extended Aggregation (Cont.)

- The **rollup** construct generates union on every prefix of specified list of attributes
- E.g.,

  select *item_name*, *color*, *size* **sum**(*number*)
  from *sales*
  group by rollup(*item_name*, *color*, *size*)

  Generates union of four groupings:

  { (*item_name*, *color*, *size*), (*item_name*, *color*), (*item_name*), ( ) }
- Rollup can be used to generate aggregates at multiple levels of a hierarchy.
- E.g., suppose table *itemcategory*(*item_name*, *category*) gives the category of each item. Then

  select *category*, *item_name* **sum**(*number*)
  from *sales*, *itemcategory*
  where *sales.item_name* = *itemcategory.item_name*
  group by rollup(*category*, *item_name*)

  would give a hierarchical summary by *item_name* and by *category*.

Extended Aggregation (Cont.)

- Multiple rollups and cubes can be used in a single group by clause
  - Each generates set of group by lists, cross product of sets gives overall set of group by lists
- E.g.,

  select *item_name, color, size,* **sum***(number)*
  from *sales*
  group by rollup*(item_name)*, **rollup** *(color, size)*

  generates the groupings

  $\{item\_name, ()\}\ X\ \{(color, size), (color), ()\}$
  $= \{\ (item\_name, color, size), (item\_name, color), (item\_name),$
  $(color, size), (color), (\ )\ \}$

Online Analytical Processing Operations

- Pivoting: changing the dimensions used in a cross-tab is called
- Slicing: creating a cross-tab for fixed values only
  - Sometimes called dicing, particularly when values for multiple dimensions are fixed.
- Rollup: moving from finer-granularity data to a coarser granularity
- Drill down: The opposite operation - that of moving from coarser-granularity data to finer-granularity data

OLAP Implementation

- The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as multidimensional OLAP (MOLAP) systems.
- OLAP implementations using only relational database features are called relational OLAP (ROLAP) systems
- Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called hybrid OLAP (HOLAP) systems.

OLAP Implementation (Cont.)

- Early OLAP systems precomputed *all* possible aggregates in order to provide online response
  - Space and time requirements for doing so can be very high
    - 2^n combinations of **group by**
  - It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
    - Can compute aggregate on (item_name, color) from an aggregate on (item_name, color, size)
    - For all but a few "non-decomposable" aggregates such as *median*
    - is cheaper than computing it from scratch
- Several optimizations available for computing multiple aggregates
  - Can compute aggregate on *(item_name, color)* from an aggregate on *(item_name, color, size)*
  - Can compute aggregates on *(item_name, color, size)*, *(item_name, color)* and *(item_name)* using a single sorting of the base data

FINAL DEL CAPÍTOL 5

Figures

item_name	*clothes_size*	*dark*	*pastel*	*white*
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3
dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
pant	small	14	1	3
pant	medium	6	0	0
pant	large	0	1	2

Figure: 5.22

Figures

item_name	*quantity*
skirt	53
dress	35
shirt	49
pant	27

Figure: 5.23

Figures

item_name	*color*	*quantity*
skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pant	dark	20
pant	pastel	2
pant	white	5

Figure: 5.24

Another Recursion Example

- Given relation

  manager(employee_name, manager_name)

- Find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)

  with recursive *empl (employee_name, manager_name)* **as**(
  select *employee_name, manager_name*
  from *manager*
  union
  select *manager.employee_name, empl.manager_name*
  from *manager, empl*
  where *manager.employee_name* = *empl.manager_name)*
  select *
  from *empl*

  This example view, empl, is the transitive closure of the manager relation

Merge statement (now in Chapter 24)

- Merge construct allows batch processing of updates.
- Example: relation *funds_received(account_number, amount)* has batch of deposits to be added to the proper account in the account relation

merge into *account* **as** *A*
using (**select** *
from *funds_received* **as** *F*)
on(*A.account_number* = *F.account_number*)
when matched then
update set *balance* = *balance* + *F.amount*