Database Systems
Spring 2013

Database Programming
SL04

- Views
- Recursive Queries
- Integrity Constraints
- Functions and Procedural Constructs
- Triggers
- Accessing Databases

Literature and Acknowledgments

Reading List for SL04:

- Database Systems, Chapters 5 (5.2 and 5.3) and 12, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Fundamentals of Database Systems, Fourth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Addison Wesley, 2004.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

Views

- Purpose of views
- Creation and use of views
- Handling views in the DBMS
- Temporary views

Views/1

- A view is a table whose rows are not stored in the database. The rows are computed when needed from the view definition.
- This is useful in cases where
  - it is not desirable for all users to see the entire logical model (that is, all the actual tables stored in the database), or
  - the user wants to access computed results (rather than the actual data stored on the disk)
- Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

  select *CustName*, *borrower.LoanNr*, *BranchName*
  from *borrower*, *loan*
  where $borrower.LoanNr = loan.LoanNr$
- A **view** provides a mechanism to hide data from the view of users, or to give users direct access to the results of (complex) computations.

Views/2

- A view is defined using the **create view** statement which has the form

  create view *v* **as** <query expression>

  where <query expression> is any legal SQL expression. The view name is represented by *v*.
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- When a view is created, the query expression is stored in the database.
- Any table that is not of the conceptual model but is made visible to a user as a "virtual table" is called a **view**.

Views/3

- A view consisting of branches and their customers:

```
create view all_customer as
     (select BranchName, CustName
     from depositor, account
     where depositor.AccNr = account.AccNr)
     union
     (select BranchName, CustName
     from borrower, loan
     where borrower.LoanNr = loan.LoanNr)
```

- Find all customers of the Perryridge branch:

```
select CustName
from all_customer
where branche_name = ‘Perryridge’
```

Views/4

- The view definition is stored in the meta database.
- The meaning of a query expression that includes views is defined through view expansion.
- The view expansion of an expression repeats the following replacement step:

```
repeat
    Find any view relation $v_i$ in $e_1$
    Replace the view relation $v_i$ by the expression defining $v_i$
until no more view relations are present in $e_1$
```

- As long as the view definitions are not recursive, this loop will terminate

Views/5

- Tables and views can be used interchangeably in queries.
- Tables and views behave differently wrt modification operations.
- Updates to views are restricted. No broad consensus/standard exists and DBMSs behave differently.
- Roughly, a view shall be **updatable** if the database system can determine the reverse mapping from the view schema to the schema of the underlying base tables.
- The exact definition of updatable views has been enlarged significantly from SQL-1992 to SQL:1999 (cf. below).

Review 4.1

Discuss the behavior of following piece of SQL code.

```
create view GoodStudents (sid, gpa) as
  select sid, gpa
  from students
  where gpa > 3.0;

insert into GoodStudents values (51234, 2.8);
```

Views/6

- In SQL-92 a view is updatable if it is defined on a single table using projections and selections with no use of aggregate operations.
- An SQL-92 view is **not updatable** if the defining query expression satisfies one of the following conditions:
  1. the keyword DISTINCT is used in the view definition
  2. the select list contains components other than column specifications, or contains more than one specification of the same column
  3. the FROM clause specifies more than one table reference or refers to a non-updatable view
  4. the GROUP BY clause is used in the view definition
  5. the HAVING clause is used in the view definition

Views/7

- In SQL:1999 primary key constraints are taken into account for defining updatability of views.
- With this also views defined through a join can be updated.
- Intuitively, an column of a view is updatable if it can be traced back to a unique tuple in one of the underlying tables (i.e., each base tuple appears at most once in the view).
- There are views that can be modified by updating rows and there are views into which tuples can be inserted.
- View defined through set operations (union, except, intersect) cannot be inserted into but might be modifiable.

