
# CSCC43H: Introduction to Databases

# Lecture 5

*Wael Aboulsaadat*

Acknowledgment: these slides are partially based on Prof. Garcia-Molina & Prof. Ullman slides accompanying the course's textbook.

# Database Management System (DBMS)

- A collection of programs that enable:
  - → Defining (describing the structure),
  - - Populating by data (Constructing),
  - – Manipulating (querying, updating),
  - – Preserving consistency,
  - – Protecting from misuse,
  - – Recovering from failure, and
  - – Concurrent using

  of a database.

# Create Table Construct

- An SQL relation is defined using the **create table** command:

  **create table** *r* ($A_1$ $D_1$, $A_2$ $D_2$, ..., $A_n$ $D_n$,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - r is the name of the relation
  - each $A_i$ is an attribute name in the schema of relation r
  - $D_i$ is the data type of values in the domain of attribute $A_i$

# Example of `create table`

```
create table Employee
  (
     RegNo     character(6),
     FirstName character(20),
     Surname   character(20),
     Salary    numeric(9),
     City      character(15)
  )
```

# Intra-Relational Constraints

- Constraints are conditions that must be verified by every database instance
- Intra-relational constraints involve a single relation
  - **`not null`** (on single attributes)
  - **`unique`**: permits the definition of keys; syntax:
    - for single attributes: **`unique`**, after the domain
    - for multiple: **`unique`** **(** ***Attribute*** **{,** ***Attribute*** **}** **)**
  - **`primary key`**: defines the primary key (once for each table; implies **not null**); syntax like **`unique`**
  - `check`: described later

# Example of Intra-Relational Constraints

- Each pair of **FirstName** and **Surname** uniquely identifies each element

```
FirstName char(20) not null,
Surname char(20) not null,
unique(FirstName,Surname)
```

- Note the difference with the following (stricter) definition:

```
FirstName char(20) not null unique,
Surname char(20) not null unique,
...
```

# Inter-Relational Constraints

Constraints may involve several relations:

- **`check`**: checks whether an assertion is true;
- **`references`** and **`foreign key`** permit the definition of referential integrity constraints;
  - Syntax for single attributes
    **`references`** after the domain
  - Syntax for multiple attributes
    **`foreign key`** **( *Attribute* {, *Attribute* } )**
    **`references`** **...**
- It is possible to associate reaction policies to violations of referential integrity constraints.

# Example

```
create table Employee
(
    RegNo char(6),
    FirstName char(20) not null,
    Surname char(20) not null,
    Dept char(15),
    Salary numeric(9) default 0,
    City char(15),
    primary key(RegNo),
    foreign key(Dept) references Department(DeptName),
    unique(FirstName,Surname)
)
```

# Database Management System (DBMS)

- A collection of programs that enable:

  Defining (describing the structure),

  → Populating by data (Constructing),

  – Manipulating (querying, updating),
  – Preserving consistency,
  – Protecting from misuse,
  – Recovering from failure, and
  – Concurrent using

  of a database.

# Modification of the Database – Insertion

- Add a new tuple to *account*

  **insert into** *account*
  **values** (‘A-9732’, ‘Perryridge’,1200)

- *or equivalently*
  **insert into** *account (branch-name, balance, account-number)*
  **values** (‘Perryridge’, 1200, ‘A-9732’)

- Add a new tuple to *account* with *balance* set to null

  **insert into** *account*
  **values** (‘A-777’,‘Perryridge’, *null*)

# Banking Example

*branch (branch_name, branch_city, assets)*

*customer (customer_name, customer_street, customer_city)*

*account (account_number, branch_name, balance)*

*loan (loan_number, branch_name, amount)*

*depositor (customer_name, account_number)*

*borrower (customer_name, loan_number)*

# Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

  Write two **update** statements:

```
update account
set balance = balance * 1.06
where balance > 10000
```

```
update account
set balance = balance * 1.05
where balance ≤ 10000
```

# Database Management System (DBMS)

- A collection of programs that enable:
  - → Defining (describing the structure),
  - Populating by data (Constructing),
  - Manipulating (querying, updating),
  - Preserving consistency,
  - Protecting from misuse,
  - Recovering from failure, and
  - Concurrent using

  of a database.

