Chapter 7

SQL: Data Definition
Main SQL DDL Directives

• Create and alter tables, including specification of fields and their data types
• Specify integrity constraints on data
• Define and maintain views
## ISO SQL Data Types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>character</td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
</tr>
<tr>
<td>bit</td>
<td>BIT</td>
</tr>
<tr>
<td></td>
<td>BIT VARYING</td>
</tr>
<tr>
<td>exact numeric</td>
<td>NUMERIC</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
</tr>
<tr>
<td>approximate numeric</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>datetime</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>interval</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>large objects</td>
<td>CHARACTER LARGE OBJECT</td>
</tr>
<tr>
<td></td>
<td>BINARY LARGE OBJECT</td>
</tr>
</tbody>
</table>
• Normal true/false values
• There is a 3rd truth value called UNKNOWN which is represented as NULL
Character Data

• Examples
  – branchNo CHAR(4): a fixed width string 4 characters long
  – address VARCHAR(30): a variable length string of up to 30 characters

• If a string is fixed length, then a short string will be padded to the right with blank spaces

• CHAR and VARCHAR can store up to 255 character strings

• Use TEXT to store longer length strings (up to 65,535 characters)

• See mysql documentation for representing even longer strings
Numeric Data

• Precise real numbers: NUMERIC or DECIMAL (DEC)-they are identical
  – Example: salary DECIMAL(7,2): stored as a string but can be manipulated like a number and aggregation functions like avg and sum work with it
    • Precision: The first number represents the total number of digits
    • Scale: The second number represents the total number of decimal digits
    • salary can store any number up to 99,999.99

• Integers: INTEGER (INT) or SMALLINT: Use SMALLINT to conserve space when your field values are small
Numeric Data (cont.)

• Approximate numeric data: Like C/C++, real numbers in these formats can only be approximated
  – Different types
    • FLOAT(precision, decimaldigits): a small decimal number
      – precision controls the total number of digits
      – decimaldigits controls number of digits to right of decimal point
    • DOUBLE(precision, decimaldigits): a large decimal number
Dates

- **Date: allows you to store a date**
  - **Three types of dates**
    - DATE: Stores calendar dates using year, month, and day
      - format YYYY-MM-DD
    - TIME [timePrecision]: Stores time as hours, minutes, and seconds
      - timePrecision specifies number of decimal digits for the seconds field
      - format: HH:MM:SS
    - TIMESTAMP[timePrecision]: Stores date and times
      - format: YYYYY-MM-DD HH:MM:SS
  - **Handy date functions in mysql**
    - NOW(): Returns the current date and time
    - CURDATE(): Returns the current date
    - CURTIME(): Returns the current time
    - DATE(): Extracts the date part of a date or date/time expression
    - DATE_ADD(): Adds a specified time interval to a date
    - DATE_SUB(): Subtracts a specified time interval from a date
    - DATEDIFF(): Returns the number of days between two dates
Intervals

• Interval: allows you to specify a time interval, either as year-month intervals or day-time intervals (day-time allows days, hours, minutes, and seconds)
  – Examples
    • INTERVAL YEAR(2) TO MONTH represents an interval of time from 0 years 0 months to 99 years 11 months
    • INTERVAL HOUR TO SECOND(4) represents an interval between 0 hours 0 minutes 0 seconds to 99 hours 59 minutes 59.9999 seconds
Integrity Constraints

• Consider five types of integrity constraints:
  – required data: Whether a data field must contain a value
  – domain constraints: A set of legal values for a field
  – entity integrity: Each primary key of a table must contain a unique, non-null value
  – referential integrity: Foreign keys must refer to a valid, existing row in the parent relation
  – general constraints: General, organization specific constraints, such as requiring that no staff member handle 100 properties

• These constraints are defined in the CREATE TABLE and ALTER TABLE commands
Integrity Constraints

