A **DDL** allows the specification of:

- The **schema** for each relation
- The **domain** of values associated with each attribute
- **Integrity constraints (ICs)**
- The **physical storage structure** of each relation on disk
- The **set of indices** to be maintained for each relation
- **Security and authorization** information for each relation
• `char(n)`: Fixed length character string, with user-specified length n
• `varchar(n)`: Variable length character string, with user-specified maximum length n
• `int`: integer (a finite subset of the integers that is machine-dependent)
• `smallint`: Small integer (a machine-dependent subset of the integer domain type)
• `numeric(p,d)`: Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point.
• **real, double**: Floating point and double-precision floating point numbers, with machine-dependent precision

• **float(n)**: Floating point number, with user-specified precision of at least $n$ digits

• **date**: Dates, containing a (4 digits) year, month and date

• **time**: Time of day, in hours, minutes and seconds.

---

• **Notes**:  
  - null values are allowed in all the domain types. Declaring an attribute to be **not null** prohibits null values for that attribute.
  - **create domain** in SQL-92 creates user-defined domain types, e.g.,

```sql
create domain person-name char(20) not null
```
• Below we create a **Student** relation. The domain of each field is specified, and enforced by the DBMS whenever tuples are added or modified:

```sql
create table Student ( sid char(20),
        name char(20),
        login char(10),
        age int,
        gpa real )
```

• As another example, the **Enrolled** table holds information about courses that students take:

```sql
create table Enrolled ( sid char(20),
        cid char(20),
        grade char(2) )
```
Integrity Constraints (IC)

ICs guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.

- Integrity constraints are based upon the semantics of the real-world enterprise that is being described in the database relations.
- 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 an instance.
  - An IC is a statement about all possible instances!
  - Example:
    - Let an IC be “sid is a primary key” in the Student relation (previous slide).
    - Let another IC be “name is a primary key”.
    - A Student instance that contains duplicate sid values definitely violates the first IC.
    - However, if a Student instance contains unique name values, it does not mean that the second IC is true as well.
• They define valid values for attributes
• They constitute the most elementary form of integrity constraints
• They test values inserted in the database, and test queries to ensure that the comparisons make sense
The check clause in SQL-92 permits domains to be restricted.

Example: Use the 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 hourly-wage is declared to be a decimal number with 5 digits, 2 of which are after the decimal point.
- The domain has a constraint that ensures that the hourly-wage is greater than 4.00.
- “constraint value-test” is optional; useful to indicate which constraint an update violated.
There are possibly many candidate keys, which are specified using unique.

One of the candidate keys is chosen as the primary key.

```sql
create table Enrolled
  ( sid  char(20),
    cid  char(20),
    grade char(2),
    primary key (sid, cid) );
```

```sql
create table Enrolled
  ( sid  char(20),
    cid  char(20),
    grade char(2),
    primary key (sid),
    unique (cid, grade) );
```

Used carelessly, an IC can prevent the storage of database instances that arise in practice!
A **foreign key** is a set of fields in one relation that is used to `refer' to a tuple in another relation; it **must** correspond to a **candidate key** of the second relation. It is like a `logical pointer'.

**Example:** Enrolled \((sid, cid, grade)\)

- \(sid\) is a **foreign key** in Enrolled referencing Student

**Referential integrity** is achieved when a value that appears in the Enrolled instance for \(sid\) appears also in the Student instance for \(sid\), i.e., there are **no dangling references** in Enrolled

**A data model without referential integrity:** Links in HTML
• **Example:** Only students listed in the **Student** relation should be allowed to enroll for courses:

```
create table Enrolled (
    sid   char(20),
    cid   char(20),
    grade char(2),
    primary key (sid,cid),
    foreign key (sid) references Student )
```

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>
• Consider Student and Enrolled; sid in Enrolled is a foreign key that references Student.