# Reaction Policies

**Violations** arise from

- (a) **updates** on referred attribute or
- (b) row **deletions**.

**Reactions** operate on internal table, after changes to an external table.

- Reactions are:
  - **`cascade`**: propagate the change;
  - **`set null`**: nullify the referring attribute;
  - **`set default`**: assign default value to the referring attribute;
  - **`no action`**: forbid the change on external table.
- Reactions may depend on the event; syntax:

```
on < delete | update >
   < cascade | set null | set default | no action >
```

# Note

- “Correct” policy is a design decision
- E.g., what does it mean if a creditcard goes away? What if a creditcard account changes its number?

# Example

```
create table Employee
(
    RegNo char(6),
    FirstName char(20) not null,
    Surname char(20) not null,
    Dept char(15),
    Salary numeric(9) default 0,
    City char(15),
    primary key(RegNo),
    foreign key(Dept)
        references Department(DeptName)
        on delete set null
        on update cascade,
    unique(FirstName,Surname)
)
```

# Attribute-based Checks

**CHECK (condition)**

- Follow an attribute by a condition that must hold for that attribute in each tuple of its relation
- Condition may involve the checked attribute
- Other attributes and relations may be involved, but *only* in subqueries
- (Different DBMS vendors may or may not support this)
- Condition is checked only when the associated attribute changes (I.e., an INSERT or UPDATE occurs)

# Example

```
CREATE TABLE Purchase (
    item CHAR(15),
    card CHAR(20),
    price REAL CHECK (
        price > 0.00 AND price <= 1000.00
    )
);
```

# Tuple-based Checks

## CHECK (condition)

- Separate element in a table declaration
- The condition can refer to any attribute of the relation
- Checked whenever a tuple is inserted or updated

# Example

- Only “SONY laptop” can be charged more than $3000.00

```
CREATE TABLE Purchase (
    item CHAR(15),
    card CHAR(20),
    price REAL,
    CHECK (item = 'SONY laptop' OR
          price <= 3000.00)
);
```

# Drop and Alter Table

- The **drop table** command deletes all information about the dropped relation from the database.
- The **alter table** command is used to add attributes to an existing relation.

  **alter table** *r* **add** *A D*

  where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
  - All tuples in the relation are assigned null as the value for the new attribute.
- The **alter table** command can also be used to drop attributes of a relation

  **alter table** *r* **drop** *A*

  where *A* is the name of an attribute of relation *r*
  - Dropping of attributes not supported by many databases

# Drop and Alter Table – cont'd

- Examples:
  - `alter table Department add column NoOfOffices numeric(4)`
  - `drop table TempTable cascade`

# Create Table Index

- Indices are created in an existing table to locate rows more quickly and efficiently.
- It is possible to create an index on one or more columns of a table, and each index is given a name.

**CREATE UNIQUE INDEX index_name**
**ON table_name (column_name)**

- Example:

**CREATE INDEX PersonIndex ON**
**Person (LastName)**

**Note: Updating a table containing indexes takes more time than updating a table without, this is because the indexes also need an update. So, it is a good idea to create indexes only on columns that are often used for a search.**

# Defining Domains

- Possible attribute values can be specified
  - Using a **CHECK** constraint or
  - Creating a new domain
- Domain can be used in several declarations
- Domain is a schema element

```
CREATE DOMAIN Grades CHAR (1)
   CHECK  (VALUE IN ('A', 'B', 'C', 'D', 'F'))

CREATE TABLE Transcript (
                                  …. Grade: GRADES,  …..)
```

# Database Management System (DBMS)

- A collection of programs that enable:
  Defining (describing the structure),
  Populating by data (Constructing),
  → Manipulating (querying, updating),
  – Preserving consistency,
  – Protecting from misuse,
  – Recovering from failure, and
  – Concurrent using
  of a database.

# SQL Query

- The generic query:

  **select** $T_1.Attr_{11}$, …, $T_h.Attr_{hm}$
  **from** $Table_1$ $T_1$, …, $Table_n$ $T_n$
  **where** *Condition*

# Algebraic Interpretation of SQL Queries

- The generic query:

  **select** $T_1.Attr_{11}$, …, $T_h.Attr_{hm}$
  **from** $Table_1$ $T_1$, …, $Table_n$ $T_n$
  **where** *Condition*

corresponds to the relational algebra query:

$$\pi_{T_1.Attr_{11},\ldots,T_h.Attr_{hm}}(\sigma_{Condition}(Table_1 \times \ldots \times Table_n))$$

# Extended Relational Algebra Operations

- Generalized Projection
  - Extends the projection operation by allowing arithmetic expression over attributes and constants to be used in the projection list
- Aggregate Functions
- Join Extensions

# Aggregate Functions and Operations

- **Aggregation function** takes a collection of values and returns a single value as a result.