**Required Data:** SQL’s “NOT NULL” command requires that a field have a defined value

- position VARCHAR(10) NOT NULL

**Domain Constraints**

(a) **CHECK:** CHECK clause allows you to make an assertion about the types of values that may appear in a column

(a) **general form:** CHECK(SEARCH CONDITION)

(b) **Example:**

- sex CHAR NOT NULL
  - CHECK (sex IN ('M', 'F'))
Domain Constraints

(b) CREATE DOMAIN: allows you to define a restricted domain of values

```
CREATE DOMAIN DomainName [AS] dataType
[DEFAULT defaultOption]
[CHECK (searchCondition)]
```

For example:

```
CREATE DOMAIN SexType AS CHAR
  CHECK (VALUE IN ('M', 'F'));
sex SexType  NOT NULL
```
**Domain Constraints**

- *searchCondition* can involve a table lookup:

```sql
CREATE DOMAIN BranchNo AS CHAR(4)
CHECK (VALUE IN (SELECT branchNo
                FROM Branch));
```
Entity Integrity

• Primary key of a table must contain a unique, non-null value for each row.
• ISO standard supports PRIMARY KEY clause in CREATE and ALTER TABLE statements:
  \[ \text{PRIMARY KEY(staffNo)} \]
  \[ \text{PRIMARY KEY(clientNo, propertyNo)} \]
• Can only have one PRIMARY KEY clause per table. Can still ensure uniqueness for alternate keys using UNIQUE:
  \[ \text{UNIQUE(telNo)} \]
• Any field declared UNIQUE must also be declared to be NOT NULL
Referential Integrity

• A **foreign key** is a column or set of columns that links each row in child table to a row of parent table containing a matching primary key.

• **Referential integrity** means that, if a foreign key contains a value, that value must refer to an existing row in the parent table.

• ISO standard supports definition of foreign keys with FOREIGN KEY clause in CREATE and ALTER TABLE:

  FOREIGN KEY(branchNo) REFERENCES Branch
Referential Integrity

- Any INSERT/UPDATE attempting to create a foreign key value in the child table without matching primary key value in the parent is rejected.
- Action taken attempting to update/delete a primary key value in the parent table with matching rows in child is dependent on referential action specified using ON UPDATE and ON DELETE subclauses:
  1. **CASCADE**: Delete row from parent and delete matching rows in child, and so on in cascading manner.
  2. **SET NULL**: Delete row from parent and set FK column(s) in child to NULL. Only valid if FK columns are allowed to be NULL.
  3. **SET DEFAULT**: Delete row from parent and set each component of FK in child to specified default. Only valid if DEFAULT specified for FK columns.
  4. **NO ACTION**: Reject delete from parent. Default.
Referential Integrity

Examples:

FOREIGN KEY (staffNo) REFERENCES staff
  ON DELETE SET NULL
FOREIGN KEY (ownerNo) REFERENCES privateOwner
  ON UPDATE CASCADE
General Constraints

• Use CHECK command outside a field definition
  – Example

    CONSTRAINT StaffNotHandlingTooMuch
    CHECK (NOT EXISTS (SELECT staffNo
                        FROM PropertyForRent
                        GROUP BY staffNo
                        HAVING COUNT(*) > 100))

    The CONSTRAINT keyword names the constraint so that it can be dropped by an ALTER TABLE statement
Limitations on Integrity Checking in MYSQL

- CREATE DOMAIN is not supported in MYSQL
- CHECK and FOREIGN KEY...REFERENCES are parsed but not enforced by MYSQL
- The **ENUM** type is enforced
  
  Example: `sex ENUM('M', 'F');`
Data Definition

- SQL DDL allows database objects such as schemas, domains, tables, views, and indexes to be created and destroyed.
- Main SQL DDL statements that we will consider are:
  - CREATE/ALTER TABLE
  - DROP TABLE
  - CREATE VIEW
  - DROP VIEW
- Many DBMSs also provide:
  - CREATE INDEX
  - DROP INDEX
CREATE TABLE

