Chapter 4: Advanced SQL
Chapter 4: Advanced SQL

- SQL Data Types and Schemas
- Integrity Constraints
- Authorization
- Embedded SQL
- Dynamic SQL
- Functions and Procedural Constructs
- Recursive Queries**
- Advanced SQL Features**
Built-in Data Types in SQL

- **date**: Dates, containing a (4 digit) year, month and date
  - Example: `date '2005-7-27'`

- **time**: Time of day, in hours, minutes and seconds.
  - Example: `time '09:00:30'`  `time '09:00:30.75'`

- **timestamp**: date plus time of day
  - Example: `timestamp '2005-7-27 09:00:30.75'`

- **interval**: period of time
  - Example: `interval '1' day`
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values
• Can extract values of individual fields from date/time/timestamp
  • Example: `extract (year from r.starttime)`

• Can cast string types to date/time/timestamp
  • Example: `cast <string-valued-expression> as date`
  • Example: `cast <string-valued-expression> as time`
User-Defined Types

- **create type** construct in SQL creates user-defined type

  \textit{create type} \textit{Dollars as numeric (12,2) final}

- **create domain** construct in SQL-92 creates user-defined domain types

  \textit{create domain} \textit{person\_name char(20) not null}

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
Domain Constraints

- **Domain constraints** are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
  - Example: `create domain Dollars numeric(12, 2) create domain Pounds numeric(12,2)`
  - We cannot assign or compare a value of type Dollars to a value of type Pounds.
    - However, we can convert type as below
      - `(cast r.A as Dollars)`
      - (Should also multiply by the dollar-to-pound conversion-rate)
Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a large object:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself.
Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number
Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** \((P)\), where \(P\) is a predicate
Not Null Constraint

- Declare `branch_name` for `branch` is not null
  \[ branch\_name \text{ char}(15) \text{ not null} \]

- Declare the domain `Dollars` to be not null
  \[ \text{create domain Dollars numeric}(12,2) \text{ not null} \]
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).
Entity integrity

- Primary key must be not null
The check clause

- **check** \((P)\), where \(P\) is a predicate

Example: Declare \(branch\_name\) as the primary key for \(branch\) and ensure that the values of assets are non-negative.

```
create table branch
    (branch_name char(15),
     branch_city char(30),
     assets integer,
     primary key (branch_name),
     check (assets >= 0))
```
The check clause (Cont.)

- The **check** clause in SQL-92 permits domains to be restricted:
  - Use **check** clause to ensure that an hourly_wage domain allows only values greater than a specified value.
    
    ```sql
    create domain hourly_wage numeric(5,2)
    constraint value_test check(value >= 4.00)
    ```

  - The domain has a constraint that ensures that the hourly_wage is greater than 4.00
  - The clause **constraint value_test** is optional; useful to indicate which constraint an update violated.
Referential Integrity

- 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”.

- 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.
  - The unique key clause lists attributes that comprise a candidate key.
  - 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 in SQL – Example

create table customer
    (customer_name char(20),
    customer_street char(30),
    customer_city char(30),
    primary key (customer_name ))

create table branch
    (branch_name char(15),
    branch_city char(30),
    assets numeric(12,2),
    primary key (branch_name ))
create table account
  (account_number char(10),
   branch_name   char(15),
   balance       integer,
   primary key (account_number),
   foreign key (branch_name) references branch )

create table depositor
  (customer_name  char(20),
   account_number char(10),
   primary key (customer_name, account_number),
   foreign key (account_number ) references account,
   foreign key (customer_name ) references customer )
Assertions

- An assertion is a predicate expressing a condition that we wish the database always to satisfy.
- An assertion in SQL takes the form
  
  \texttt{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 may violate the assertion.
  
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

- Asserting
  
  for all $X$, $P(X)$

  is achieved in a round-about fashion using

  not exists $X$ such that not $P(X)$
Assertion Example

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

```sql
create assertion balance_constraint check
(not exists (
    select *
    from loan
    where not exists ( 
        select *
        from borrower, depositor, account
        where loan.loan_number = borrower.loan_number
        and borrower.customer_name = depositor.customer_name
        and depositor.account_number = account.account_number
        and account.balance >= 1000)))
```
Assertion Example

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

```sql
create assertion sum_constraint check
    (not exists (select *
        from branch
        where (select sum(amount)
            from loan
            where loan.branch_name = branch.branch_name)
        >= (select sum(amount)
            from account
            where account.branch_name = branch.branch_name )))
```
Triggers

- A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database.

- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.

- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
Trigger Example

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - setting the account balance to zero
  - creating a loan in the amount of the overdraft
  - giving this loan a loan number identical to the account number of the overdrawn account
- The condition for executing the trigger is an update to the account relation that results in a negative balance value.
Trigger Example in SQL:1999

create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
    insert into borrower
    (select customer-name, account-number
     from depositor
     where nrow.account-number = depositor.account-number);
    insert into loan values
    (n.row.account-number, nrow.branch-name, − nrow.balance);
    update account set balance = 0
    where account.account-number = nrow.account-number
end
Triggering Events and Actions in SQL

- Triggering event can be `insert`, `delete` or `update`
- Triggers on update can be restricted to specific attributes
  - E.g. `create trigger overdraft-trigger after update of balance on account`
- Values of attributes before and after an update can be referenced
  - referencing old row as : for deletes and updates
  - referencing new row as : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blanks to null.

```
create trigger setnull-trigger before update on r
referencing new row as nrow for each row
when nrow.phone-number = ' ' 
set nrow.phone-number = null
```
Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called *transition tables*) containing the affected rows
- Can be more efficient when dealing with SQL statements that update a large number of rows
• Alter trigger *trigger-name* disable

• *drop* trigger *trigger-name*
External World Actions

- We sometimes require external world actions to be triggered on a database update
  - E.g. re-ordering an item whose quantity in a warehouse has become small, or turning on an alarm light,
- Triggers cannot be used to directly implement external-world actions, BUT
  - Triggers can be used to record actions-to-be-taken in a separate table
  - Have an external process that repeatedly scans the table, carries out external-world actions and deletes action from table
- E.g. Suppose a warehouse has the following tables
  - inventory \((item, level)\): How much of each item is in the warehouse
  - minlevel \((item, level)\): What is the minimum desired level of each item
  - reorder \((item, amount)\): What quantity should we re-order at a time
  - orders \((item, amount)\): Orders to be placed (read by external process)
create trigger reorder-trigger after update of amount on inventory
referencing old row as orow, new row as nrow
for each row
  when nrow.level <= (select level
    from minlevel
    where minlevel.item = orow.item)
    and orow.level > (select level
    from minlevel
    where minlevel.item = orow.item)
begin
  insert into orders
    (select item, amount
    from reorder
    where reorder.item = orow.item)
end
When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g. total salary of each department)
  - Replicating databases by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica

- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.
Users can be given authorization on views, without being given any authorization on the relations used in the view definition.

Ability of views to hide data serves both to simplify usage of the system and to enhance security by allowing users access only to data they need for their job.

A combination or relational-level security and view-level security can be used to limit a user’s access to precisely the data that user needs.
Suppose a bank clerk needs to know the names of the customers of each branch, but is not authorized to see specific loan information.

- Approach: Deny direct access to the loan relation, but grant access to the view cust-loan, which consists only of the names of customers and the branches at which they have a loan.
- The cust-loan view is defined in SQL as follows:

```
create view cust-loan as
    select branchname, customer-name
    from borrower, loan
    where borrower.loan-number = loan.loan-number
```
The clerk is authorized to see the result of the query:

```
select *
from cust-loan
```

When the query processor translates the result into a query on the actual relations in the database, we obtain a query on borrower and loan.

Authorization must be checked on the clerk’s query before query processing replaces a view by the definition of the view.
Authorization on Views

- Creation of view does not require resources authorization since no real relation is being created.
- The creator of a view gets only those privileges that provide no additional authorization beyond that he already had.
- E.g. if creator of view cust-loan had only read authorization on borrower and loan, he gets only read authorization on cust-loan.
Authorization Specification in SQL

- The grant statement is used to confer authorization
  
  ```sql
  grant <privilege list>
  on <relation name or view name> to <user list>
  ```

- `<user list>` is:
  - a user-id
  - `public`, which allows all valid users the privilege granted
  - A role (more on this later)

- Granting a privilege on a view does not imply granting any privileges on the underlying relations.

- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).
Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users U₁, U₂, and U₃ *select* authorization on the *branch* relation:
    ```
    grant select on branch to U₁, U₂, U₃
    ```

- **insert**: the ability to insert tuples

- **update**: the ability to update using the SQL update statement

- **delete**: the ability to delete tuples.

- **references**: ability to declare foreign keys when creating relations.

- **usage**: In SQL-92; authorizes a user to use a specified domain

- **all privileges**: used as a short form for all the allowable privileges
Privilege To Grant Privileges

• **with grant option**: allows a user who is granted a privilege to pass the privilege on to other users.
• Example:

  ```sql
  grant select on branch to U_1 with grant option
  ```

  gives U_1 the **select** privileges on branch and allows U_1 to grant this privilege to others
Granting of Privileges

- The passage of authorization from one user to another may be represented by an authorization graph.
- The nodes of this graph are the users.
- The root of the graph is the database administrator.
- Consider graph for update authorization on loan.
- An edge $U_i \rightarrow U_j$ indicates that user $U_i$ has granted update authorization on loan to $U_j$. 

![Diagram]

- The root of the graph is the database administrator (DBA).
- The nodes $U_1, U_2, U_3, U_4, U_5$ represent different users.
- An edge from $U_i$ to $U_j$ indicates that $U_i$ has granted update authorization on loan to $U_j$. 

[Diagram: A directed graph with nodes labeled $U_1, U_2, U_3, U_4, U_5$ and edges indicating authorization relationships.]
Authorization Grant Graph

- **Requirement**: All edges in an authorization graph must be part of some path originating with the database administrator.

- If DBA revokes grant from $U_1$:
  - Grant must be revoked from $U_4$ since $U_1$ no longer has authorization.
  - Grant must not be revoked from $U_5$ since $U_5$ has another authorization path from DBA through $U_2$.

- Must prevent cycles of grants with no path from the root:
  - DBA grants authorization to $U_7$.
  - $U_7$ grants authorization to $U_8$.
  - $U_8$ grants authorization to $U_7$.
  - DBA revokes authorization from $U_7$.

- Must revoke grant $U_7$ to $U_8$ and from $U_8$ to $U_7$ since there is no path from DBA to $U_7$ or to $U_8$ anymore.
Roles

- Roles permit common privileges for a class of users can be specified just once by creating a corresponding “role”
- Privileges can be granted to or revoked from roles, just like user
- Roles can be assigned to users, and even to other roles

- create role  
  teller
- create role  
  manager

  grant select on  
  branch to  
  teller
  grant update (balance) on  
  account to  
  teller
  grant all privileges on  
  account to  
  manager

  grant  
  teller to  
  manager

  grant  
  teller to  
  alice, bob
  grant  
  manager to  
  avi
Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.
  
  ```sql
  revoke <privilege list>
  on <relation name or view name> from <user list> [restrict | cascade]
  ```

- Example:
  
  ```sql
  revoke select on branch from U1, U2, U3 cascade
  ```

- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the **revoke**.

- We can prevent cascading by specifying **restrict**:
  
  ```sql
  revoke select on branch from U1, U2, U3 restrict
  ```

  With **restrict**, the **revoke** command fails if cascading revokes are required.
Revoking Authorization in SQL (Cont.)

- `<privilege-list>` may be **all to** revoke all privileges the revokee may hold.
- If `<revokee-list>` includes `public` all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.
Limitations of SQL Authorization

- SQL does not support authorization at a tuple level
  - E.g. we cannot restrict students to see only (the tuples storing) their own grades
- With the growth in Web access to databases, database accesses come primarily from application servers.
  - End users don't have database user ids, they are all mapped to the same database user id
- All end-users of an application (such as a web application) may be mapped to a single database user
- The task of authorization in above cases falls on the application program, with no support from SQL
  - Benefit: fine grained authorizations, such as to individual tuples, can be implemented by the application.
  - Drawback: Authorization must be done in application code, and may be dispersed all over an application
  - Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code
Embedded SQL

• The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, Fortran 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> END_EXEC

Note: this varies by language (for example, the Java embedding uses

  # SQL { .... } ; )
Connect to database

- EXEC SQL **connect** to server **user** user-name
  END-EXEC

Communication between host language and SQL

- SQLCA - SQLSTATE
- Host variable :amount
  Indicator Variable :amount:ID
- Cursor
Example Query

- From within a host language, find the names and cities of customers with more than the variable amount dollars in some account.

