UNIT 6

Structured Query Language (SQL)

Text: Chapter 5
Learning Goals

Given a database (a set of tables) you will be able to

- express a query in SQL, involving set operators, subqueries and aggregations
- rewrite SQL queries in one style (with one set of operators) with queries in a different style (using another set of operators)
- show that two SQL queries (with or without null values) are/aren’t equivalent
- translate RA (or Datalog) queries to SQL queries and vice versa
- write SQL statements to insert, delete, update the database and define views
- write SQL statements to set certain constraints
- (in the project) use JDBC and Java to design DB transactions for the database users
Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Modification of the Database
- Views
- Integrity Constraints
- Embedded SQL, JDBC
The SQL Query Language

- Developed by IBM (System R) in the 1970s
- Often pronounced as **SEQUEL**!
- Need for a standard since relational queries are used by many vendors

**Standards:**
- SQL-86
- SQL-89 (minor revision)
- SQL-92 (major revision, current standard)
- SQL-99 (major extensions)

**Consists of several parts:**
- Data Definition Language (DDL)
- Data Manipulation Language (DML)
  - Data Query
  - Data Modification
Creating Tables in SQL(DDL)

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

  ```sql
  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint_1),
  ..., (integrity-constraint_k))
  ```

- Integrity constraints (ICs) can be:
  - primary and candidate keys
  - foreign keys
  - general assertions
    - e.g., check (grade between 0 and 100)

- Example:  
  ```sql
  CREATE TABLE Student
  (cid CHAR(20) not null,
  name CHAR(20),
  address CHAR(20),
  phone CHAR(8),
  major CHAR(4),
  primary key (cid))
  ```
Domain Types in SQL

- **char(n)**. Fixed length character string with length $n$.
- **varchar(n)**. Variable length character strings, with maximum length $n$.
- **int**. Integer (machine-dependent).
- **smallint**. Small integer (machine-dependent).
- **numeric(p,d)**. Fixed point number, with user-specified precision of $p$ digits, with $d$ digits to the right of decimal point.
- **real, double precision**. 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.

Null values are allowed in all the domain types.

To preclude null values declare attribute to be **not null**.

**create domain** in SQL-92 and 99 creates user-defined domain types

- e.g., **create domain person-name char(20) not null**
Date/Time Types in SQL

- **date.** Dates, containing a (4 digit) year, month and date
  - E.g. date '2001-7-27'

- **time.** Time of day, in hours, minutes and seconds.
  - E.g. time '09:00:30' time '09:00:30.75'

- **timestamp:** date plus time of day
  - E.g. timestamp '2001-7-27 09:00:30.75'

- **Interval:** period of time
  - E.g. Interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

- Relational DBMS offer a variety of functions to
  - extract values of individual fields from date/time/timestamp
  - convert strings to dates and vice versa
  - For instance in Oracle (date is a timestamp):
    - TO_CHAR( date, format)
    - TO_DATE( string, format)
    - format looks like: 'DD-Mon-YY HH:MI.SS'
Running Examples

Customer Database
- **Customer**$(cid: \text{integer}, \ cname: \text{string}, \ rating: \text{integer}, \ salary: \text{real})$
- **Item**$(iid: \text{integer}, \ iname: \text{string}, \ type: \text{string})$
- **Order**$(cid: \text{integer}, \ iid: \text{integer}, \ day: \text{date}, \ qty: \text{real})$

Student Database
- **Student**$(sid, \ name, \ address, \ phone, \ major)$
- **Course**$(dept, \ cno, \ title, \ credits)$
- **Instructor**$(iname, \ degree)$
- **Section**$(dept, \ cno, \ secno, \ term, \ ins\_name)$
- **Enrolled**$(sid, \ dept, \ cno, \ secno, \ term, \ mark)$
- **Prerequisite**$(dept, \ cno, \ pre\_dept, \ pre\_cno)$
Basic SQL Query

- SQL is based on set and relational operations with certain modifications and enhancements.
- A typical SQL query has the form:
  
  ```
  select [distinct] A_1, A_2, ..., A_n 
  from r_1, r_2, ..., r_m 
  where P
  ```

  - $A_i$s represent attributes.
  - $r_i$s represent relations.
  - $P$ is a predicate.

- This query is nearly equivalent to the relational algebra expression.

  $$
  \Pi_{A_1, A_2, ..., A_n} (\sigma_P (r_1 \times r_2 \times ... \times r_m))
  $$

- The result of a SQL query is a table (relation).
- If `distinct` is used, duplicates are eliminated. By default duplicates are not eliminated!