CREATE TABLE TableName
{(colName dataType [NOT NULL] [UNIQUE]
  [DEFAULT defaultOption]
  [CHECK searchCondition] [,...])
[PRIMARY KEY (listOfColumns),]
{[UNIQUE (listOfColumns),] [...,]}
{[FOREIGN KEY (listOfFKColumns)
  REFERENCES ParentTableName [(listOfCKColumns)],
  [ON UPDATE referentialAction]
  [ON DELETE referentialAction ]] [,...]}[[CHECK (searchCondition)] [,... ])}
CREATE TABLE

- Creates a table with one or more columns of the specified *dataType*.
- With NOT NULL, system rejects any attempt to insert a null in the column.
- Can specify a DEFAULT value for the column.
- Primary keys should always be specified as NOT NULL.
- FOREIGN KEY (FK) clause specifies
  - the foreign key
  - the parent relation
  - actions to perform when the associated primary key is updated/deleted from the parent relation.
Example: Create PropertyForRent Table

PropertyForRent(propertyNo, street, city, postcode, propertyType, rooms, rent, ownerNo, staffNo, branchNo)

CREATE DOMAIN OwnerNumber AS VARCHAR(5)
    CHECK (VALUE IN (SELECT ownerNo FROM PrivateOwner));
CREATE DOMAIN StaffNumber AS VARCHAR(5)
    CHECK (VALUE IN (SELECT staffNo FROM Staff));
CREATE DOMAIN PNumber AS VARCHAR(5);
CREATE DOMAIN PRooms AS SMALLINT;
    CHECK(VALUE BETWEEN 1 AND 15);
CREATE DOMAIN PRent AS DECIMAL(6,2)
    CHECK(VALUE BETWEEN 0 AND 9999.99);
Example (cont): Create PropertyForRent Table

CREATE TABLE PropertyForRent (  
  propertyNo PNumber NOT NULL,
  ....
  rooms PRooms NOT NULL DEFAULT 4,
  rent PRent NOT NULL DEFAULT 600,
  ownerNo OwnerNumber NOT NULL,
  staffNo StaffNumber
          Constraint StaffNotHandlingTooMuch ....
  branchNo BranchNumber NOT NULL,
  PRIMARY KEY (propertyNo),
  FOREIGN KEY (staffNo) REFERENCES Staff
          ON DELETE SET NULL
  ON UPDATE CASCADE ....);
ALTER TABLE

- Add a new column to a table.
- Drop a column from a table.
- Add a new table constraint.
- Drop a table constraint.
- Set a default for a column.
- Drop a default for a column.
Example: ALTER TABLE

Change Staff table by removing default of ‘Assistant’ for position column and setting default for sex column to female (‘F’).

ALTER TABLE Staff
  ALTER position DROP DEFAULT;

ALTER TABLE Staff
  ALTER sex SET DEFAULT ‘F’;
Example: ALTER TABLE

Remove constraint from PropertyForRent that staff are not allowed to handle more than 100 properties at a time. Add new column to Client table.

ALTER TABLE PropertyForRent
    DROP CONSTRAINT StaffNotHandlingTooMuch;
ALTER TABLE Client
    ADD prefNoRooms PRooms;
DROP TABLE

DROP TABLE TableName [RESTRICT | CASCADE]

e.g. DROP TABLE PropertyForRent;

• Removes named table and all rows within it.
• With RESTRICT, if any other objects depend for their existence on continued existence of this table, SQL does not allow request.
• With CASCADE, SQL drops all dependent objects (and objects dependent on these objects).
Indices

• An *index* can help the query optimizer quickly locate rows associated with a certain condition
  
  – *Example: Suppose the query is:*

  ```sql
  SELECT fName, lName, salary
  FROM Staff;
  WHERE lName = 'White';
  ```

• If there is an index on lName, then the query optimizer can retrieve only those rows that contain ‘White’ as a last name

• If there is no index on lName, then the query optimizer must retrieve every row in the relation, check whether the last name in the row is ‘White’, and if so, output it
Index Creation

• **Syntax:**
  
  CREATE INDEX IndexName
  
  ON TableName (columnName [ASC | DESC] [, ...])
  
