LECTURE 8
STRUCTURED QUERY LANGUAGE (SQL)
SQL Basic Structure and Operations

Additional Basic Operations

Aggregate Queries

Nested Subqueries and Set Operations

Database Definition
  - Basic Types, User-defined Types/Domains
  - Creating, Altering, Destroying Relations
  - Integrity Constraints: Domain, Key, Foreign Key, General

Database Modification

Using SQL in Applications
The SQL DDL allows the specification of:

- The schema for each relation (attributes).
- The types of values associated with each attribute (i.e., the domain of values the attribute, such as string, number, date, etc.).
- Integrity constraints (ICs).
  - domain, key, foreign key, general
- The set of indices to be maintained for each relation.
- The physical storage structure of each relation on disk.
- Security and authorization information for each relation.
BASIC TYPES

- `char(n)`: Fixed length character string with length $n$.
- `varchar2(n)`: Variable-length character string with maximum length $n$.
- `int`: An integer (a finite subset of the integers that is machine-dependent).
- `smallint`: A small integer (a machine-dependent subset of the integer domain type).
- `number(p,d)`: A fixed point number with a total of $p$ digits (the precision) and $d$ digits to the right of the decimal point.
- `float(n)`: Floating point number, with user-specified precision of at least $n$ digits.
- `date`: A date containing a (4 digit) year, month and day of month.
- `time`: The time of day, in hours, minutes and seconds.
- `timestamp`: A combination of date and time.

☞ Some relational systems also allow user-defined types.
The `create table` command is used to define and create a relation.

- The **domain type** of each attribute needs to be specified.
  - A **default** value can be specified for an attribute (only used when no value is provided when inserting with attributes *explicitly* specified).
  - Null values are allowed in all the basic domain types.

☞ The domain type of an attribute is enforced by the DBMS whenever tuples are added or modified.

```
create table Student (
  studentId  char(8) not null,
  name       varchar2(45) not null,
  email      varchar2(15),
  birthdate  date not null,
  cga        number(3,2));
```

```
create table EnrollsIn (
  studentId  char(8) not null,
  courseId   char(8) not null,
  grade      number(4,1) default 0 not null);
```
The *alter table* command is used to add attributes to, modify attributes in or drop attributes from an existing relation.

**Example:**

```
alter table Student
  add firstYear int;
```

The schema is altered by adding a new attribute and extending every tuple in the current instance with a null value for the new attribute.

```
alter table Student
  drop column firstYear;
```

The schema is altered by dropping the attribute from the relation and deleting its value in every tuple.

The *drop table* command deletes *all* information about a relation (both *data and schema*).

**Example:**

```
drop table Student;
```
INTEGRITY CONSTRAINTS (IC)

An integrity constraint (IC) ensures that authorized changes to the database do not result in a loss of data consistency.

☞ An IC guards against accidental damage to the database.

- ICs are obtained from the requirements of the real-world application that is being described in the database relations.
  - An IC is a statement about all possible instances!
  - For the Student relation, we know, from common knowledge, that name is not a key, but the constraint that an attribute, such as studentId, is a key must be given to us by the client.

- We can check a database instance to see if an IC is violated, but we can never infer that an IC is true by looking at a database instance. Why?
Domain constraints define **valid values** for attributes and are used to **test values** inserted into the database and **test queries** to ensure that the comparisons make sense.

Besides a basic domain type, additional constraints can be specified on attributes in the **create table** command.

- **not null** specifies that null values are **not allowed**.
- **primary key** specifies a key for a relation (the value of a key attribute **cannot be null** ⇒ no need to specify **not null**).
- **unique** specifies that an attribute or a set of attributes is a candidate key (the attribute value(s) **can be null**).
- **foreign key** specifies that one or more attributes refer to a primary key attribute in another relation.
- **check** specifies a predicate that the values in every tuple of the relation must satisfy.
FOREIGN KEY CONSTRAINT

A foreign key is a set of attributes in one relation whose values must match the primary key values in another relation or be null.