  **avg**: average value
  **min**: minimum value
  **max**: maximum value
  **sum**: sum of values
  **count**: number of values

- **Aggregate operation** in relational algebra

$$_{G1, G2, \ldots, Gn}\mathcal{g}_{F1(A1), F2(A2), \ldots, Fn(An)}(E)$$

  - $E$ is any relational-algebra expression
  - $G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)
  - Each $F_i$ is an aggregate function
  - Each $A_i$ is an attribute name

- Result of aggregation does not have a name
  - Can use rename operation to give it a name
  - For convenience, we permit renaming as part of aggregate operation

# Aggregate Operation Example

account

| branch-name | account-number | balance |
|---|---|---|
| Perryridge | A-102 | 400 |
| Perryridge | A-201 | 900 |
| Brighton | A-217 | 750 |
| Brighton | A-215 | 750 |
| Redwood | A-222 | 700 |

| branch-name | balance |
|---|---|
| Perryridge | 1300 |
| Brighton | 1500 |
| Redwood | 700 |

$$branchName\ \mathcal{g}\ sum(balance)\ (account)$$

# Outer Join

- An extension of the join operation that avoids loss of information.
- First, computes the natural join and then
- adds tuples form one of the operand relations that do not match tuples in the other operand relation to the result of the above join
- Uses *null* values:
  - *null* signifies that the value is unknown or does not exist
  - All comparisons involving *null* are (roughly speaking) **false** by definition.

# Outer Join – Example

- Relation *loan*

| loan-number | branch-name | amount |
|---|---|---|
| L-170 | Downtown | 3000 |
| L-230 | Redwood | 4000 |
| L-260 | Perryridge | 1700 |

- Relation borrower

| customer-name | loan-number |
|---|---|
| Jones | L-170 |
| Smith | L-230 |
| Hayes | L-155 |

# Outer Join – Example

- **Inner Join:** *loan* ⋈ *Borrower*

| loan-number | branch-name | amount | customer-name |
|---|---|---|---|
| L-170 | Downtown | 3000 | Jones |
| L-230 | Redwood | 4000 | Smith |

- Left Outer Join: loan ⟕ Borrower

| loan-number | branch-name | amount | customer-name |
|---|---|---|---|
| L-170 | Downtown | 3000 | Jones |
| L-230 | Redwood | 4000 | Smith |
| L-260 | Perryridge | 1700 | null |

# Outer Join – Example

- **Right Outer Join :** *loan* ⟖ *borrower*

| loan-number | branch-name | amount | customer-name |
|---|---|---|---|
| L-170 | Downtown | 3000 | Jones |
| L-230 | Redwood | 4000 | Smith |
| L-155 | null | null | Hayes |

- **Full Outer Join**

  loan ⟗ borrower

| loan-number | branch-name | amount | customer-name |
|---|---|---|---|
| L-170 | Downtown | 3000 | Jones |
| L-230 | Redwood | 4000 | Smith |
| L-260 | Perryridge | 1700 | null |
| L-155 | null | null | Hayes |

# Example Database

**EMPLOYEE**

| FirstName | Surname | Dept | Office | Salary | City |
|---|---|---|---|---|---|
| Mary | Brown | Administration | 10 | 45 | London |
| Charles | White | Production | 20 | 36 | Toulouse |
| Gus | Green | Administration | 20 | 40 | Oxford |
| Jackson | Neri | Distribution | 16 | 45 | Dover |
| Charles | Brown | Planning | 14 | 80 | London |
| Laurence | Chen | Planning | 7 | 73 | Worthing |
| Pauline | Bradshaw | Administration | 75 | 40 | Brighton |
| Alice | Jackson | Production | 20 | 46 | Toulouse |

