Types:

- Table attributes have types.
- When creating a table, you must define the type of each attribute.
  This is analogous to declaring a variable's type in a program.
- Built-in Types:
  - **CHAR(n):** This is a fixed-length string of n characters. It can be padded with blanks if necessary.
  - **VARCHAR(n):** This is a variable-length string of up to n characters.
  - **TEXT:** This is a variable-length string with an unlimited number of characters. While this is not in the SQL standard, PSQL and others support it.
  - **INT:** INTEGER
  - **FLOAT:** REAL
  - **BOOLEAN**
  - **DATE**
  - **TIME**
  - **TIMESTAMP:** This is date plus time.

  E.g.
  - Strings: 'ABC'
    Note: Strings must be surrounded with single quotes.
  - INT: 37
  - FLOAT: 1.49, 37.96e2
  - BOOLEAN: TRUE, FALSE
  - DATE: '2011-09-22'
  - TIME: '15:00:02', '15:00:02.5'
  - TIMESTAMP: 'Jan-12-2011 10:25'
- These are not the only built-in types. There are many more built-in types.

User-defined types:

- Defined in terms of a built-in type.
- You make it more specific by defining constraints and perhaps a default value.
- E.g.
  create domain Grade as int default null check (value>=0 and value <=100);

  The check happens every time a user tries to put in a value of type Grade. It checks that the number is between 0 and 100 inclusive.
- E.g.
  create domain Campus as varchar(4) default 'StG' check (value in ('StG','UTM','UTSC'));
- Constraints on a type are checked every time a value is assigned to an attribute of that type.
- You can use these to create a powerful type system.
- The default value for a type is used when no value has been specified.
- This is useful because you can run a query and insert the resulting tuples into a relation even if the query does not give values for all attributes.
- Table attributes can also have default values.
  The difference between attribute default and type default is that:
  - with **attribute default**, it is only for that one attribute in that one table.
  - with **type default**, it is for every attribute defined to be of that type.

Key Constraints:

- **Primary Key Constraints:**
- A **PRIMARY KEY constraint** uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values and cannot contain NULL values.

- A table can have at most one primary key and this primary key can consist of one or multiple columns.
- Declaring that a set of one or more attributes is the PRIMARY KEY for a relation means:
  - they form a key (they are unique).
  - their values will never be null (you don't need to separately declare that).
- Primary keys are a big hint to the DBMS. They optimize for searches by this set of attributes.
- Every table must have 0 or 1 primary key.
  A table can have no primary key, but in practise, every table should have one. This is because if you have duplicate rows in a table, it can make queries useless and it can take up space, unnecessarily.
  You cannot declare more than one primary key.
- There are 2 ways to declare a primary key:
  1. For a single-attribute key, can be part of the attribute definition.
     E.g.
     ```
     create table Test (
             ID integer primary key,
             name varchar(25)
     );
     ```
  2. Or they can be at the end of the table definition. This is the only way for multi-attribute keys. The brackets are required.
     E.g.
     ```
     create table Test (
             ID integer,
             name varchar(25),
             primary key (ID)
     );
     ```
- **Unique Constraints:**
- The **UNIQUE constraint** ensures that all values in a column are different.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.
- However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.
- Declaring that a set of one or more attributes is UNIQUE for a relation means:
  - They form a key (They are unique.)
  - Their values can be null. This is because null ≠ null.
    Note: If they mustn't be null, you need to separately declare that.
- **Note:** You can declare more than one set of attributes to be UNIQUE.
- There are 2 ways to declare uniqueness:
  1. If only one attribute is involved, can be part of the attribute definition.
     E.g.
     ```
     create table Test (
             ID integer unique,
             name varchar(25)
     );
     ```

2. Or they can be at the end of the table definition. This is the only way if multiple attributes are involved. The brackets are required.

```
create table Test (
        ID integer,
        name varchar(25),
        unique (ID)
);
```

- For uniqueness constraints, no two nulls are considered equal.
- E.g. Consider

```
create table Testunique (
        first varchar(25),
        last varchar(25),
        unique(first, last)
);
```

This would prevent two insertions of ('Diane', 'Horton'), but it would allow two insertions of (null, 'Schoeler').
This can't occur with a primary key because primary keys can't be null.

- **Foreign Key Constraints:**
- A **FOREIGN KEY** is a key used to link two tables together.
- A FOREIGN KEY is a column or collection of columns in one table that refers to the PRIMARY KEY or unique columns in another table.
- The table containing the foreign key is called the **child table**, and the table containing the primary key is called the **referenced/parent table**.
- The **FOREIGN KEY constraint** is used to prevent actions that would destroy links between tables.
- The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.
- E.g. Consider **foreign key (sID) references Student**
  This means that the attribute sID is a foreign key that references the primary key of table Student.
  Every value for sID in this table must actually occur in the Student table.
- The requirement for foreign keys is that they must be declared either primary key or unique in the "home" table.
  I.e. The attribute(s) the foreign key references to must be either a primary key or unique.
- There are 2 ways to declare foreign keys:
  Suppose we have the table people as defined here:

```
create table People (
        SIN integer primary key,
        name text,
        OHIP text unique
);
```