  –Default order is ASC

• **Examples**
  
  –CREATE INDEX StaffNoIndex ON Staff(staffNo);
  
  –CREATE INDEX PropertyNoIndex
    
    ON PropertyForRent (propertyNo DESC);
  
  –CREATE INDEX RentIndex ON PropertyForRent(city, rent);

  • Entries in the index will be in alphabetical order by city and then by rent within each city
Views

• **Definition:** Virtual relation that does not necessarily actually exist in the database but is produced upon request, at time of request.

• **Example:** A view that shows how many properties are managed by each staff member:

<table>
<thead>
<tr>
<th>branchNo</th>
<th>staffNo</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>SG14</td>
<td>7</td>
</tr>
<tr>
<td>B003</td>
<td>SG37</td>
<td>23</td>
</tr>
<tr>
<td>B005</td>
<td>SL41</td>
<td>10</td>
</tr>
</tbody>
</table>
Views

• Contents of a view are defined as a query on one or more base relations.

• Alternative system approaches to supporting views
  – **view resolution**: any operations on the view are automatically translated into operations on the relations from which it is derived.
  – **view materialization**: the view is stored as a temporary table, which is maintained as the underlying base tables are updated.
**Horizontal View**

- A horizontal view restricts a user’s access to selected rows of one or more tables
- Example: Create view so that manager at branch B003 can only see details for staff who work in his or her office.

```sql
CREATE VIEW Manager3Staff
AS
SELECT * FROM Staff
WHERE branchNo = 'B003';
```

**Table 6.3** Data for view Manager3Staff.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
<th>sex</th>
<th>DOB</th>
<th>salary</th>
<th>branchNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>Assistant</td>
<td>F</td>
<td>10-Nov-60</td>
<td>12000.00</td>
<td>B003</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>Supervisor</td>
<td>M</td>
<td>24-Mar-58</td>
<td>18000.00</td>
<td>B003</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>Manager</td>
<td>F</td>
<td>3-Jun-40</td>
<td>24000.00</td>
<td>B003</td>
</tr>
</tbody>
</table>
Vertical View

• A **vertical view** restricts a user’s access to selected columns of one or more tables

• Example: Create view of staff details at branch B003 excluding salaries.

```
CREATE VIEW Staff3
AS SELECT staffNo, fName, lName, position, sex FROM Staff
WHERE branchNo = 'B003';
```

Table 6.4  Data for view Staff3.

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>Assistant</td>
<td>F</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>Supervisor</td>
<td>M</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>Manager</td>
<td>F</td>
</tr>
</tbody>
</table>
Views that Use Aggregate Operations and Draw Data From Multiple Views

Create a view of staff who manage properties for rent, including branch number they work at, staff number, and number of properties they manage.

CREATE VIEW StaffPropCnt (branchNo, staffNo, cnt)
AS SELECT s.branchNo, s.staffNo, COUNT(*)
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo
GROUP BY s.branchNo, s.staffNo;

Table 6.5  Data for view StaffPropCnt.

<table>
<thead>
<tr>
<th>branchNo</th>
<th>staffNo</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>SG14</td>
<td>1</td>
</tr>
<tr>
<td>B003</td>
<td>SG37</td>
<td>2</td>
</tr>
<tr>
<td>B005</td>
<td>SL41</td>
<td>1</td>
</tr>
<tr>
<td>B007</td>
<td>SA9</td>
<td>1</td>
</tr>
</tbody>
</table>
SQL – View Creation Syntax

CREATE VIEW ViewName [ (newColumnName [, ...]) ] 
  AS subselect [WITH CHECK OPTION]

- Can assign a name to each column in view.
- If list of column names is specified, it must have same number of items as number of columns produced by subselect.
- If omitted, each column takes name of corresponding column in subselect.
• List must be specified if there is any ambiguity in a column name (e.g., an aggregated value produced by the SELECT does not have a name, so it must be given one)

• The subselect is known as the defining query.