☞ A foreign key must reference the primary key of the referenced relation.

Example: Only students listed in the Student relation should be allowed to enroll for courses.

create table Student (  
    studentld   char(8) primary key,  
    name        varchar2(45) not null,  
    email       varchar2(30),  
    birthdate   date not null,  
    cga         number(3,2),  
    unique (email));

create table EnrollsIn (  
    studentld   char(8),  
    coursedld   char(8),  
    grade       number(4,1) default 0 not null,  
    primary key (studentld, coursedld),  
    foreign key (studentld) references Student(studentld));

☞ Every studentld value in the EnrollsIn relation must reference a tuple in the Student relation with a matching studentld value.
FOREIGN KEY: ENFORCING REFERENTIAL INTEGRITY

- What should be done if an EnrollIn tuple with a non-existent student id is inserted?
  
  ✅ Reject it!

- What should be done if a Student tuple is deleted?
  
  1. Disallow deletion of a Student tuple that is referred to by an EnrollIn tuple (default action).
  2. Alternatively, delete all EnrollIn tuples that refer to it (on delete cascade).
  3. Set studentId in EnrollIn tuples that refer to it to a default value (on delete set default).
  4. Set studentId in EnrollIn tuples that refer to it to a null value (on delete set null).

  ✅ 3 and 4 are not applicable in the example since studentId is part of the primary key.
FOREIGN KEY:
ENFORCING REFERENTIAL INTEGRITY (cont’d)

- What should be done if the primary key student id of a tuple in Student is updated?

  Reject it!

- Alternatively, propagate the update to the tuples in the EnrollsIn relation with matching student ids (on update cascade).

```
create table EnrollsIn (  
    studentId   char(8),
    courseId    char(10),
    grade       number(4,1) default 0 not null,
    primary key (studentId, courseId),
    foreign key (studentId) references Student(studentId)
      on delete cascade
      on update cascade);
```

The referential integrity actions in the referencing relation (EnrollsIn) are triggered when a tuple in the referenced relation (Student) is deleted or updated.

Oracle Note
Oracle does not support on update cascade.
CHECK CLAUSE: ATTRIBUTES

- The **check** clause is used to add an integrity constraint for an attribute and can contain an **arbitrary predicate**.
  
  The predicates are similar to those allowed in a **where clause**.

- The predicate is specified in the definition of a relation and checked whenever there is an update to the relation.

**Example:** Ensure that semester can have only specified values and that year is between 2020 and 2024.

```sql
create table Section (  
courseId char(8),  
sectionId char(2),  
semester char(6),  
year char(4) check (year between '2020' and '2024'),  
check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```
STRUCTURED QUERY LANGUAGE (SQL)

EXERCISE 1
BOOK STORE RELATIONAL SCHEMA

Book(bookId, title, subject, quantityInStock, price, authorId)

Author(authorId, firstName, lastName)

Customer(customerId, firstName, lastName)

BookOrder(orderId, customerId, orderYear)

OrderDetails(orderId, bookId, quantity)

Assumptions

– Each author has authored at least one book in the store.
– Each book has exactly one Author.
– Each order is made by exactly one customer and has one or more associated tuples in OrderDetails (e.g., one order may contain several different books).
EXERCISE 1

Given the foreign keys of the Book Store relations and assuming the referential integrity constraints are included in the SQL create statements, what should be the create order?

Book (bookId, title, subject, quantityInStock, price, authorId)
Author (authorId, firstName, lastName)
Customer (customerId, firstName, lastName)
BookOrder (orderId, customerId, orderYear)
OrderDetails (orderId, bookId, quantity)

Table | Possible create order
--- | ---
Author | 1 1 2 2 1 3
Customer | 2 2 1 1 3 1
Book | 3 4 3 4 2 4
BookOrder | 4 3 4 3 4 2
OrderDetails | 5 5 5 5 5 5

Create Order

| Table | Order
|---|---
| Author | before Book
| Customer | before BookOrder
| Book, BookOrder | before OrderDetails
STRUCTURED QUERY LANGUAGE (SQL): OUTLINE

✓ SQL Basic Structure and Operations
✓ Additional Basic Operations
✓ Nested Subqueries and Set Operations
✓ Aggregate Queries
✓ Database Definition

➡️ Database Modification
  – Deleting Tuples
  – Inserting Tuples
  – Updating Tuples

Using SQL in Applications
The **delete** command deletes *zero or more tuples* from a relation.

**Example:** Delete all accounts at the Pacific Place branch.