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

• What should be done if a Student tuple is deleted?
  - Disallow deletion of a Student tuple that is referred to by an Enrolled tuple
  - Also delete all Enrolled tuples that refer to it (cascading deletion)
  - Set sid in Enrolled tuples that refer to it to a default sid
  - Set sid in Enrolled tuples that refer to it to a special value null (not applicable in this example because sid is part of the primary key)

• What should be done if the primary key of a Student tuple is updated?
  - Update also the enrollments of that student
• SQL-92 supports all 4 options on delete and update.
  - no action (default): deletion is rejected
  - cascade: also delete all tuples that refer to deleted tuple
  - set null / set default: sets the foreign key value of the referencing tuple

```
create table Enrolled
  ( sid  char(20),
    cid  char(20),
    grade char(2),
    primary key (sid,cid),
    foreign key (sid)
    references Student
    on delete cascade
    on update cascade
)
```
Example: Does every department have a manager?
- If so, this is a participation constraint
- The participation of Department in Manages below is total (vs. partial)
  - Every HKID value in the Department table must have a non-null HKID value!
• We can capture total participation constraints using `not null`

• **Example:**

```sql
create table Department (
    did int,
    dname char(20),
    budget real,
    HKID char(11) not null,
    since date,
    primary key (did),
    foreign key (HKID) references Employee
    on delete no action)
```
• A weak entity can be identified uniquely by considering its partial key and the primary key of another identifying (or owner) entity
  - The owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities)
  - The weak entity set must have a total participation in this identifying relationship set
• The weak entity set and the identifying relationship set are translated into a single table
  - When the owner entity is deleted, all owned weak entities must also be deleted

```sql
create table Dep_Policy (  
  pname  char(20),  
  age    int,  
  cost   real,  
  HKID   char(11) not null,  
  primary key (pname, HKID),  
  foreign key (HKID) references Employee  
  on delete cascade)
```
**Destroying and Altering Relations**

- We use clause **drop table** to **destroy** a relation. The schema information and the tuples are deleted.
  - Example:
    ```
    drop table Student
    ```

- We use clause **alter table** to **add** or **delete** a column in a relation. If a column is added, every tuple in the current instance is extended with a *null* value in the new field.
  - Examples:
    ```
    alter table Student add firstYear int
    ```
    ```
    alter table Student drop age
    ```
Record Deletion

• **Example:** Delete all account records at the Perryridge branch

```sql
delete from Account
where branch-name = "Perryridge"
```

• **Conceptually,** delete is done in two steps:
  - **find** the tuples you want to delete:
    ```sql
    select *
    from Account
    where branch-name = "Perryridge"
    ```
  - **delete** the tuples you found.
Complex Deletion

• Example: Delete all accounts at every branch located in Needham. *Must also delete depositors of these accounts*

```sql
delete from Depositor
where account-number in
  ( select account-number from Branch, Account
    where branch-city = "Needham"
    and Branch.branch-name = Account.branch-name )
```

```sql
delete from Account
where branch-name in
  ( select branch-name from Branch
    where branch-city = "Needham" )
```

- The first delete removes Depositor records for accounts in Needham
- Such deletions of depositors can happen automatically, if we use “on delete cascade” for the foreign key account-number in Depositor

Branch (branch-name, branch-city, assets)
Depositor (customer-name, account-number)
Account (account-number, balance, branch-name)
Record Insertion

- **Example:** Add a new tuple to **Account**

  ```sql
  insert into Account values ("A-9732", 1200, "Perryridge")
  ```

  To reorder attributes, specify attribute names explicitly:

  ```sql
  insert into Account (branch-name, balance, account-number)
  values ("Perryridge", 1200, "A-9732")
  ```

- **Example:** Add a new tuple to **Account** with balance set to `null`

  ```sql
  insert into Account values ("A-777", null, "Perryridge")
  ```
Complex Insertion

- **Example**: Create a $200 savings account for all loan customers of the Perryridge branch. Let the loan number serve as the account number for the new savings account.

```sql
insert into Account
    select loan-number, 200, branch-name
    from Loan
    where branch-name = "Perryridge"
```

```sql
insert into Depositor
    select customer-name, loan-number
    from Loan, Borrower
    where branch-name = "Perryridge"
        and Loan.account-number = Borrower.account-number
```

Depositor (customer-name, account-number)
Account (account-number, balance, branch-name)
Loan (loan-number, amount, branch-name)
Borrower (customer-name, loan-number)
**Example:** Increase all accounts with balance over $10,000 by 6%; all other accounts receive 5%.

- Write two update statements:
  
  ```sql
  update Account
  set balance = balance * 1.06
  where balance > 10000
  
  update Account
  set balance = balance * 1.05
  where balance <= 10000
  ```

- the **order** is important! (Why?)