With Clause/1

- The **with** clause provides a way of defining temporary views whose definition is available only to the query in which the **with** clause occurs.
- The with clause is useful to structure complex SQL statements and eliminate code repetitions.
- Find all accounts with the maximum balance

```
with
max_balance (Value) as (
    select max(Balance)
    from account
)
select AccNr
from account, max_balance
where account.Balance = max_balance.Value
```

With Clause/2

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

```
with
branch_total (BranchName, Value) as (
    select BranchName, sum(Balance)
    from account
    group by BranchName
),
branch_total_avg (Value) as (
    select avg(Value)
    from branch_total
)
select BranchName
from branch_total, branch_total_avg
where branch_total.Value > branch_total_avg.Value
```

Review 4.2

Consider the DDL statements:

```
create table account(AccNr integer primary key,
  BranchName char(9), Balance integer);
create table depositor(AccNr integer, CustName char(9),
  primary key(AccNr, CustName));
create view v as
  select CustName, BranchName
  from depositor natural join account;
```

Explain the behavior of the following statements:

```
update v set BranchName = 'P' where CustName = 'J';
update v set CustName = 'M' where CustName = 'J';
```

Recursion in SQL

- Recursive views in SQL-1999
- Example of a recursive view

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: 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(EmpName, MgrName ) as (
          select EmpName, MgrName
          from   manager
          union
          select manager.EmpName, empl.MgrName
          from   manager, empl
          where manager.MgrName = empl.EmpName)
select *
from empl
```

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

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 *manager* with itself
    - This can give only a fixed number of levels of managers
    - Given a program we can construct a database with a greater number of levels of managers on which the program will not work
- Computing transitive closure
  - The next slide shows a *manager* relation
  - Each step of the iterative process constructs an extended version of *empl* from its recursive definition.
  - The final result is called the *fixpoint* of the recursive view definition.
- Recursive views are required to be *monotonic*. That is, if we add tuples to *manager* the view contains all of the tuples it contained before, plus possibly more.

Example of Fixed-Point Computation

manager

EmpName	MgrName
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Iteration nr	Tuples added to empl
1	(A,B), (B,E), (C,D), (D,J), (E,J), (J,K), (R,K)
2	(A,E), (B,J), (C,J), (D,K), (E,K)
3	(A,J), (B,K), (C,K)
4	(A,K)

Integrity Constraints

- Domain constraints
- Not null constraints
- Primary keys
- Check constraint
- Referential integrity (foreign keys)
- Assertions

Integrity Constraints/1

- Integrity constraints guard against damage to the database, by ensuring that changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000
  - A salary of a bank employee must be at least $4.00 an hour.
  - A customer must have a (non-null) phone number.
  - A customer can only get a loan if she has an account.
- An integrity constraint is a closed first order formula that must be true, i.e., that each database instance must satisfy.
  - Example: $\forall B(account(\_, \_, B) \Rightarrow B < 10M)$
- SQL offers **special-purpose syntax** and **efficient checking mechanisms** for the most important classes of integrity constraints.

Integrity Constraints/2

Integrity constraints on single relations:

- domain constraints
- **not null**
- **primary key**
- **unique**
- **check**(P), where P is a predicate over a single relation

Integrity constraints on multiple relations:

- **foreign key**
- **check**(P), where P is a predicate over multiple relations
- **assertion**

Domain Constraints

- **Domain constraints** are the most elementary form of integrity constraints. They check values inserted in the database, and they check queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
  - Example:

    create domain *Dollars* **integer**
    create domain *Pounds* **integer**

- We cannot assign or compare a value of type Dollars to a value of type Pounds.
  - However, we can convert values of type Dollar as follows:

    cast(*r.Amnt*/1.5 **as** *Pounds*)

Not Null Constraint

- Declare *BranchName* for *branch* to be **not null**

  BranchName **char**(15) **not null**

- Declare the domain *Dollars* to be **not null**

  create domain *Dollars* **integer not null**

Primary Key

- A primary key ensures that an attribute value is not null and unique across all rows of a table. Therefore it can be used to identify a unique row in a table, and is used for query optimization.
- Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
- The **primary key** clause lists attributes that comprise the primary key.
- Example:

```
create table customer (
    CustName char(20),
    CustStreet char(30),
    CustCity char(30),
    primary key (CustName))
```

The Unique Constraint

- **unique** $(A_1, A_2, \ldots, A_m)$
- The unique specification states that the attributes
  $A_1, A_2, \ldots, A_m$
  form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).

The check Clause/1

- **check** (P), where P is a predicate
  Example: Declare *BranchName* as the primary key for *branch* and ensure that the values of *assets* are non-negative.