• WITH CHECK OPTION prevents a row from being inserted into the view if it violates the WHERE condition of the defining query

• Need SELECT privilege on all tables referenced in subselect and USAGE privilege on any domains used in referenced columns.
View Resolution

- It is permissible to write queries on views
  - view resolution refers to the process of merging such a query with the defining query for the view
  - Example: Count number of properties managed by each member at branch B003.

SELECT staffNo, cnt
FROM StaffPropCnt
WHERE branchNo = ‘B003’
ORDER BY staffNo;
View Resolution

(a) View column names in SELECT list are translated into their corresponding column names in the defining query:

```
SELECT s.staffNo As staffNo, COUNT(*) As cnt
```

(b) View names in FROM are replaced with corresponding FROM lists of the defining query:

```
FROM Staff s, PropertyForRent p
```
View Resolution

(c) WHERE from the user query is combined with the WHERE from the defining query using AND:

\[
\text{WHERE s.staffNo = p.staffNo AND branchNo = 'B003'}
\]

(d) GROUP BY and HAVING clauses are copied from the defining query:

\[
\text{GROUP BY s.branchNo, s.staffNo}
\]

(e) ORDER BY copied from the user query with the view column name translated into the defining query column name

\[
\text{ORDER BY s.staffNo}
\]
(f) Final merged query is now executed to produce the result:

```sql
SELECT s.staffNo AS staffNo, COUNT(*) AS cnt
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo AND
    branchNo = 'B003'
GROUP BY s.branchNo, s.staffNo
ORDER BY s.staffNo;
```
SQL Restrictions on Views

(a) If column in view is based on an aggregate function:
   – Column may appear only in SELECT and ORDER BY clauses of queries that access view.
   – Column may not be used in WHERE nor be an argument to an aggregate function in any query based on view.

For example, the following queries would fail:

```
SELECT COUNT(cnt) FROM StaffPropCnt;
SELECT * FROM StaffPropCnt WHERE cnt > 2;
```
Restrictions on Views

(b) Grouped view (i.e., a view with an aggregate function) may never be joined with a base table or a view.

– For example, StaffPropCnt view is a grouped view, so any attempt to join this view with another table or view fails.
View Updatability

- All updates to base table are reflected in all views that encompass the base table.
- Similarly, you may expect that if the view is updated then the base table(s) will reflect change.
  - Update Restriction: For view to be updatable, DBMS must be able to trace any row or column back to its row or column in the source table.
Examples of Non-Updateable Views

- Consider the view StaffPropCnt.
- If we tried to insert record showing that at branch B003, SG5 manages 2 properties:

  ```sql
  INSERT INTO StaffPropCnt
  VALUES ('B003', 'SG5', 2);
  ```

- Have to insert 2 records into PropertyForRent showing which properties SG5 manages. However, do not know which properties they are; i.e. do not know primary keys!
View Updatability

• If change definition of view and replace count with actual property numbers:

```sql
CREATE VIEW StaffPropList (branchNo, staffNo, propertyNo)
AS SELECT s.branchNo, s.staffNo, p.propertyNo
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo;
```

• Now try to insert the record:

```sql
INSERT INTO StaffPropList
VALUES (‘B003’, ‘SG5’, ‘PG19’);
```

–Still problem, because in PropertyForRent all columns except postcode/staffNo are not allowed nulls.
–However, have no way of giving remaining non-null columns values.
View Updatability

• ISO specifies that a view is updatable if and only if:
  1. DISTINCT is not specified in SELECT of defining query
  2. Every element in SELECT list of defining query is a column name and no column appears more than once.
  3. FROM clause specifies only one table: this restriction excludes any views based on a join, union, intersection or difference.
  4. The defining query contains no GROUP BY or HAVING clause.