Dub-elim is expensive. How would you implement it?
Basic SQL Query

- Called *conjunctive query*.
- Equivalent RA expression involves select, project, and join.
- So, also called SPJ query.
- SPJ/conjunctive queries correspond in Datalog to:
  
  $$ p(\bar{X}) \leftarrow r_1(\bar{Y}), \ldots, r_k(\bar{Z}), V_i \text{op} U_j, \ldots, V_\ell \text{op} c, \ldots $$

- Union of SPJ queries (i.e., SPJU) queries => set of Datalog rules with the same head $p(\bar{X})$.
- SPJ with aggregation => SPJA queries; no counterpart in (pure datalog), although researchers have extended Datalog with aggregation (we won’t cover this).
Conceptual Evaluation Strategy

- Typical SQL query:
  ```sql
  SELECT [DISTINCT] attr-list
  FROM relation-list
  WHERE qualification
  ```

- Semantics of a SQL query defined in terms of the following conceptual evaluation strategy (in order):
  - Compute the cross-product of `relation-list`.
  - Discard any resulting tuples that fail `qualification`.
  - Drop attributes that are not in `attr-list`.
  - If `DISTINCT` is specified, eliminate duplicate rows.

- This strategy is not a super efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.
### Example Instances

We will use these instances of the Customer, Item and Order relations in our examples.

### Customer

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>rating</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>J. Justin</td>
<td>7</td>
<td>70</td>
</tr>
<tr>
<td>35</td>
<td>G. Grumpy</td>
<td>8</td>
<td>90</td>
</tr>
<tr>
<td>50</td>
<td>R. Rusty</td>
<td>10</td>
<td>80</td>
</tr>
</tbody>
</table>

### Item

<table>
<thead>
<tr>
<th>iid</th>
<th>iname</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Inspiron6400</td>
<td>laptop</td>
</tr>
<tr>
<td>102</td>
<td>LatituteD520</td>
<td>laptop</td>
</tr>
<tr>
<td>105</td>
<td>DimensionE520</td>
<td>desktop</td>
</tr>
<tr>
<td>110</td>
<td>CanonMP830</td>
<td>printer</td>
</tr>
</tbody>
</table>

### Order

<table>
<thead>
<tr>
<th>cid</th>
<th>iid</th>
<th>day</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>102</td>
<td>10/10/06</td>
<td>2</td>
</tr>
<tr>
<td>50</td>
<td>105</td>
<td>11/12/06</td>
<td>5</td>
</tr>
</tbody>
</table>
Example of Conceptual Evaluation

```
SELECT  cname
FROM    Customer, Order
WHERE Customer.cid=Order.cid AND iid=105
```

**Customer X Order**

<table>
<thead>
<tr>
<th>(cid)</th>
<th>cname</th>
<th>rating</th>
<th>salary</th>
<th>(cid)</th>
<th>iid</th>
<th>day</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>J. Justin</td>
<td>7</td>
<td>70</td>
<td>40</td>
<td>102</td>
<td>10/10/06</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>J. Justin</td>
<td>7</td>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>G. Grumpy</td>
<td>8</td>
<td>90</td>
<td>40</td>
<td>102</td>
<td>10/10/06</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>G. Grumpy</td>
<td></td>
<td></td>
<td>50</td>
<td>105</td>
<td>11/12/06</td>
<td>5</td>
</tr>
<tr>
<td>50</td>
<td>R. Rusty</td>
<td>10</td>
<td>80</td>
<td>40</td>
<td>102</td>
<td>10/10/06</td>
<td>2</td>
</tr>
<tr>
<td>50</td>
<td>R. Rusty</td>
<td></td>
<td></td>
<td>50</td>
<td>105</td>
<td>11/12/06</td>
<td>5</td>
</tr>
</tbody>
</table>
Renaming Attributes in Result

- SQL allows renaming relations and attributes using the `as` clause:

  \[ old-name \text{ as new-name } \]

- Example: Find the name of customers who have ordered item 105 and the day they placed the order; rename `cname` to “customer_name”:

  ```sql
  SELECT  cname AS customer_name, day
  FROM     Customer, Order
  WHERE   Customer.cid=Order.cid AND iid=105
  ```
Range Variables

- We can use variables to name relations in the FROM clause
  - Usually used when same relation appears twice.
- The previous query can also be written as:

  ```
  SELECT  cname, day
  FROM    Customer C, Order R
  WHERE   C.cid=R.cid AND iid=105
  ```

  OR

  ```
  SELECT  C.cname, R.day
  FROM    Customer C, Order R
  WHERE   C.cid=R.cid AND R.iid=105
  ```

  Nothing but
  
  ```
  ans(N, D) ← customer(I, N, R, S), order(I, `105', D, Q).
  ```
Using DISTINCT

- Find customers (id’s) who’ve ordered at least one item:
  
  ```
  SELECT C.cid
  FROM Customer C, Order R
  WHERE C.cid=R.cid
  ```

- Would adding DISTINCT to this query make a difference?