```
create table branch (
          BranchName char(15),
          BranchCity char(30),
          Assets integer,
          primary key (BranchName),
          check (Assets >= 0))
```

- Note that with subqueries (not exists, etc) the check constraint can become very general. Implementations limit the predicates that are allowed in the check clause.

Referential Integrity/1

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Perryridge" is a branch name appearing in one of the tuples in the account relation, then there exists a tuple in the branch relation for branch "Perryridge".
- Foreign keys can be specified as part of the SQL **create table** statement.
- The **foreign key** clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

Referential Integrity/2

Examples of integrity constraints:

```
create table customer(
    CustomerName char(20),
    CustStreet char(30),
    CustCity char(30),
    primary key (CustomerName))

create table branch(
    BranchName char(15),
    BranchCity char(30),
    Assets integer,
    primary key (BranchName))
```

Referential Integrity/3

Examples of integrity constraints:

```
create table account (
      AccNr char(10),
      BranchN char(15),
      Balance integer,
      primary key (AccNr),
      foreign key (BranchN) references branch)

create table depositor (
      CustName char(20),
      AccNum char(10),
      primary key (CustName, AccNum),
      foreign key (AccNum) references account,
      foreign key (CustName) references customer)
```

Review 4.3

Assume tables p(X) and q(Y). p.X is a primary key. q.Y is a foreign key that references p.X

1. Use a check constraint to formulate the foreign key constraint.
2. Formulate a query that returns an empty result if the foreign key is satisfied and a non-empty result otherwise.

Assertions/1

- An **assertion** is a predicate expressing a condition that the database must satisfy.
- An assertion in SQL takes the form

  create assertion <assertion-name> **check** <predicate>

- When an assertion is made, the system tests it for validity, and tests it again on every update that might violate the assertion.
  - This testing may introduce a significant amount of overhead; hence assertions should be used with care.
- Asserting

  $\forall X(P(X))$

  is achieved using

  $\neg\exists\neg(P(X))$

Assertion/2

- Every loan has at least one borrower who maintains an account with a minimum balance or $1000.00

```
create assertion balance_constraint check
  (not exists (
    select *
    from loan
    where not exists (
      select *
      from borrower, depositor, account
      where loan.LoanNr = borrower.LoanNr
        and borrower.CustName = depositor.CustName
        and depositor.AccNr = account.AccNr
        and account.Balance >= 1000)))
```

Assertion/3

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
create assertion sum_constraint check
     (not exists (select *
          from branch
          where
               (select sum(Amount)
               from loan
               where loan.BranchName = branch.BranchName)
               >=
               (select sum(Balance)
               from account
               where loan.BranchName = branch.BranchName)))
```

Review 4.4

Consider the tables

```
  create table branch(BranchName char(9),
    BranchCity char(9), Assets integer);
  create table account(AccNr integer primary key,
    BranchName char(9), Balance integer);
```

Explain how to efficiently check the integrity constraint

$$\forall BN(account(\_, BN, \_) \Rightarrow branch(BN, \_, \_))$$

Review 4.5

Create a table for cities and a table for states. For each state its capital shall be recorded and for each city the state it is located in shall be recorded. Define primary and foreign keys. Discuss your solution.

User Defined Functions

- PL/pgSQL value functions
- PL/pgSQL table functions
- External language functions

User-Defined Functions (UDF)

- User defined functions or stored procedures allow to execute application logic in the process space of the DBMS.
- This is good for the performance since it can reduce the amount of data that is transferred between client and server.
- The SQL standard defines SQL/PSM (SQL/Persistent Stored Modules).
- PostgreSQL supports different kinds of user-defined functions:
  - Query language functions i.e., written in SQL
  - Procedural language functions such as PL/pgSQL
  - C-language functions, dynamically loadable objects (shared libraries)
- UDF functions can
  - Make Arithmetic calculations
  - Query tables
  - Manipulate tables
  - Return single values or tables

PL/pgSQL Functions/1

- Structure

  create function *somefunc*()
  returns $<$ *retype* $>$ **as $$**

  [**declare**
  $<$ *declarations* $>$]
  begin
  $<$ *statements* $>$
  end;

  $$ language plpgsql;

- $<$ *retype* $>$

  integer
  record
  table
  . . .

