# Ceng 302
# Database Systems

# SQL: Structured Query Language-3

**Prof. Dr. Adnan YAZICI**
Department of Computer Engineering,
METU

**(Fall 2021)**

# Queries With GROUP BY and HAVING

*recap*

| | |
|---|---|
| SELECT | [DISTINCT] *target-list* |
| FROM | *relation-list* |
| WHERE | *qualification* |
| GROUP BY | *grouping-list* |
| HAVING | *group-qualification* |

- The *target-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
  - The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group.
  - A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.

# Queries With GROUP BY and HAVING

recap

## Conceptual Evaluation:

- The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, `*unnecessary*' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* is then applied to eliminate some groups.
- In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*.

**Q:** *"Find the age of the youngest sailor with age ≥18, for each rating with at least 2 such sailors"*

*recap*

```
SELECT  S.rating,  MIN (S.age)
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

| sid | sname | rating | age |
|---|---|---|---|
| 22 | dustin | 7 | 45.0 |
| 31 | lubber | 8 | 55.5 |
| 71 | zorba | 10 | 16.0 |
| 64 | horatio | 7 | 35.0 |
| 29 | brutus | 1 | 33.0 |
| 58 | rusty | 10 | 35.0 |

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes `*unnecessary*'.
- 2nd column of result is unnamed.

  (Use AS to name it.)

| rating | age |
|---|---|
| 1 | 33.0 |
| 7 | 45.0 |
| 7 | 35.0 |
| 8 | 55.5 |
| 10 | 35.0 |

| rating | |
|---|---|
| 7 | 35.0 |

*Answer relation*

Q: Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors.

*recap*

| rating | age |
|---|---|
| 7 | 45.0 |
| 1 | 33.0 |
| 8 | 55.5 |
| 8 | 25.5 |
| 10 | 35.0 |
| 7 | 35.0 |
| 10 | 16.0 |
| 9 | 35.0 |
| 3 | 25.5 |
| 3 | 63.5 |
| 3 | 25.5 |

| rating | age |
|---|---|
| 1 | 33.0 |
| 3 | 25.5 |
| 3 | 63.5 |
| 3 | 25.5 |
| 7 | 45.0 |
| 7 | 35.0 |
| 8 | 55.5 |
| 8 | 25.5 |
| 9 | 35.0 |
| 10 | 35.0 |

| rating | minage |
|---|---|
| 3 | 25.5 |
| 7 | 35.0 |
| 8 | 25.5 |

# Interactive DML - built-in functions

*recap*

Q: "*Find the average ticket price by airline for scheduled flights out of Atlanta for airlines with more than 5 scheduled flights out of Atlanta from FLT-SCHEDULE*"

**SELECT** AIRLINE, **AVG**(PRICE)
**FROM** FLT-SCHEDULE
**WHERE** FROM-AIRPORTCODE = "ATL"
**GROUP BY** AIRLINE
**HAVING COUNT** (FLT#) >= 5;

# Interactive DML - insert, **delete, update**

*recap*

**INSERT INTO** FLT-SCHEDULE
**VALUES** ("DL212", "DELTA", 11-15-00, "ATL",
13-05-00, "CHI", 650, 00351.00);

**INSERT INTO** FLT-SCHEDULE(FLT#,AIRLINE)
**VALUES** ("DL212", "DELTA"); /*default nulls added*/

**Q: "***Insert into FLT-INSTANCE all flights scheduled for Thursday, 9/10/98*"

**INSERT INTO** FLT-INSTANCE(FLT#, DATE)
(**SELECT** S.FLT#, 1998-09-10
**FROM** FLT-SCHEDULE S, FLT-WEEKDAY D
**WHERE** S.FLT#=D.FLT#
**AND** D.WEEKDAY="TH");

# Interactive DML - **insert,** delete, **update**

*recap*

***Q:*** *"Cancel all flight instances for Delta on 9/10/98"*

**DELETE FROM** FLT-INSTANCE

**WHERE** DATE=1998-09-10

**AND** FLT# **IN**

(**SELECT** FLT#

**FROM** FLT-SCHEDULE

**WHERE** AIRLINE="DELTA");

# Interactive DML - **insert, delete,** update

recap

***Q:*** *"Update all reservations for customers on DL212 on 9/10/98 to reservations on AA121 on 9/10/98"*

**UPDATE** RESERVATION
**SET** FLT#="AA121"
**WHERE** DATE=1998-09-10
    **AND** FLT#="DL212";

# Embedded DML- impedance mismatch

- SQL is a powerful, set-oriented, declarative language
- SQL queries return sets of rows
- Host languages cannot handle large sets of structured data
- Cursors resolve the mismatch:

```
EXEC SQL BEGIN DECLARE SECTION;
   ..........
   ..........
EXEC SQL END DECLARE SECTION;
EXEC SQL
      DECLARE <key attribute-name> CURSOR
      FOR <SQL QUERY>
   EXEC SQL OPEN <ATTR>;                    /*query executed; cursor open;*/
   WHILE MORE <ATTR> DO                     /*first row is the current row */
     EXEC SQL FETCH <ATTR>                  /*one row of query placed in   */
      INTO : <Host variables>               /*host variables*/
       <DO YOUR THING WITH THE DATA>;
   END-WHILE;
   EXEC SQL CLOSE <ATTR>;
```

# Embedded DML- database access

- to access the result of the query, one row at a time, the following is used:

```
EXEC SQL BEGIN DECLARE SECTION;
DECLARE FROM-AIRPORTCODE CHAR(3);            /*input to query*/
..........
DECLARE FLT# CHAR(5);                        /*targets for FETCH*/
DECLARE AIRLINE VARCHAR(25);
DECLARE PRICE DECIMAL(7,2);
EXEC SQL END DECLARE SECTION;
EXEC SQL
DECLARE FLT CURSOR
FOR SELECT FLT#, AIRLINE, PRICE              /*SQL query;*/
        FROM FLT-SCHEDULE
        WHERE FROM-AIRPORTCODE = FROM-AIRPORTCODE;
EXEC SQL OPEN FLT;                           /*query executed; cursor open;*/
WHILE MORE FLTs DO                           /*first row is the current row */
                                             /*one row of query placed in   */
    INEXEC SQL FETCH TO :FLT#, :AIRLINE, :PRICE;      /*host variables*/
    <DO YOUR THING WITH THE DATA>;
END-WHILE;
EXEC SQL CLOSE FLT;                          /*cursor closed*/
```

# Views- definition, use, update

```
Create View Vname As
<Query>
```

## What is a view?

- A view is a virtual table.

## Why use views?

- Simplify Queries
- Hide some data from some users (security)
- Make some queries easier / more natural
- Modularity of database access
- Improve performance of database queries (only if materialized views used)

## Querying views

- Once *View is* defined, one can reference *View* like any table
- Queries involving *View* rewritten to use base tables

# Types of Views

- **Virtual views:**
  - Used in databases
  - Computed only on-demand – slower at runtime
  - Always up to date

- **Materialized views**
  - Used in data warehouses, OLAP,
  - Precomputed offline – faster at runtime
  - May have stale data

# Views- definition, use, update

- how a view is defined:

```
CREATE VIEW ATL-FLT
  AS SELECT FLT#, AIRLINE, PRICE
       FROM FLT-SCHEDULE
       WHERE FROM-AIRPORTCODE = “ATL”;
```

- how a query on a view is written:

```
SELECT *
FROM ATL-FLT
WHERE PRICE <= 00200.00;
```

- how this query on a view is computed:

```
SELECT FLT#, AIRLINE, PRICE
FROM FLT-SCHEDULE
WHERE FROM-AIRPORTCODE=“ATL” AND PRICE<00200.00;
```

- how a view definition is dropped:

```
DROP VIEW ATL-FLT [RESTRICT|CASCADE];
```

# Views- definition, use, update

**Materialized views:**

- View $V$ = ViewQuery($R_1$, $R_2$, …, $R_n$)
- Create table $V$ with schema of query result
- Execute ViewQuery and put results in $V$
- Queries refer to $V$ as if it's a table

**But…**

- $V$ could be very large
- Modifications to $R_1$, $R_2$, …, $R_n$ ⇒ recompute or modify $V$

- Some DBMS allow views to be stored (*materialised views*)
  - materialised views have to be updated when its relations change (*view maintenance*)

# Materialized Views