```
delete from Account
where branchName='Pacific Place';
```

A **delete** statement **where** clause predicate can be as complex as in a **select** statement.

**Example:** Delete all depositors at the Langham Place branch.

```
delete from Depositor
where accountNo in (select accountNo
from Depositor natural join Account
where branchName= 'Langham Place');
```

Must also delete the accounts of these depositors!

Can only delete if **no integrity constraints are violated**!
TUPLE INSERTION

- The `insert` command adds one or more tuples to a relation.

**Example:** Add a new Account.

```
insert into Account values ('A-732', 1200, 'Pacific Place');
```

**Example:** Add a new Account with balance set to null.

```
insert into Account values ('A-733', null, 'Pacific Place');
```

☞ The order of the values must match the order of the attributes in the relation.

- Attribute names need to be specified explicitly for order-independent insertion and to make use of default values.

```
insert into Account (accountNo, branchName, balance) values ('A-734', 'Pacific Place', 1200);
```
Insertion values can be obtained from the result of a query.

**Example:** Create a $200 savings account for all loan clients of the Pacific Place branch. Let the loan number serve as the account number for the new savings account.

```
insert into Account
select loanNo, 200, branchName
from Loan
where branchName='Pacific Place';
```

```
insert into Depositor
select clientId, loanNo
from Loan natural join Borrower
where branchName='Pacific Place';
```

**Note:** The keyword values is omitted when the values are obtained from a select statement.
TUPLE UPDATE

- The update command is used to change a value in a tuple.

**Example:** Increase all accounts with balance over $10,000 by 6%; all other accounts receive 5%.

```sql
update Account
set balance = balance * 1.06
where balance > 10000;
```

```sql
update Account
set balance = balance * 1.05
where balance <= 10000;
```

Need two update statements! The order is important! Why?

- This update can be specified using the case statement.

```sql
update Account
set balance = case
    when balance <= 10000 then balance * 1.05
    else balance * 1.06
end;
```
STRUCTURED QUERY LANGUAGE (SQL)

EXERCISES 2, 3
BOOK STORE RELATIONAL SCHEMA

Book(\textit{bookId}, title, subject, quantityInStock, price, \textit{authorId})

Author(\textit{authorId}, firstName, lastName)

Customer(\textit{customerId}, firstName, lastName)

BookOrder(\textit{orderld}, \textit{customerId}, orderYear)

OrderDetails(\textit{orderld}, \textit{bookId}, quantity)

\textbf{Assumptions}

- Each author has authored at least one book in the store.
- Each book has exactly one Author.
- Each order is made by exactly one customer and has one or more associated tuples in OrderDetails (e.g., one order may contain several different books).
For all authors who wrote books on at least two subjects, increase the price of all their books by 5%.

\[
\text{update Book}
\]
\[
\text{set price}=1.05*\text{price}
\]
\[
\text{where authorId in (select B1.authorId}
\]
\[
\text{from Book B1, Book B2}
\]
\[
\text{where B1.authorId=B2.authorId}
\]
\[
\text{and B1.subject<>B2.subject)}
\]

Note: Natural join cannot be used if self join is required. Why?
For all authors who wrote books on at least two subjects, increase the price of all their books by 5%.