PL/pgSQL Functions/2

- *< declarations >*

```
quantity integer := 30;

tbl_row account%rowtype;
```

- *< statements >*

```
quantity := 4;

if quantity < 5 then
quantity := 5;
end if;

while quantity < 10 loop
quantity := quantity + 1;
end loop;
```

PL/pgSQL Value Functions/1

- A **value function** returns a value (or tuple).
- A value function can be used instead of a value in an SQL statement.
- Returning a single value is fairly straightforward and does not raise new performance issues.

- Example (cf. next slide): Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

PL/pgSQL Value Functions/2

- Count of the number of accounts owned by a customer:

```
create function accountCnt(CName varchar(9))
returns integer as $$
declare
    accCnt integer;
begin
    select count(*) into accCnt
    from depositor
    where depositor.CustName = CName;
    return accCnt;
end;
$$ language plpgsql;
```

- Usage: **select** *accountCnt*(‘*Bob*‘);

PL/pgSQL Table Functions/1

- **Table functions** return a table.
- Table functions can be used instead of a table.
- Table functions allow input parameters.
- The type of the return table must be defined.
- In the simplest case a table function returns the result of an SQL query as its result.

- Example (cf. next slide): Define a function that returns all accounts with a balance above an application-specified threshold.

PL/pgSQL Table Functions/2

- All accounts with a balance above a threshold:

```
create function highAccnts (limitVal integer)
returns table (AccNum char(10),
               BrName char(15)
               Bal integer) as $$
begin
    return query
      select AccNr, BranchName, Balance
      from account A
      where A.Balance > highAccnts.limitVal;
end;
$$ language plpgsql;
```

- Usage: **select * from** *highAccnts*(1000);

PL/pgSQL Table Functions/3

- Table functions may return large tables.
- The cursor mechanism (similar to iterators) was introduced to deal with result tables.
- A cursor points to the current row and a loop is used to iterate through all rows of a table.
- With a cursor the return type is a record that represents a row in a database table.
- For each row actions can be executed and optionally one or more result rows can be returned.

- Example (cf. next slide): Define a function that returns all accounts with a balance above an application-specified threshold.

PL/pgSQL Table Functions/4

- Accounts with balance above a threshold:

```
create function highAccnts (limitVal integer)
returns setof account as $$
declare
    accrow account%rowtype;
begin
    for accrow in select * from account loop
        if accrow.Balance > highAccnts.limitVal then
            return next accrow;
        end if;
end loop; end; $$ language plpgsql;
```

- Usage: **select * from** *highAccnts*(1000);

PostgreSQL C-Language Functions/1

- Define a function adding 1 to its argument

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);
    PG_RETURN_INT32(arg + 1);
}
```

PostgreSQL C-Language Functions/2

- Compile as dynamic library func.so
- Map to a DBMS function

  create function *add_one*(**integer**)
  returns integer
  as '*func*', '*add_one*'
  language c strict;

- Usage:

  select *add_one*(5);

Triggers

- Purpose of triggers
- Definition of triggers

Triggers/1

- A database trigger is a procedural piece of code that is automatically executed in response to certain events on a particular table in a database.
- Triggers are executed when a specified condition occurs during insert/delete/update.
- Triggers are actions that fire automatically if the condition is satisfied.
- Triggers follow an event-condition-action (ECA) model
  - Event: Database modification (e.g., insert, delete, update)
  - Condition: Any expression that evaluates to true/false
  - Action: Sequence of SQL statements that will be executed

Triggers/2

- Example of a trigger: When a new employees is added to a department, modify the TotSal of the Department to include the new employees salary

```
create trigger TotSal1
after insert on employee
for each row
when (new.Dno is not null)
    update department
    set TotSal = TotSal + new.Salary
    where Dno = new.Dno;
```

- The above syntax is the one of the SQL standard.

Triggers/3

Explanation of the trigger:

- We **create** a trigger TotSal1
- Trigger TotSal1 will execute **after insert** on employee table.
  - Instead of **after** we could also have **before** or **instead of**.
  - Instead of **insert** we could also have **update** or **delete**.
- The trigger fires (is executed) **for each row** that is inserted.
  - The trigger fires for each statement if **for each statement** is specified instead.
- The **when** condition determines if a trigger is executed or not.
- The trigger will update department by setting the new TotSal to be the sum of old TotSal and new.Salary where the Dno matches the new.Dno

Triggers/4

A PostgreSQL example of a trigger.