- Suppose we replace `C.cid` by `C.cname` in the SELECT clause. Would adding DISTINCT to this variant of the query make a difference?

- What if we use * in SELECT (* selects whole tuples)?
  
  ```
  SELECT *
  FROM Customer C, Order R
  WHERE C.cid=R.cid
  ```
Expressions and Strings

SELECT  C.salary, tax=(C.salary-10)*0.3, C.salary*0.01 AS prof_fees
FROM  Customer C
WHERE  C.cname LIKE ‘B_%B’

Illustrates use of arithmetic expressions and string pattern matching:

Returns triples of values, each consisting of
the salary
the income tax deducted (30% of salary minus 10K)
and
the professional fees (1% of the salary)
for customers whose names begin and end with B and contain at least three characters.

AS and = are two ways to name fields in result.
More on Strings

- **LIKE** is used for string matching:
  - `_' stands for any one character and
  - `%' stands for 0 or more arbitrary characters.

- To match the name “Strange%”, need to use an escape character:

  ```
  like 'Strange\%' escape '\'
  ```

- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.
Ordering of Tuples

- List in alphabetic order the names of the customers who have ordered a laptop

  ```sql
  select  cname
  from     Customer, Item, Order
  where    Customer.cid = Order.cid and
           Item.iid = Order.iid and type='laptop'
  order by cname
  ```

- Order is specified by:
  - `desc` for descending order or
  - `asc` for ascending order; ascending order is the default.
  - E.g., `order by cname desc`

Ordering goes beyond RA and Datalog!
Set Operations

- **union**, **intersect**, and **except** operate on tables (relations) and correspond to the RA operations $\cup$, $\cap$, $\neg$.

- Each of the above operations automatically eliminates duplicates;
  To retain all duplicates use the corresponding multiset versions:
    
    union all, intersect all and except all.

- Suppose a tuple occurs $m$ times in $r$ and $n$ times in $s$, then, it occurs:
  
  $\begin{align*}
  &m + n \text{ times in } r \text{ union all } s \\
  &\text{min}(m,n) \text{ times in } r \text{ intersect all } s \\
  &\text{max}(0, m - n) \text{ times in } r \text{ except all } s
  \end{align*}$
Set Operations: UNION

- **Example:** Find cid’s of customers who’ve ordered a laptop or a desktop
  
  ```sql
  SELECT C.cid
  FROM Customer C, Item I, Order R
  WHERE C.cid=R.cid AND R.iid=I.iid AND (I.type='laptop' OR I.type='desktop')
  UNION
  SELECT C.cid
  FROM Customer C, Item I, Order R
  WHERE C.cid=R.cid AND R.iid=I.iid AND I.type='laptop'
  UNION
  SELECT C.cid
  FROM Customer C, Item I, Order R
  WHERE C.cid=R.cid AND R.iid=I.iid AND I.type='desktop'
  ```

- **UNION** can be used to compute the union of any two *compatible* (corresponding attributes have same domains) sets of tuples (which are themselves the result of SQL queries):
  
  ```sql
  SELECT C.cid
  FROM Customer C, Item I, Order R
  WHERE C.cid=R.cid AND R.iid=I.iid AND I.type='laptop'
  UNION
  SELECT C.cid
  FROM Customer C, Item I, Order R
  WHERE C.cid=R.cid AND R.iid=I.iid AND I.type='desktop'
  ```

What do we get if we replace OR by AND here?
Set Operations : EXCEPT

- **EXCEPT** can be used to compute the difference of two *compatible* sets of tuples.
- Some systems use **MINUS** instead of **EXCEPT**.
- What does the following query return?

```sql
SELECT C.cid
FROM Customer C, Item I, Order R
WHERE C.cid = R.cid AND R.iid = I.iid AND I.type = 'laptop'
EXCEPT
SELECT C.cid
FROM Customer C, Item I, Order R
WHERE C.cid = R.cid AND R.iid = I.iid AND I.type = 'desktop'
```