1. If only one attribute is involved, can be part of the attribute definition.
   E.g.

```
create table Volunteers (
        email text primary key,
        OHIPnum text references People(OHIP)
);
```

2. Or they can be at the end of the table definition. This is the only way if multiple attributes are involved. The brackets are required.

```
create table Volunteers (
        email text primary key,
        OHIPnum text,
        FOREIGN KEY (OHIPnum) references People(OHIP)
);
```

- Suppose there is a foreign-key constraint from relation R to relation S.
  When must the DBMS ensure that:
  - The referenced attributes are PRIMARY KEY or UNIQUE?
  - The values actually exist?

  Also, what could cause a violation?
  You get to define what the DBMS should do. This is called specifying a **reaction policy**.
- **Check Constraints:**
- The **CHECK constraint** is used to limit the value range that can be placed in a column, in a tuple, in a relation or in a user-defined type.
- If you define a CHECK constraint on a single column it allows only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.
- The CHECK constraint can also be used for a user-defined type, as stated before.
  - **Attribute-based check constraints:**
  - Defined with a single attribute and constrains its value in every tuple.
  - Can only refer to that attribute.
  - Can include a subquery.
  - E.g.

```
CREATE TABLE Persons (
        ID int,
        LastName varchar(255),
        FirstName varchar(255),
        Age int CHECK (Age>=18)
);
```

  - E.g.

```
CREATE TABLE Student (
        ID int,
        LastName varchar(255),
        FirstName varchar(255),
        Program varchar(5) CHECK (program in (select post from P))
);
```

    Note: The condition can be anything that could go in a WHERE clause.
  - The condition is checked only when a tuple is inserted into that relation, or its value for that attribute is updated.
  - If a change somewhere else violates the constraint, the DBMS will not notice.
    E.g.
    If a student's program changes to something not in table P, we get an error.
    But if table P drops a program that some student has, there is no error.
    - **Not Null Constraints:**
    - By default, a column can hold NULL values.
    - The NOT NULL constraint enforces a column to NOT accept NULL values.

  - This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.
  - You can declare that an attribute of a table is NOT NULL.
  - E.g.

```
create table Course(
        cNum integer,
        name varchar(40) not null,
        dept Department,
        wr boolean,
        primary key (cNum, dept)
);
```

  - E.g.

```
create table User(
        username varchar(10) not null,
        password varchar(10) not null,
        first_name text not null,
        last_name text not null,
);
```

  - In practise, many attributes should be not null.
  - This is a very specific kind of attribute-based constraint.

- **Tuple-based check constraints:**
- Defined as a separate element of the table schema, so it can refer to any attributes of the table.
- Again, the condition can be anything that could go in a WHERE clause, and can include a subquery.
- E.g.

```
create table Student (
        sID integer,
        age integer,
        year integer,
        college varchar(4) check college in (select name from Colleges),
        check (year = age - 18),
);
```

  Here, both age and years are columns. However, with “check (year = age - 18)”, we cannot, for example, put age = 30 and year = 40 as that would violate the constraint.

- The constraint(s) are checked only when a tuple is inserted into that relation, or updated.
- Again, if a change somewhere else violates the constraint, the DBMS will not notice.
- **How nulls affect check constraints:**
- A check constraint only fails if it evaluates to false.
- It is not picky like a WHERE condition.

- E.g. Suppose we have check (age > 0)

age	Value of condition	CHECK outcome	WHERE outcome
19	TRUE	pass	pass
-5	FALSE	fail	fail
NULL	unknown	pass	fail

- Suppose you created this table:

```
create table Frequencies(
        word varchar(10),
        num integer check (num > 5)
);
```

It would allow you to insert ('hello', null) since null passes the constraint check (num > 5).

If you need to prevent that, use a "not null" constraint.
I.e.

```
create table Frequencies(
        word varchar(10),
        num integer not null check (num > 5)
);
```

- **Naming your constraints:**
- If you name your constraint, you will get more helpful error messages.
- This can be done with any of the types of constraint we've seen.
- To add a name to a constraint, do:
  Add constraint «name» before the check («condition»)
- E.g.

```
create domain Grade as smallint
        default null
        constraint gradeInRange
                check (value>=0 and value <=100));
```

- E.g.

```
create domain Campus as varchar(4)
        not null
        constraint validCampus
                check (value in ('StG', 'UTM', 'UTSC'));
```

- E.g.

```
create table Offering(...
        constraint validCourseReference
        foreign key (cNum, dept) references Course);
```

- The order of constraints doesn't matter, and doesn't dictate the order in which they're checked.
- **Assertions:**
- Check constraints can't express complex constraints across tables.
  Check constraints are good for checking data types or information within 1 table.
- Assertions are schema elements at the top level, so they can express cross-table constraints.
- Syntax: **create assertion (<name>) check (<predicate>);**

- E.g. Suppose we have:
    - Every loan has at least one customer, who has an account with at least $1,000.
    - For each branch, the sum of all loan amounts < the sum of all account balances.