```sql
update Book
set price = 1.05 * price
where authorId in (select authorId
from Book
group by authorId
having count(distinct subject) >= 2);
```

Update the price by 5%.

Authors who wrote books on more than one subject.
Exercise 3

Find the last name and first name of all authors who wrote books on both the subjects of Art and Business.

Can we say \( \Rightarrow \text{where subject='Art' and subject='Business'} \)? Why?

\( \Rightarrow \) Selects nothing.

Can we say \( \Rightarrow \text{where subject='Art' or subject='Business'} \)? Why?

\( \Rightarrow \) Selects authors who wrote either Art or Business books, but not necessarily on both subjects.

\[
\text{select lastName, firstName from (select authorId, lastName, firstName from Author natural join Book where subject='Art'} \text{intersect select authorId, lastName, firstName from Author natural join Book where subject='Business'}); \]

Authors who wrote books on Art.

Select only those authors in the Art set who are also in the Business set.

Authors who wrote books on Business.
Find the last name and first name of all authors who wrote books on both the subjects of Art and Business.

```sql
SELECT lastName, firstName
FROM Author NATURAL JOIN Book
WHERE subject='Art'
AND authorId IN (SELECT authorId
FROM Author NATURAL JOIN Book
WHERE subject='Business');
```

Authors who wrote books on Art (Art set).

Select only those authors in the Art set who are also in the Business set.

Authors who wrote books on Business (Business set).
STRUCTURED QUERY LANGUAGE (SQL): OUTLINE

✓ SQL Basic Structure and Operations
✓ Additional Basic Operations
✓ Aggregate Queries
✓ Nested Subqueries and Set Operations
✓ Database Definition
✓ Database Modification

⇒ Using SQL in Applications
  – Database APIs
  – Oracle PL/SQL
To utilize DBMS services, client applications use a specific application programming interface (API) provided by the DBMS.

- Facebook, Google, Instagram, etc. have such APIs.
- Proprietary versus generic APIs (e.g., ODBC, JDBC, ADO.NET).

The DBMS API exposes an interface through which the services provided by the DBMS can be accessed.

- The client and server interfaces often are implemented in the form of network sockets that use a specific port number on the server (e.g., port 1521 for the course Oracle Database server).
EMBEDDED VS CALL-LEVEL API

Embedded API

- SQL statements are part of the host programming language source code.

- An SQL pre-compiler parses and checks the SQL instructions before the program is compiled and replaces these with source code instructions native to the host programming language used.

Call-level API

- Passes SQL instructions to the DBMS by direct calls to a series of procedures, functions or methods provided by the API.

- The calls perform actions such as setting up a database connection, sending queries and iterating over the query result.
EARLY VS LATE BINDING

- SQL binding is the translation of SQL statements in a programming language into a form that can be executed by the DBMS.
  - Involves performing tasks such as validating table and attribute names, checking whether the user or client has sufficient access rights and generating an efficient query plan to access the data.

- Early binding performs these tasks *only once before program execution* (i.e., using a pre-compiler with an embedded API).

- Late binding performs these tasks *every time at runtime* (i.e., when using a call-level API).

☞ It is still possible to do early binding using call-level APIs by using stored procedures in the DBMS.
• PL/SQL (Procedural Language/SQL) allows SQL statements to be embedded into a procedural programming language.

• A PL/SQL program is stored as a database object (stored procedure/function) and can be
  – a procedure, which does not return a value and is invoked using the exec keyword.
  – a function, which returns a value using the return keyword and is invoked by assigning its result to a variable or using it in a select statement.

• Both types of PL/SQL programs can accept parameters.
ORACLE PL/SQL: BASIC STRUCTURE

- The basic processing unit is a **block**, which is delimited by `begin...end` and which can be nested.

| create or replace procedure procedure_name [ as | is ] |
|-----------------------------------------------|
| Declaration section: contains declaration of variables, types, and local subprograms. |
| begin |
| Executable section: contains procedural and SQL statements. This is the only section of a block that is required. |
| exception |
| Exception handling section: contains error handling statements. |
| end; |

**Allowed SQL statements**: `select, insert, update, delete` (i.e., DML)

**Not allowed SQL statements**: `create, drop, alter, rename` (i.e., DDL)
Increment the rating of a sailor if the rating is less than 5.