**DEPARTMENT**

| DeptName | Address | City |
|---|---|---|
| Administration | Bond Street | London |
| Production | Rue Victor Hugo | Toulouse |
| Distribution | Pond Road | Brighton |
| Planning | Bond Street | London |
| Research | Sunset Street | San José |

# SQL Query

- The generic query:

  **select** $T_1.Attr_{11}$, …, $T_h.Attr_{hm}$
  **from** $Table_1$ $T_1$, …, $Table_n$ $T_n$
  **where** *Condition*

# * in the Target List

- "Find all the information relating to employees named Brown":

```
select *
from Employee
where Surname = 'Brown'
```

- Result:

| FirstName | Surname | Dept | Office | Salary | City |
|---|---|---|---|---|---|
| Mary | Brown | Administration | 10 | 45 | London |
| Charles | Brown | Planning | 14 | 80 | London |

# Predicate Conjunction

- "Find the first names and surnames of employees who work in office number 20 of the Administration department":

```
select FirstName, Surname
from Employee
where Office = '20' and
      Dept = 'Administration'
```

- Result:

| FirstName | Surname |
|---|---|
| Gus | Green |

# Predicate Disjunction

- "Find the first names and surnames of employees who work in either the Administration or the Production department":

```
select FirstName, Surname
from Employee
where Dept = 'Administration' or
      Dept = 'Production'
```

- Result:

| FirstName | Surname |
|---|---|
| Mary | Brown |
| Charles | White |
| Gus | Green |
| Pauline | Bradshaw |
| Alice | Jackson |

# Complex Logical Expressions

- "Find the first names of employees named Brown who work in the Administration department or the Production department":

```
select FirstName
from Employee
where Surname = 'Brown' and
      (Dept = 'Administration' or
       Dept = 'Production')
```

- Result:

| FirstName |
|---|
| Mary |

# Column Aliases

- "Find the salaries of employees named Brown":

```
select Salary as Remuneration
from Employee
where Surname = 'Brown'
```

- Result:

| Remuneration |
|---|
| 45 |
| 80 |

# Simple Join Query

- "Find the names of employees and their cities of work":

```
select Employee.FirstName,
   Employee.Surname, Department.City
from Employee, Department
where Employee.Dept = Department.DeptName
```

- Result:

| FirstName | Surname | City |
|---|---|---|
| Mary | Brown | London |
| Charles | White | Toulouse |
| Gus | Green | London |
| Jackson | Neri | Brighton |
| Charles | Brown | London |
| Laurence | Chen | London |
| Pauline | Bradshaw | London |
| Alice | Jackson | Toulouse |

# Table Aliases

- "Find the names of employees and the cities where they work" (using an alias):

```
select FirstName, Surname, D.City
from Employee, Department D
where Dept = DeptName
```

- Result:

| FirstName | Surname | City |
|---|---|---|
| Mary | Brown | London |
| Charles | White | Toulouse |
| Gus | Green | London |
| Jackson | Neri | Brighton |
| Charles | Brown | London |
| Laurence | Chen | London |
| Pauline | Bradshaw | London |
| Alice | Jackson | Toulouse |

# Table Variables

- Table aliases may be interpreted as table variables. These correspond to the renaming operator ρ.
- "Find all first names and surnames of employees who have the same surname and different first names with someone in the Administration department":

```
select E1.FirstName, E1.Surname
from Employee E1, Employee E2
where E1.Surname = E2.Surname and
      E1.FirstName <> E2.FirstName and
      E2.Dept = 'Administration'
```

- Result:

| FirstName | Surname |
|---|---|
| Charles | Brown |

# With Attribute Expressions

- Find the monthly salary of the employees named White:

```
select Salary / 12 as MonthlySalary
from Employee
where Surname = 'White'
```

- Result:

| MonthlySalary |
|---|
| 3.00 |

# Operator `like`

- "Find employees with surnames that have ‘r’ as the second letter and end in ‘n’":

```
select *
from Employee
where Surname like ‘_r%n’
```

*0 or more chars*

*exactly 1 char*

- Result:

| FirstName | Surname | Dept | Office | Salary | City |
|---|---|---|---|---|---|
| Mary | Brown | Administration | 10 | 45 | London |
| Gus | Green | Administration | 20 | 40 | Oxford |
| Charles | Brown | Planning | 14 | 80 | London |