- Specify the query in SQL and declare a cursor for it

```
EXEC SQL
declare c cursor for
select depositor.customer_name, customer_city
from depositor, customer, account
where depositor.customer_name = customer.customer_name
    and depositor.account_number = account.account_number
    and account.balance > :amount
END_EXEC
```
Cursor in embedded SQL

- The **open** statement causes the query to be evaluated
  ```sql
  EXEC SQL open c END_EXEC
  ```
- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.
  ```sql
  EXEC SQL fetch c into :cn, :cc END_EXEC
  ```
  Repeated calls to **fetch** get successive tuples in the query result
- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to ‘02000’ to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.
  ```sql
  EXEC SQL close c END_EXEC
  ```

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.
Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update
  
  ```sql
  declare c cursor for
  select *
  from account
  where branch_name = 'Perryridge'
  for update
  ```

- To update tuple at the current location of cursor `c`

  ```sql
  update account
  set balance = balance + 100
  where current of c
  ```
Example: small C program with Embedded SQL

EXEC SQL INCLUDE SQLCA;
    /* (1) 定义SQL通信区 */
EXEC SQL BEGIN DECLARE SECTION;
    /* (2) 说明主变量 */
CHAR   title_id(7);
CHAR   title(81);
INT    royalty;
EXEC SQL END DECLARE SECTION;
main()
{
    EXEC SQL DECLARE C1 CURSOR FOR
    SELECT tit_id, tit, roy FROM titles;
    /* (3) 游标操作（定义游标）*/
    /* 从titles表中查询tit_id, tit, roy */
    EXEC SQL OPEN C1;
    /* (4) 游标操作（打开游标）*/
for(;;)
{
    EXEC SQL FETCH C1 INTO :title_id, :title, :royalty;
    /* (5) 游标操作（将当前数据放入主变量并推进游标指针） */
    if (SQLCA.SQLSTATE <> SUCCESS)
        /* (6) 利用SQLCA中的状态信息决定何时退出循环 */
        break;
    printf("Title ID: %s, Royalty: %d", :title_id, :royalty);
    printf("Title: %s", :title);
    /* 打印查询结果 */
}
EXEC SQL CLOSE C1;
    /* (7) 游标操作（关闭游标） */
Dynamic SQL

- Allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```c
char * sqlprog = "update account
    set balance = balance * 1.05
    where account_number = ?"
EXEC SQL prepare dynprog from :sqlprog;
EXEC SQL execute dynprog using :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.
ODBC and JDBC

- 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
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java
Open DataBase Connectivity (ODBC) standard
- standard for application program to communicate with a database server.
- 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
ODBC (Cont.)

- 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 program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password
JDBC

- 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
Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.

- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language
  - Functions are particularly useful with specialized data types such as images and geometric objects
    - Example: functions to check if polygons overlap, or to compare images for similarity
  - Some database systems support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
SQL Functions

• Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

```sql
create function account_count (customer_name varchar(20))
returns integer
begin
declare a_count integer;
select count(*) into a_count
from depositor
where depositor.customer_name = customer_name
return a_count;
end
```

• Find the name and address of each customer that has more than one account.

```sql
select customer_name, customer_street, customer_city
from customer
where account_count (customer_name) > 1
```
Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```sql
create function accounts_of (customer_name char(20))
returns table (account_number char(10),
branch_name char(15),
balance numeric(12,2))

return table
(select account_number, branch_name, balance
from account A
where exists (
    select *
    from depositor D
    where D.customer_name = accounts_of.customer_name
    and D.account_number = A.account_number )
```
Table Functions (cont’d)

- Usage

```sql
select *
from table (accounts_of (‘Smith’))
```
The *author_count* function could instead be written as procedure:

```
create procedure account_count_proc (in title varchar(20),
    out a_count integer)
begin
    select count(author) into a_count
    from depositor
    where depositor.customer_name = account_count_proc.customer_name
end
```

Procedures can be invoked either from an SQL procedure or from embedded SQL, using the `call` statement.