```
Create Materialized View CA-CS As
Select C.cName, S.sName
From College C, Student S, Apply A
Where C.cName = A.cName And S.sID = A.sID
And C.state = 'CA' And A.major = 'CS'
```

+ Can use **CA-CS** as if it's a table (it is!)
+ Modifications to base data invalidate view **CA-CS**?

College

| cName | state | enr |
|---|---|---|
| | | |
| | | |
| | | |
| | | |

Student

| sID | sName | GPA | HS |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

Apply

| sID | cName | major | dec |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

# Views- definition, use, update

## Modifying views

- Once *View* defined, can we modify *View* like any other table ?
- One way of thinking of this is that it doesn't make sense, when *View* is not stored
- The other way of thinking of this is that it has to make sense when *Views* are some users' entire "view" of the database

❖ Soln: Modifications to *View* rewritten to modify base tables

(1) Rewriting process specified explicitly by view creator

+ Can handle all modifications

– No guarantee of correctness (or meaningful)

(2) Restrict views + modifications so that translation to base table modifications is meaningful and unambiguous

+ No user intervention

– Restrictions are significant

# Views- definition, use, update

a view is **updatable if and only if**:

- it does not contain any of the keywords JOIN, UNION, INTERSECT, EXCEPT
- it does not contain the keyword DISTINCT
- every column in the view corresponds to a uniquely identifiable base table column
- Attributes not in view can be NULL or have default value
- the FROM clause references exactly one table which must be a base table or an updatable view
- the table referenced in the FROM clause in the view cannot be referenced in the FROM clause of a nested WHERE clause
- it does not have a GROUP BY clause or aggregation
- it does not have a HAVING clause

Updatable means insert, delete, update all ok

# Views- definition, use, update

```
CREATE VIEW LOW-ATL-FARES            /* updatable view*/
   AS SELECT *
      FROM FLT-SCHEDULE
      WHERE FROM-AIRPORTCODE="ATL"
      AND PRICE<00200.00;

UPDATE LOW-ATL-FARES                 /* moves row       */
 SET   PRICE = 00250.00              /* outside the view*/
 WHERE TO-AIRPORTCODE = "BOS";

 INSERT INTO LOW-ATL-FARES           /* creates row     */
 VALUES ("DL222", "DELTA",           /* outside the view*/
         "BIR", 11-15-00, "CHI", 13-05-00, 00180.00);

 CREATE VIEW LOW-ATL-FARES
   AS SELECT *
     FROM FLT-SCHEDULE
     WHERE FROM-AIRPORTCODE="ATL"
     AND PRICE<00200.00
 WITH CHECK OPTION;                  /* prevents updates*/
                                     /* outside the view*/
```

# Views- definition, use, update

these views are **not** updatable

**CREATE VIEW** ATL-PRICES
**AS SELECT** AIRLINE, PRICE
**FROM** FLT-SCHEDULE
**WHERE** FROM-AIRPORTCODE="ATL";

not unique

computed attribute

**CREATE VIEW** AVG-ATL-PRICES
**AS SELECT** AIRLINE, AVG(PRICE)
**FROM** FLT-SCHEDULE
**WHERE** FROM-AIRPORTCODE="ATL"
**GROUP BY** AIRLINE;

this view is theoretically updatable, but cannot be updated in SQL

**CREATE VIEW** FLT-SCHED-AND-DAY
**AS SELECT** S.*, D.WEEKDAY
**FROM** FLT-SCHEDULE S, FLT-WEEKDAY D
**WHERE** D.FLT# = S.FLT#;

join of two tables

# Views- definition, use, update

## Queries over materialized views

- View $V$ = ViewQuery($R_1$, $R_2$, …, $R_n$)
- Create table $V$ with schema of query result
- Execute ViewQuery and put results in $V$
- Queries refer to $V$ as if it's a table

## Modifications on materialized views?

- Good news: just update the stored table
- Bad news: base tables must stay in synchronized
  - Same issues as with virtual views

# Picking which materialized views to create

(Efficiency) benefits of a materialized view depend on:

- Size of data
- Complexity of view
- Number of queries using view
- Number of modifications affecting view

- Also "incremental maintenance" versus full recomputation

# Integrity constraints

Impose restrictions on allowable data, beyond those imposed by structure and types

- Non-null constraints
- Key constraints
- Attribute-based and tuple-based constraints
- Referential Integrity (foreign key) constraints
- General assertions

# Integrity- constraints

- constraint: a conditional expression required not to evaluate to *false*
- a constraint cannot be created if it is already violated
- a constraint is enforced from the point of creation forward
- a constraint has a unique name
- if a constraint is violated, its name is made available to the user
- constraints cannot reference parameters or host variables; they are application independent.
- data type checking is a primitive form of constraint

# Integrity Constraints

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- *Types of IC's*: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  - *Domain constraints*: Field values must be of right type. Always enforced.

# Integrity- domain constraints

- associated with a domain; applies to all columns defined on the domain

**CREATE DOMAIN** WEEKDAY CHAR(2)
**CONSTRAINT** IC-WEEKDAY
**CHECK** (**VALUE IN**
( "MO", "TU", "WE", "TH", "FR", "SA", "SU"));

**CREATE DOMAIN** PRICE DECIMAL(7,2)
**CONSTRAINT** IC-PRICE
**CHECK** (**VALUE** > 00000.00 );

**CREATE DOMAIN** FLT# CHAR(5)
**CONSTRAINT** IC-FLT#
**CHECK** (**VALUE NOT NULL**);

# Integrity- base table, column constraints

- associated with a specific base table

