Chapter 4: Advanced SQL

- Know about some Special Data Types
- Know how to use:
  - Integrity Constraints
  - Authorization
- Know about Functions and Procedural Constructs

Date and Time Data Types in SQL

- date: Dates, contain a (4 digit) year, month and date
  - Example: `date '2005-7-27'`
- time: Time of day, in hours, minutes and seconds
  - Examples: `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

Manipulating Dates and Times

- Extracting values of individual fields from date/time/timestamp
  - Example: `extract (year from r.starttime)`
  - Related method: `year(r.starttime)`
- Translating string to date/time/timestamp
  - Example: `cast <string-valued-expression> as date`
  - Example: `cast <string-valued-expression> as time`
- Many more functions are available:
  - MySQL 5.0 Ref Man, Sections 10.1.2, 10.3: date & time types
  - MySQL 5.0 Ref Man, Section 11.6: date & time functions

Large-Object Types – BLOB and CLOB

- Large objects (photos, videos, CAD files, etc.) are stored as a large object:
  - BLOB: binary large object
    - A large collection of uninterpreted binary data
    - Interpretation is left to the application outside of the database
  - CLOB: character large object
    - A large collection of character data
  - Size limits are vendor-specific
  - When a query returns a large object, a pointer is returned rather than the large object itself.

User-Defined Data Types

- Supported in most DBMS products, but not MySQL
- Creating a Domain (SQL-92):
  - `create domain person_name char(20) not null`
- Creating a User-defined type (SQL-99):
  - `create type Dollars as numeric (12,2) final`
- User-defined Domains vs. Types
  - Domains may have constraints and default values
  - Types are strongly typed (compiler catches type mismatches)
  - Types can also be used in procedures, not just attributes

Integrity Constraints

- We need to guarantee that illegal or inconsistent data values or relationships do not occur.
  - A checking account balance must be greater than $10,000.00
  - A customer must provide a phone number
  - A checking account must always refer to a valid customer

- Types of Integrity
  - Data Integrity (within one table)
  - Referential Integrity (table-to-table)
Data Integrity Constraints

- **Column-level**
  - not null
  - unique
  - primary key
  - Check the [MySQL 5.0 CREATE TABLE syntax](#)

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

SQL Syntax for Column Constraints

- Most column constraints can be defined in the same clause with the column:
  - `col_nameA col_typeA [constraints],
  - `col_nameB col_typeB [constraints],

- Constraints that apply to a group of columns (multi-attribute primary keys and candidate keys) appear separate clauses:
  - `constraint (list of columns),`

Choosing Column Constraints

- Use the constraints that imply a column’s key status:
  - A primary key must be both **Unique** and **Not Null**.
    (In MySQL, **Primary Key** implies **Unique** and **Not Null**)
  - A candidate key must be **Unique**.
  - Columns of integer type can be set to **Auto_Increment**.
    - When inserting a new row, if the user omits a value, the DBMS will automatically assign a value (last new value + 1)
    - Useful for “artificial” primary keys.
  - A key obviously should NOT have a **Default** value.
  - Not Null can be used anytime you want to require a value.

- Warning: **Key** is not the same as **Primary Key**. A plain **Key** is an index, related to physical data arrangement and access (Ch. 12)

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.

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

CHECK on a Domain

- In SQL-92, domain definitions may include a **Check** clause to restrict domain values:
  - Use **check** to ensure that an `hourly_wage` domain allows only values greater than 4.00
  - `create domain hourly_wage numeric(5,2) constraint value_test check(value > 4.00)`
  - The phrase **constraint value_test** is optional; this allows constraint violation messages to refer to constraints by name

Referential Integrity

- Ensures that every **value** appearing in a given set of attributes in one relation also appears in a given set of attributes in another relation.
  - Example: If “Perryridge” is a branch name appearing in a tuple in the `account` table, 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 Example in SQL

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)

Another Foreign Key Example

<table>
<thead>
<tr>
<th>branch_id</th>
<th>city</th>
<th>mgr_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>Louisville</td>
<td>Cassius Clay</td>
</tr>
<tr>
<td>45</td>
<td>Winterset</td>
<td>Marion Morrison</td>
</tr>
<tr>
<td>16</td>
<td>Lexington</td>
<td>Mary Todd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>employee_name</th>
<th>salary</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassius Clay</td>
<td>100000</td>
<td>444-2908</td>
</tr>
<tr>
<td>Marion Morrison</td>
<td>100000</td>
<td>892-2770</td>
</tr>
<tr>
<td>Mary Todd</td>
<td>1000</td>
<td>Null</td>
</tr>
</tbody>
</table>

FOREIGN KEYMgr_name REFERENCES Employee (employee_name)

Insert, Update, and Delete Integrity

- Assume manager_name is a foreign key in the Branch table, referencing the Employee table.
  - The basic Foreign Key constraint applies to Insert: you can’t add a branch that references a non-existent manager.
  - Update: What if a manager wishes to change his/her name?
    - Can the name be safely changed in the Branch table?
    - Can the name be safely changed in the Employee table?
    - What would be the effect on the other table?
  - Delete: What if a manager leaves the company?
    - Can the manager be safely removed from the Employee table?
    - What would be the effect on the other table?

Cascading Update and Delete

- Update: What if a manager wishes to change his/her name?
  - Can the name be safely changed in the Branch table? NO
  - Can the name be safely changed in the Employee table? ONLY IF…
  - What would be the effect on the other table?
    - We allow updates to cascade: changes in the primary table automatically ripple through to the referencing table.
- Delete: What if a manager leaves the company?
  - Can the manager be safely removed from the Employee table? ONLY IF we wish to replace the name with NULL
  - What would be the effect on the other table?
    - we either replace referencing values with NULL, OR we allow deletions to cascade: remove referencing rows

SQL for Update & Delete Constraints

create table Branch
  (branch_id int,
   city char(30),
   mgr_name char(50),
   primary key (branch_id),
   foreign key (mgr_name) references Employee
     on delete set null
     on update cascade )

create table Employee
  (employee_name char(50),
   salary numeric(10,2),
   phone char(13),
   primary key (employee_name) )

Assertions – more CHECKs

- An assertion is a predicate expressing a condition that we wish the database always to satisfy.
  - Stand-alone statement; not part of a table
- 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 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 or $1000.00.

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.

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

Authorization – User Permissions

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 schema (Ch. 8):
- **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.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization
  
  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 in Chapter 8)

  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 administrator).

  The **revoke** statement rescinds authorization

Grant 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:

  ```sql
  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.
- **All privileges** - used as a short form for all the allowable privileges

Procedural SQL

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

- SQL now has **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

  These features are covered in Chapter 9
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
  - Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

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
    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’))
  ```

SQL Procedures

- The author_count function could instead be written as procedure:
  ```sql
  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.
  ```sql
  declare a_count integer;
  call account_count_proc (‘Smith’, a_count);
  ```

Procedural Constructs – While loops

- Compound statement: begin ... end,
  - Local variables can be declared within a compound statements
- While and repeat statements:
  ```sql
  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 – For loops

- **For loop**
  - Permits iteration over all results of a query
  - Example: find total of all balances at the Perryridge branch
    
    ```
    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 – If, Case, Exceptions

- **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
    ```
  - SQL:1999 also supports a case statement similar to C case statement
  - 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
    ```

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(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.employee_name) 
  select * 
  from empl  
  ```

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

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

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