```sql
create or replace procedure L9Example1 (sid in int) as

    sailorName Sailors.sName%type;
    sailorRating Sailors.rating%type;

begin
    -- Fetch the sailor's name and rating into the variables sailorName and sailorRating
    select sName, rating into sailorName, sailorRating from Sailors where sailorId=sid;

    if sailorRating<5 then
        update Sailors set rating=sailorRating+1 where sailorId=sid;

        -- Write record updated message to the Script Output tab
        dbms_output.put_line('Sailor ' || sailorName || '(' || sid || ') rating updated from ' || sailorRating || ' to ' || (sailorRating+1) || '.');
    else
        -- Write record NOT updated message to the Script Output tab
        dbms_output.put_line('Sailor ' || sailorName || '(' || sid || ') rating ' || sailorRating || ' NOT updated.');
    end if;
end L9Example1;
```

Local variables `sailorName` and `sailorRating` are of the same type as `sName` and `rating` in the `Sailor` relation.

Must fetch at most one record.
Procedural programming languages normally process only one record at a time.

Thus, if a `select` statement returns more than one record, a cursor is needed to process the records one-at-a-time.
- A cursor is like a pointer that points to a single record in a query result and allows access to the attribute values of that record.

In PL/SQL a cursor is defined in the `declare` section

```plsql
cursor cursor_name is select_statement;
```

and can be used and managed
- *explicitly* using the `open`, `fetch` and `close` commands and by checking cursor status.
- *implicitly* using the `for...loop` statement where the `cursor_name` replaces the range limit so the loop ranges from the first record of the cursor to the last record of the cursor.
create or replace procedure L9Example2 as
    currentSailorId  Sailor.sailorId%type;
    -- Declare the cursors for the sailor and reserves tables
    cursor sailorCursor is select * from Sailor order by sName;
    cursor reservesCursor is select count(boatid) reservations from reserves where sailorId=currentSailorId;
begin
    -- Fetch the sailorCursor records one-by-one
    for sailorRecord in sailorCursor loop
        -- Assign the sailor id for the current sailor record
        currentSailorId:=sailorRecord.sailorId;
        -- Fetch the reservesCursor records one-by-one
        for reservesRecord in reservesCursor loop
            -- Insert into appropriate table
            if reservesRecord.reservations=0 then
                insert into ReservationSummary values (sailorRecord.sailorId, sailorRecord.sName, 'No');
            else
                insert into ReservationSummary values (sailorRecord.sailorId, sailorRecord.sName, 'Yes');
            end if;
        end loop;
    end loop;
end L9Example2;
PL/SQL EXCEPTIONS

- Predefined exceptions are raised implicitly by PL/SQL if the exception occurs.

- User-defined exceptions are declared in the declaration section,

  ```plsql
decl
    exception_name exception;
end;
```

  raised explicitly within a `begin...end` block

  ```plsql
  if condition then
    raise exception_name;
  end if;
  ```

  and handled in the `exception` section within the `begin...end` block.