```
declare a_count integer;
call account_count_proc(‘Smith’, a_count);
```

Procedures and functions can be invoked also from dynamic SQL.

SQL:1999 allows more than one function/procedure of the same name (called name overloading), as long as the number of arguments differ, or at least the types of the arguments differ.
Procedural Constructs

- Compound statement: `begin ... end`,
  - May contain multiple SQL statements between begin and end.
  - Local variables can be declared within a compound statements

- While and repeat statements:
  ```
  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.)

- **For** loop
  - Permits iteration over all results of a query
  - Example: find total of all balances at the Perryridge branch

```sql
declare n integer default 0;
for r as
    select balance from account
    where branch_name = 'Perryridge'
do
    set n = n + r.balance
end for
```
Procedural Constructs (cont.)

- Conditional statements  (**if-then-else**)
  E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)

  ```
  if r.balance < 1000
    then set l = l + r.balance
  elseif r.balance < 5000
    then set m = m + r.balance
  else set h = h + r.balance
  end if
  ```

- Signaling of exception conditions, and declaring handlers for exceptions

  ```
  declare out_of_stock condition
  declare exit handler for out_of_stock
  begin
    ...
    .. signal out-of-stock
  end
  ```

  - The handler here is **exit** -- causes enclosing **begin..end** to be exited
  - Other actions possible on exception
External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```sql
create procedure account_count_proc(
    in customer_name varchar(20),
    out count integer)
language C
external name '/usr/avi/bin/account_count_proc'

create function account_count(
    customer_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/author_count'
```
External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power

- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system’s address space
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data
  - There are alternatives, which give good security at the cost of potentially worse performance
  - Direct execution in the database system’s space is used when efficiency is more important than security
Security with External Language Routines

• To deal with security problems
  • Use sandbox techniques
    • that is use a safe language like Java, which cannot be used to access/damage other parts of the database code
  • Or, run external language functions/procedures in a separate process, with no access to the database process’ memory
    • Parameters and results communicated via inter-process communication

• Both have performance overheads

• Many database systems support both above approaches as well as direct executing in database system address space
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’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.manager_name = empl.employe_name) 
  select * 
  from empl 
```

This example view, `empl`, is called the transitive closure of the manager relation.
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 fixed point 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

<table>
<thead>
<tr>
<th>employee_name</th>
<th>manager_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alon</td>
<td>Barinsky</td>
</tr>
<tr>
<td>Barinsky</td>
<td>Estovar</td>
</tr>
<tr>
<td>Corbin</td>
<td>Duarte</td>
</tr>
<tr>
<td>Duarte</td>
<td>Jones</td>
</tr>
<tr>
<td>Estovar</td>
<td>Jones</td>
</tr>
<tr>
<td>Jones</td>
<td>Klinger</td>
</tr>
<tr>
<td>Rensal</td>
<td>Klinger</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Tuples in empl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(Duarte), (Estovar)</td>
</tr>
<tr>
<td>2</td>
<td>(Duarte), (Estovar), (Barinsky), (Corbin)</td>
</tr>
<tr>
<td>3</td>
<td>(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)</td>
</tr>
<tr>
<td>4</td>
<td>(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)</td>
</tr>
</tbody>
</table>
Advanced SQL Features**

- Create a table with the same schema as an existing table:
  ```sql
  create table temp_account like account
  ```
- SQL:2003 allows subqueries to occur anywhere a value is required provided the subquery returns only one value. This applies to updates as well.
- SQL:2003 allows subqueries in the `from` clause to access attributes of other relations in the `from` clause using the `lateral` construct:
  ```sql
  select C.customer_name, num_accounts
  from customer C,
      lateral (select count(*)
                from account A
                where A.customer_name = C.customer_name )
  as this_customer (num_accounts )
  ```
Advanced SQL Features (cont’d)

- 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

```sql
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
```
End of Chapter 4