```
CREATE TABLE AIRLINE.FLT-SCHEDULE
   (FLT#              FLIGHTNUMBER NOT NULL,
   AIRLINE                      VARCHAR(25),
   FROM-AIRPORTCODE   AIRPORT-CODE,
   DTIME                        TIME,
   TO-AIRPORTCODE          AIRPORT-CODE,
   ATIME                        TIME,
   CONSTRAINT  FLTPK PRIMARY KEY (FLT#),
   CONSTRAINT FROM-AIRPORTCODE-FK
   FOREIGN KEY (FROM-AIRPORTCODE)
    REFERENCES AIRPORT(AIRPORTCODE)
        ON DELETE SET NULL ON UPDATE CASCADE,
   FOREIGN KEY (FROM-AIRPORTCODE)
    REFERENCES AIRPORT(AIRPORTCODE)
        ON DELETE SET NULL ON UPDATE CASCADE,
   CONSTRAINT IC-DTIME-ATIME
          CHECK (DTIME < ATIME);
```

# General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

```
CREATE TABLE  Sailors
        (sid  INTEGER,
        sname  CHAR(10),
        rating  INTEGER,
        age  REAL,
        PRIMARY KEY  (sid),
        CHECK  ( rating >= 1
                AND rating <= 10 )
```

```
CREATE TABLE  Reserves
        ( sname  CHAR(10),
        bid  INTEGER,
        day  DATE,
        PRIMARY KEY  (bid,day),
        CONSTRAINT  noInterlakeResv
        CHECK  (`Interlake' <>
                ( SELECT  B.bname
                FROM  Boats B
                WHERE  B.bid=bid)))
```

# Database Authorization

- Make sure users see only the data that they are supposed to see
- Guard the database against modifications by malicious users
- Users have privileges; can only operate on data for which they are authorized
- **`Grant`** and **`Revoke`** statements
- Beyond simple table-level privileges: use views privacy

# Authorization

## Obtaining Privileges

- Relation creator is owner