  ```plsql
  exception
    when exception_name then
  ```

---

**Predefined Exceptions**

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_INTO_NULL</td>
<td>ORA-06530</td>
</tr>
<tr>
<td>CASE_NOT_FOUND</td>
<td>ORA-06592</td>
</tr>
<tr>
<td>COLLECTION_IS_NULL</td>
<td>ORA-06531</td>
</tr>
<tr>
<td>CURSOR_ALREADY_OPEN</td>
<td>ORA-06511</td>
</tr>
<tr>
<td>DUP_VAL_ON_INDEX</td>
<td>ORA-00001</td>
</tr>
<tr>
<td>INVALID_CURSOR</td>
<td>ORA-01001</td>
</tr>
<tr>
<td>INVALID_NUMBER</td>
<td>ORA-01722</td>
</tr>
<tr>
<td>LOGIN_DENIED</td>
<td>ORA-01017</td>
</tr>
<tr>
<td>NO_DATA_FOUND</td>
<td>ORA-01403</td>
</tr>
<tr>
<td>NOT_LOGGED_ON</td>
<td>ORA-01012</td>
</tr>
<tr>
<td>PROGRAM_ERROR</td>
<td>ORA-06501</td>
</tr>
<tr>
<td>ROWTYPE_MISMATCH</td>
<td>ORA-06504</td>
</tr>
<tr>
<td>SELF_IS_NULL</td>
<td>ORA-30625</td>
</tr>
<tr>
<td>STORAGE_ERROR</td>
<td>ORA-06500</td>
</tr>
<tr>
<td>SUBSCRIPT_BEYOND_COUNT</td>
<td>ORA-06533</td>
</tr>
<tr>
<td>SUBSCRIPT_OUTSIDE_LIMIT</td>
<td>ORA-06532</td>
</tr>
<tr>
<td>SYS_INVALID_ROWID</td>
<td>ORA-01410</td>
</tr>
<tr>
<td>TIMEOUT_ON_RESOURCE</td>
<td>ORA-00051</td>
</tr>
<tr>
<td>TOO_MANY_ROWS</td>
<td>ORA-01422</td>
</tr>
<tr>
<td>VALUE_ERROR</td>
<td>ORA-06502</td>
</tr>
<tr>
<td>ZERO_DIVIDE</td>
<td>ORA-01476</td>
</tr>
</tbody>
</table>
create or replace procedure L9Example3 (sid in int) as
    sailorName Sailor.sName%type;
    sailorRating Sailor.rating%type;
begin
    -- Fetch the sailor’s name and rating into the variables sailorName and sailorRating
    select sName, rating into sailorName, sailorRating from Sailor where sailorId=sid;
    if sailorRating<5 then
        update Sailor set rating=sailorRating+1 where sailorId=sid;
        -- Write record updated message to the Script Output tab
        dbms_output.put_line('Sailor ' || sailorName || '(' || sid || ') rating updated from ' || sailorRating || ' to ' || (sailorRating+1) || '.');
    else
        -- Write record NOT updated message to the Script Output tab
        dbms_output.put_line('Sailor ' || sailorName || '(' || sid || ') rating ' || sailorRating || ' NOT updated.');
    end if;
exception
    when no_data_found then
        -- Write exception message to the Script Output tab
        dbms_output.put_line('There is no sailor with id ' || sid || '.');
end L9Example3;

If the sailor id does not exist, then the no_data_found exception is raised causing execution to pass to the exception section and to the no_data_found exception code.
Structured Query Language (SQL) is a relational query language that provides facilities to:

**Query Relations**
- Select-From-Where Statement
- Set Operations (Union, Intersect, Except)
- Nested Subqueries (to test for set membership, comparison, cardinality)
- Aggregate Functions (avg, min, max, sum, count)
- Group By with Having clause

**Create and Modify Relations**
- Create, Alter, Drop Tables
- Specify integrity constraints: domain, key, foreign key, general
- Insert, Delete, Update Tuples

**Access a Database from a Programming Language**
COMP 3311: SYLLABUS

- Introduction
- Entity-Relationship (E-R) Model and Database Design
- Relational Algebra
- Structured Query Language (SQL)
- Relational Database Design
  ➔ Storage and File Structure
    - Indexing
    - Query Processing
    - Query Optimization
    - Transactions
    - Concurrency Control
    - Recovery System
    - NoSQL Databases
Upload your completed exercise worksheet to Canvas by 11 p.m. Feb 26th