• Additionally, every row added through view must not violate integrity constraints of base table.
Migrating Rows

• Rows that enter or leave a view are called *migrating rows*.

• Rows exist in a view because they satisfy WHERE condition of defining query.

• If a row changes and no longer satisfies condition, it disappears from the view.
Migrating Rows (cont)

• New rows appear within view when insert/update on view cause them to satisfy WHERE condition of the defining query.
  – The WITH CHECK OPTION clause prevents a row from being inserted/updated in the view if it violates the WHERE condition of the defining query.

• Example: Consider the following view
  CREATE VIEW Manager3Staff AS SELECT * FROM Staff
  WHERE branchNo = 'B003' WITH CHECK OPTION;

and the query
  INSERT INTO Manager3Staff
  VALUES('SL15', 'Mary', 'Assistant', 'F', Date '1967-06-21', 8000, 'B002');

The query will be rejected because the inserted branch number, ‘B002’, does not satisfy the WHERE condition of the defining query.
Advantages of Views

• **Data independence:** A view presents a consistent, unchanging picture of the structure of the database, even if the underlying source tables are changed.

• **Currency:** Changes to any of the base tables are immediately reflected in the view.

• **Improved security:** Views restrict and control what data a user is allowed to see.
Advantages of Views (Cont)

- **Reduced complexity:**
  - A view can simplify queries, by drawing data from several tables into a single table, thereby transforming multi-table queries into single-table queries
  - Users are presented with only that part of the database that they need to see

- **Customization:** Views allow different users to view the same underlying base tables in different ways

- **Data integrity:** The WITH CHECK OPTION ensures that a view cannot be updated with a row that violates the WHERE condition of the defining query
Disadvantages of Views

• **Update restriction:** In some cases a view cannot be updated
  
• **Structure restriction:** The structure of the view is determined by the defining query at the time the view is created. If the underlying base tables are altered, the view’s structure cannot be updated. Instead you must drop and then re-add the table
  
  – For example, if you create a view with a “SELECT *” and later add a column to the base relation, the new column will not appear in the view

• **Performance:** View resolution imposes a performance hit since it must be performed each time the view is requested
View Materialization

- View resolution mechanism may be slow, particularly if view is accessed frequently.
- View materialization stores view as temporary table when view is first queried.
- Thereafter, queries based on materialized view can be faster than recomputing view each time.
View Maintenance

- Difficulty is maintaining the currency of the view while base tables(s) are being updated.
- View maintenance aims to apply only those changes necessary to keep view current.
- Consider following view:

  ```sql
  CREATE VIEW StaffPropRent(staffNo)
  AS SELECT DISTINCT staffNo
  FROM PropertyForRent
  WHERE branchNo = 'B003' AND rent > 400;
  ```

<table>
<thead>
<tr>
<th>staffNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>SG37</td>
</tr>
<tr>
<td>SG14</td>
</tr>
</tbody>
</table>

Table 6.8 Data for view StaffPropRent.
View Materialization

- If insert row into PropertyForRent with rent \( \leq 400 \) then view would be unchanged.
- If insert row for property PG24 at branch B003 with staffNo = SG19 and rent = 550, then row would appear in materialized view.
- If insert row for property PG54 at branch B003 with staffNo = SG37 and rent = 450, then no new row would need to be added to materialized view.
- If delete property PG24, row should be deleted from materialized view.
- If delete property PG54, then row for PG37 should not be deleted (because of existing property PG21).
Transactions: Sometimes you want to group a set of queries, especially insert/update/delete queries, so that they either all take effect, or none of them take effect.

- A transaction allows you to group a set of queries based on COMMIT and ROLLBACK.
- Example: I might want my create table queries to be one transaction, and each set of table insertions to be transactions

Access Privileges: SQL allows the DBA to grant select-insert/update/delete privileges to users on a per relation basis