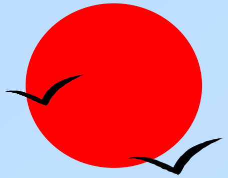
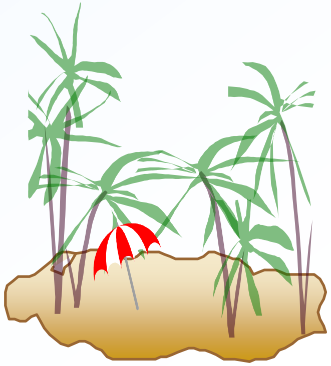


Chapter 4: Advanced SQL



Contents

- SQL Data Types and Schemas
- Integrity Constraints
- Authorization
- Embedded SQL
- Dynamic SQL
- ODBC and JDBC

Built-in Data Types in SQL

- **date**: Made up of year-month-day in the format yyyy-mm-dd

  '2005-07-27'

- **time**: Made up of hour:minute:second in the format hh:mm:ss

  '09:00:30'

Cont.

Built-in Data Types in SQL

- **time(i)**:
  - Made up of hour:minute:second plus i additional digits specifying fractions of a second
  - format is hh:mm:ss:ii...i

‘09:00:30.75’

Cont.

Built-in Data Types in SQL

- **Timestamp**: date plus time of day

  ‘2005-7-27 09:00:30’

- **interval**: period of time
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Cont.

Build-in Data Types in SQL

- Can extract values of individual fields from date/time/timestamp

```
extract (year from r.starttime)
```

- Can cast string types to date/time/timestamp

```
cast <string-valued-expression> as date
```

```
cast <string-valued-expression> as time
```

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.

Example

Find all customers who have the same name as branch

Not a meaningful query

⌗ To forbid this kind of query, customer_name and brach_name should have distinct domains

Domain Constraints Cont.

- New domains can be created from existing data types

```
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

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.

Example

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  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

  A1, A2, … Am

  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.

The check clause Cont.

```
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

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

Referential Integrity Cont.

- The **primary key** clause lists attributes that comprise the primary key.
- The **unique** 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.

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 ))
```

Example

```
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>
```

Assertions Cont.

- 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*)

  SQL do not provide this construct directory

  is achieved in a round-about fashion using

  not exists *X* such that not *P*(*X*)

Example

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

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 account.branch_name = branch.branch_name )))
```

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.

Authorization Cont.

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

A list of Privileges

All privilege, all allowable privileges

grant <privilege list>

on <relation name or view name> **to** <user list>

A list of user-id

public, all current and future users of the system

Authorization Specification in SQL

- 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*

Privileges in SQL

- **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

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** *U1, U2, U3*

Cont.

Revoking Authorization in SQL

- 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.

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 { …. };)

Example Query

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

```
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

Embedded SQL

Cont.

- 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
```

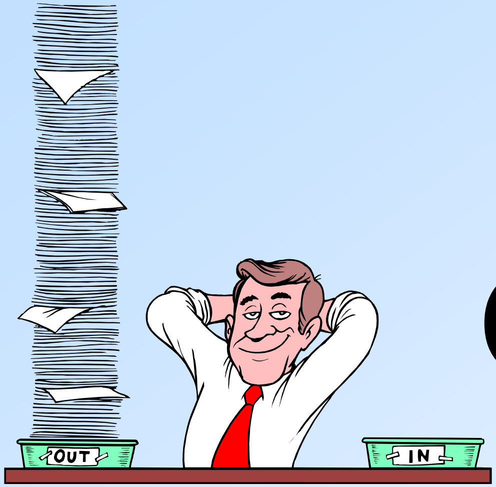
Dynamic SQL

- 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;
```

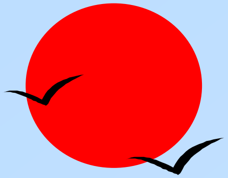
Dynamic SQL

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



Conclusions

Questions?



End of Chapter