Chapter 4: Advanced SQL

Database System Concepts, 5th Ed.

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

User-Defined Types

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

  create type *Dollars* **as numeric (12,2) final**

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

  create domain *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** *Pounds*)
    (Should also multiply by the dollar-to-pound conversion-rate)

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

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

  create domain *Dollars* **numeric**(12,2) **not null**

The Unique Constraint

- **unique** (A_1, A_2, …, A_m)
- The unique specification states that the attributes

  *A*1, *A*2, … *A*m

  form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).

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.

    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 ))
```

Referential Integrity in SQL – Example (Cont.)

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

  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 *
                 from branch
                 where (select sum(amount )
                        from loan
                        where loan.branch_name =
                              branch.branch_name )
                     >= (select sum (amount )
                         from account
                         where loan.branch_name =
                               branch.branch_name )))
```

Limitations of Check & Assertions

- ???
- For that you need triggers …

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 schema (covered in Chapter 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 database administrator).

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>

  on <relation name or view name> **from** <user list>
- Example:

  revoke select on *branch* **from** U_1, U_2, U_3
- <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.

Embedded SQL

- 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.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- **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 { …. };)

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

Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated

  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.

  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.

  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

  declare *c* **cursor for**
  select *
  from *account*
  where *branch_name* = 'Perryridge'
  for update

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

  update *account*
  set *balance* = *balance* + 100
  where current of *c*

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.

```
char * sqlprog = "update account
                  set balance = balance * 1.05
                  where account_number = ?"
EXEC SQL prepare dynprog from :sqlprog;
char account [10] = "A-101";
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

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

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.

```
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.

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


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'

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

End of Chapter

Database System Concepts, 5th Ed.