  Both of these require multiple tables. Hence, we need to use assertions instead of checks.
- Assertions are powerful but costly.
- SQL has a fairly powerful syntax for expressing the predicates, including quantification.
- Assertions are costly because:
    1. They have to be checked upon every database update, although a DBMS may be able to limit this.
    2. Each check can be expensive.
- Testing and maintenance are also difficult.
- So assertions must be used with great care.
- **Triggers:**
- Assertions are powerful, but costly.
- Check constraints are less costly, but less powerful.
- Triggers are a compromise between these extremes:
    1. They are cross-table constraints, as powerful as assertions.
    2. But you control the cost by having control over when they are applied.
- A trigger is a database object that is associated with the table, it will be activated when a defined action is executed for the table.The trigger can be executed when we run the following statements:
    1. INSERT
    2. UPDATE
    3. DELETE

  And it can be invoked before or after the event.
  Syntax:

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

  E.g.

```
create trigger t1  before  UPDATE on sailors
for each row
begin
  if new.age>60 then
    set new.age=old.age;
  else
    set new.age=new.age;
  end if;
end;
```

- Differences between assertions and triggers:

ASSERTIONS	TRIGGERS
We can use assertions when we know that the given particular condition is always true.	We can use triggers even if a particular condition may or may not be true.
When the SQL condition is not met then there are chances for an entire table or even Database to get locked up.	Triggers can catch errors if the condition of the query is not true.
Assertions are not linked to specific tables or events. It performs tasks specified or defined by the user.	Triggers help in maintaining the integrity constraints in the database tables, especially when the primary key and foreign key constraint are not defined.
Assertions do not maintain any track of changes made in table.	Triggers maintain track of all changes occurring in the table.
Assertions have small syntax compared to triggers.	They have large syntax to indicate each and every specific of the created trigger.

- **Reaction Policies:**
- Suppose we have 2 relations, R and S, where R = Took and S = Student.
  Can you delete a student from S without making a change to R first? (Answer: No)
  Consider if you could. Say student R1 took CSCA08 and CSCA67. If you delete R1 from S without deleting its corresponding values in R, you'll have CSCA08 and CSCA67 in R but you can't find the student who's taking it.
- Your reaction policy can specify one of these reactions:
  1. **Cascade:**
  - Cascade propagates the change to the referring table.
  - **DELETE CASCADE:** When we create a foreign key using this option, it deletes the referencing rows in the child table when the referenced row is deleted in the parent table which has a primary key.
    Example:

```
create table Took (
        …
        foreign key (sID) references Student on delete cascade
        …
);
```

  - **UPDATE CASCADE:** When we create a foreign key using UPDATE CASCADE the referencing rows are updated in the child table when the referenced row is updated in the parent table which has a primary key.
    Example:

```
create table Took (
        …
        foreign key (sID) references Student on update cascade
        …
);
```

- Note the asymmetry.
  Suppose table R refers to table S.
  You can define fixes that propagate changes backwards from S to R.
  You define them in table R because it is the table that will be affected.
  However, you cannot define fixes that propagate forward from R to S.
- Add your reaction policy where you specify the foreign key constraint, as shown above.

2. **Set Null:**

- Sets the referring attribute(s) to null.
  I.e. Set the corresponding value in the referring tuple to null.

3. **Restrict:**

- Don't allow the deletion/update.

Note: There are more methods.
Note: If you say nothing, the default is to forbid the change in the parent table.

- Your reaction policy can specify what to do on:

1. **On delete:**

- This is when a deletion creates a dangling reference.
- **On Delete Cascade:** When data is removed from a parent table, the foreign key associated cell will be deleted in the child table.
- **On Delete Set Null:** When data is removed from a parent table, the foreign key associated cell will be null in the child table.
- **On Delete Restrict:** When data is removed from a parent table, and there is a foreign key associated with the child table, it gives an error and you can not delete the record.

2. **On update:**

- **On Update Cascade:** If the parent primary key is changed, the child value will also change to reflect that.
- **On Update Set Null:** The SQL Server sets the rows in the child table to NULL when the corresponding row in the parent table is updated. Note that the foreign key columns must be nullable for this action to execute.
- **On Update Restrict:** When data is updated from a parent table, and there is a foreign key associated with the child table, it gives an error and you can not update the record.

3. **Both:**

- Just put them one after the other.
- E.g. on delete restrict on update cascade

- **Semantics of Deletion:**
- Consider the following cases:
  1. What if deleting a tuple violates a foreign key constraint?
  2. What if deleting one tuple violates a foreign key constraint, but deleting others does not?
  3. What if deleting one tuple affects the outcome for a tuple encountered later?

  To prevent such interactions, deletion proceeds in two stages:
  1. Mark all tuples for which the WHERE condition is satisfied.
  2. Go back and delete the marked tuples.
- **Note:** If you drop a table that is referenced by another table, you must specify cascade.