- Owner has all privileges and may grant or revoke privileges

- Discretionary Access Control (DAC) is supported by GRANT and REVOKE:

**GRANT** <privileges>

**ON** <table>

**TO** <users>

[**WITH GRANT OPTION**];

**REVOKE** [**GRANT OPTION FOR**] <privileges>

**ON** <table>

**FROM** <users> {**RESTRICT** | **CASCADE**};

<privileges>: SELECT, INSERT(X), INSERT, UPDATE(X), UPDATE, DELETE

<users>: public, U2, U8..

CASCADE: revoke cascades through its subtree

RESTRICT: revoke succeeds only if there is no subtree

# Authorization

**GRANT** INSERT, DELETE
**ON** FLT-SCHEDULE
**TO** U1, U2
**WITH GRANT OPTION**;

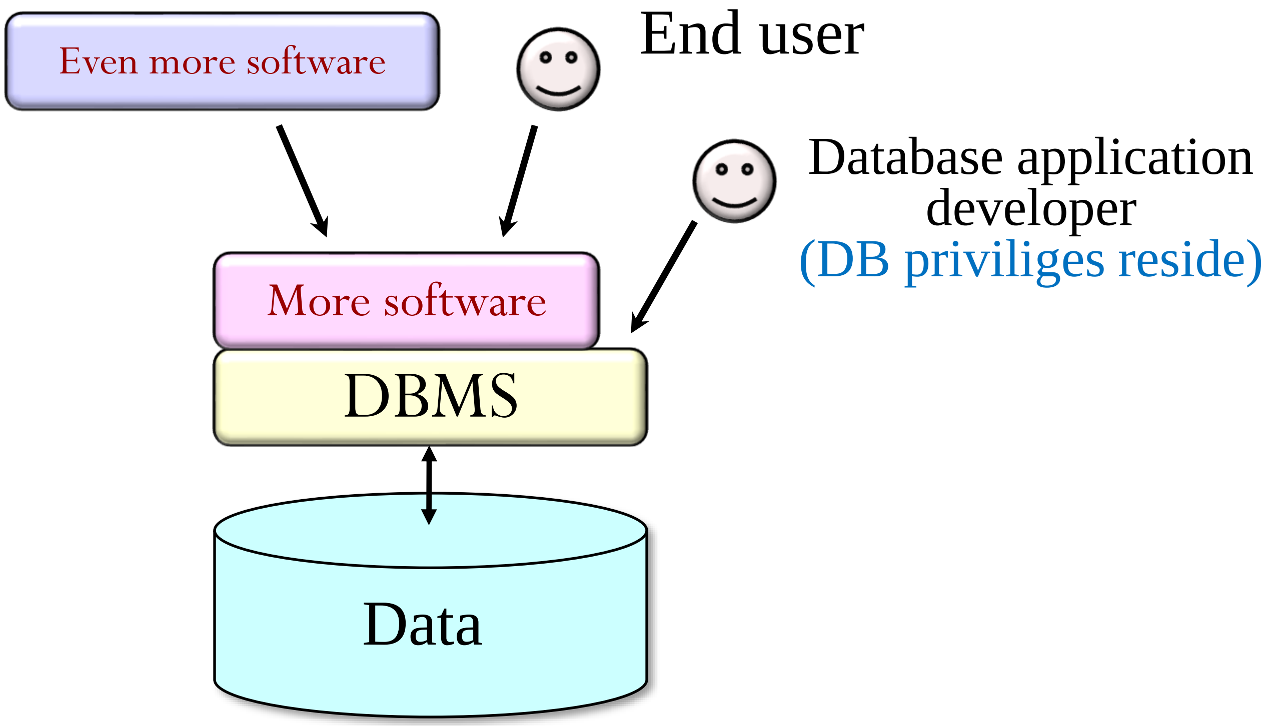
**GRANT** UPDATE(PRICE)
**ON** FLT-SCHEDULE
**TO** U3;

**REVOKE GRANT OPTION FOR** DELETE
**ON** FLT-SCHEDULE
**FROM** U2 **CASCADE**;

**REVOKE** DELETE
**ON** FLT-SCHEDULE
**FROM** U2 **CASCADE**;

# Database Authorization

## Where Privileges Reside



# Summary

- SQL was an important factor in the early acceptance of the relational model; more natural than earlier, procedural query languages.
- Relationally complete; in fact, significantly more expressive power than relational algebra.
- Even queries that can be expressed in Relational Algebra can often be expressed more naturally in SQL.
- Many alternative ways to write a query; optimizer should look for most efficient evaluation plan.
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.
- NULL for unknown field values brings many complications
- SQL allows specification of rich integrity constraints
- Triggers respond to changes in the database
- SQL supports linear recursion to increase the expressiveness.