- can be done better using the **case** statement (see next slide)
Conditional Updates

- **Example**: Same query as before. Increase all accounts with balances over $10,000 by 6%; all other accounts receive 5%.

update Account
set balance = case
    when balance <= 10000 then balance * 1.05
    else balance * 1.06
end

Account (account-number, balance, branch-name)
• The **views** provide a mechanism to hide certain data from the view of certain users.

• **Syntax:**

  ```sql
  create view view-name as <query expression>
  where <query expression> is any legal SQL query
  ```

• **Example (creation):** Create a view from `Loan(loan-number, amount, branch-name)` that hides the amount.

  ```sql
  create view Loan-view as
  select branch-name, loan-number
  from Loan
  ```

• **Example (query):** Find all loans in the Perryridge branch.

  ```sql
  select loan-number
  from Loan-view
  where branch-name = "Perryridge"
  ```

  - A user who has access to `Loan-view`, but not `Loan`, cannot see the amount.
• Assume that we allow users who have access to Loan-view, to insert records in the view.
• Add a new tuple to Loan-view:

\[
\text{insert into Loan-view values ("Perryridge", "L-307")}
\]

• This insertion must be represented by the insertion of the tuple ( "L-307", null, "Perryridge") into the Loan relation!
Consider the following view:

```sql
create view Branch-Borrower as
    select branch-name, customer-name
    from Loan, Borrower
    where Loan.loan-number = Borrower.loan-number
```

Assume that we want to insert ("Choi Hung", "Lei Chen") into Branch-Borrower. The Account and Depositor tables have to be updated accordingly:

```sql
insert into Loan values (null, null, "Choi Hung")
insert into Borrower values ("Lei Chen", null)
```

These updates cannot be performed because the key values are null. Even if they were allowed, they would not have the desired effect since Branch-Borrower still does not include ("Choi Hung", "Lei Chen") - the new tuples cannot be joined on the loan number because it is null.
• **Rules for legal view updates:**

  - A view built on a *single* defining table is updatable, if the view contains the *primary key* of the defining table
  - Views defined on *multiple* tables are in general *not updatable*
  - Views involving *aggregate* functions on the defining table are *not updatable*
General Constraints

• Useful when more general ICs than keys are involved.
• Created in the table definition
• Checked whenever there is an update within the table

```sql
create table Loan
    ( loan-number int,
      amount int,
      branch-name char(20),
      primary key (loan-number),
      foreign key (branch-name)
          references Branch
          on delete cascade,
      check ( amount >= 1 and amount <= 10000),
      check ( branch-name <> "Choi Hung") )
```
• An **assertion** is a complex constraint that the database must always satisfy

• **Syntax:** An assertion in SQL-92 takes the form

  ```sql
  create assertion <assertion-name> check <predicate>
  ```

• **Difference from general constraints:**
  - A constraint is associated with a single table and checked when there is an update on this specific table
  - An assertion may be associated with several tables, and is checked every time there is an update anywhere

• **Assertion testing may introduce a significant amount of overhead;** hence assertions should be used with great care

• Any predicate allowed in SQL can be used
• **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(balance) from Account
        where Account.branch-name = Branch.branch-name)
  ) )
```

- Note that the assertion refers to multiple tables. Therefore, it cannot be included as a constraint in the definition of Loan or Amount.
A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database.

To design a trigger mechanism, we must:
- Specify the conditions under which the trigger is to be executed
- Specify the actions to be taken when the trigger executes

The SQL-92 standard does not include triggers, but many implementations support triggers.
Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
- setting the account balance to zero
- creating a loan in the amount of the overdraft
- giving this loan a loan number which is identical to the account number of the overdrawn account

The condition for executing the trigger is an update to the account relation that results in a negative balance value.
create trigger overdraft after update on Account T
( if new T.balance < 0
then ( insert into Loan
values (T.account-number, - new T.balance, T.branch-name)
insert into Borrower
( select customer-name, account-number
from Depositor
where T.account-number = Depositor.account-number)
update Account S
set S.balance = 0
where S.account-number = T.account-number ) )

• The keyword new used before “T.balance” indicates that the value of “T.balance” after the update should be used; if it is omitted, the value before the update is used.