```
create or replace function checkTemp() returns trigger as $$
   declare
   begin
      if new.val < -273 then
         raise exception 'invalid value: %', new.val;
      end if;
      return new;
   end;
$$ language plpgsql;


create trigger TrigTempCheck
before insert or update
on temperature
for each row
execute procedure checkTemp();
```

Review 4.6

List advantages and disadvantages of triggers.

Accessing Databases

- Embedded SQL
- ODBC
- JDBC

Accessing Databases

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- **Embedded SQL**: many languages allow to embed SQL statements in their code. The embedded code can be
  - static (i.e., code is known at compile time)
  - dynamic (i.e., code is unknown at compile time; created at runtime)
- **ODBC** (Open Database Connectivity) is a Microsoft standard works with C, C++, C#, and Visual Basic
- **JDBC** (Java Database Connectivity) is from Sun Microsystems and works with Java

Embedded SQL/1

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement>;

  Note: this varies by language (e.g., #sql {...}; for Java)

Embedded SQL/2

```
int main () {
  EXEC SQL BEGIN DECLARE SECTION;
    char relname[20];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT TO
    tcp:postgresql://pg.ifi.uzh.ch/boehlen AS conn USER boehlen/xxx;

  EXEC SQL DECLARE cur CURSOR FOR
    SELECT relname FROM pg_class;
  EXEC SQL OPEN cur;
  while (true) {
    EXEC SQL FETCH IN cur INTO :relname;
    printf("%s\n", relname);
    if (sqlca.sqlcode != 0) break;
  }
  EXEC SQL CLOSE cur;

  EXEC SQL DISCONNECT;
  return 0;
}
```

ODBC/1

- Open DataBase Connectivity (ODBC) standard
  - standard for application program to communicate with a DBMS.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
- Each database system supporting ODBC provides a “driver” library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.

ODBC/2

```
int main() {
  SQLAllocEnv(&hEnv);
  SQLAllocConnect(hEnv, &hDbc);
  SQLConnect(hDbc, "PostgreSQL", SQL_NTS, "boehlen", SQL_NTS, "xxx", SQL_NTS);

  SQLAllocStmt(hDbc,&hstmt);
  SQLPrepare(hstmt,
    "select tablename from pg_tables where tableowner='boehlen'", SQL_NTS);

  SQLExecute(hstmt);
  SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER)name, 30, NULL);
  for (;;) {
    if (SQLFetch(hstmt)) break;
    printf("  '%s'\n", name);
  }

  SQLFreeStmt(hstmt, SQL_DROP);

  SQLDisconnect(hDbc);
  SQLFreeConnect(hDbc);
  SQLFreeEnv(hEnv);
}
```

JDBC/1

- **JDBC** is a Java API for communicating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
  - Open a connection
  - Create a “statement” object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC/2

```
import java.sql.*;

public class pgJDBC {

  public static void main(String[] argv) {

    Class.forName("org.postgresql.Driver");

    Connection conn = DriverManager.getConnection(
      "jdbc:postgresql://pg.ifi.uzh.ch/boehlen?ssl=true" +
      "&sslfactory=org.postgresql.ssl.NonValidatingFactory",
      "boehlen", "xxx");

     Statement stmt = conn.createStatement();

     ResultSet rset = stmt.executeQuery(
       "select tablename from pg_tables where tableowner='boehlen'");

     while (rset.next())
        System.out.println(rset.getString(1));

  }
}
```

Summary

- **Views** are essential to break up large tasks (SQL statements) in small, independent and manageable blocks.
- **Recursive queries** are used to compute ancestors, descendants, transitive closures, etc.
- **Integrity constraints** ensure a good data quality. Enforcing integrity constraints is not easy: people try to work around.
- **Functions** and **procedural constructs** are used heavily by many applications. For example PostGIS uses functions heavily to add advanced spatial functionality to PostgreSQL.
- Program access is through **ODBC** and **JDBC**. General-purpose graphical tools exist to interact with databases.