What does **EXCEPT** remind you of in Datalog?
Set Operations: INTERSECT

- **Example:** Find cid’s of customers who’ve ordered a laptop and a desktop item:
  ```sql
  SELECT C.cid
  FROM Customer C, Item I1, Order R1, Item I2, Order R2
  WHERE C.cid=R1.cid AND R1.iid=I1.iid
  AND C.cid=R2.cid AND R2.iid=I2.iid
  AND I1.type='laptop' AND I2.type='desktop')
  ```

- **INTERSECT** can be used to compute the intersection of two compatible sets of tuples (included in SQL/92, but some systems may not support it).
  ```sql
  SELECT C.cid
  FROM Customer C, Item I, Order R
  WHERE C.cid=R.cid AND R.iid=I.iid AND I.type='laptop'
  INTERSECT
  SELECT S.cid
  FROM Customer S, Item I, Order R
  WHERE C.cid=R.cid AND R.iid=I.iid AND I.type='desktop'
  ```

Important to include the Key!
Nested Queries

Find names of customers who’ve ordered item #105:

```sql
SELECT C.cname
FROM Customer C
WHERE C.cid IN (SELECT R.cid
                 FROM Order R
                 WHERE iid=105)
```

- A very powerful feature of SQL: a `WHERE` clause can itself contain a SQL query! (Actually, so can `FROM` and `HAVING` clauses.)
- To find customers who’ve not ordered item #105, use `NOT IN`.
- To understand semantics of nested queries, think of a `nested loops` evaluation:
  - For each `Customer` tuple, check the qualification by computing the `subquery`.
Nested Queries with Correlation

Find names of customers who’ve ordered item #105:

```
SELECT C.cname
FROM Customer C
WHERE EXISTS (SELECT *
               FROM Order R
               WHERE iid=105 AND C.cid=R.cid)
```

- **EXISTS** is another set operator: *returns true if the set is not empty.*
- **UNIQUE** checks for duplicate tuples: *returns true if there are no duplicates.*
- If **UNIQUE** is used above, and * is replaced by *iid*, finds customers with at most one order for item #105.
  - (* denotes all attributes. Why do we have to replace * by iid?*)
- Illustrates why, in general, subquery must be re-computed for each Customer tuple.
More on Set-Comparison Operators

- We’ve already seen **IN, EXISTS** and **UNIQUE**. Can also use **NOT IN, NOT EXISTS** and **NOT UNIQUE**.

- Also available: **op ANY, op ALL**, where **op** is one of: >, <, =, <=, >=, <>

- Find customers whose salary is greater than that of every customer with last name “Rusty”:

  ```sql
  SELECT * 
  FROM Customer C 
  WHERE C.salary > ALL (SELECT C2.salary 
                         FROM Customer C2 
                         WHERE C2.cname LIKE ‘% Rusty’) 
  ```

How did we write such queries in RA?
How about Datalog?
Rewriting INTERSECT Queries Using IN

Find cid’s of customers who’ve ordered both a laptop and a desktop:

```sql
SELECT  C.cid
FROM    Customer C, Item I, Order R
WHERE   C.cid=R.cid AND R.iid=I.iid AND I.type='laptop'
        AND C.cid IN (SELECT  C2.cid
                        FROM    Customer C2, Item I2, Order R2
                        WHERE   C2.cid=R2.cid AND R2.iid=I2.iid
                                AND I2.type='desktop')
```

- Similarly, EXCEPT queries can be re-written using NOT IN.
- To find names (not cids) of customers who’ve ordered both laptops and desktops, just replace `C.cid` by `C.cname` in SELECT clause. Could we replace cid by cname throughout? (What about INTERSECT query?)
Division in SQL

Find customers who’ve ordered all items.

Let’s do it the hard way without EXCEPT:

1. SELECT  
cname  
FROM     Customer C  
WHERE  NOT EXISTS  (SELECT  *  
FROM  Item I  
WHERE  NOT EXISTS  (SELECT  *  
FROM  Order R  
WHERE  R.iid=I.iid  
AND  R.cid=C.cid))

2. SELECT  
cname  
FROM     Customer C  
WHERE  NOT EXISTS  
((SELECT  I.iid  
FROM  Item I)  
EXCEPT  
(SELECT  R.iid  
FROM  Order R  
WHERE  R.cid=C.cid))

select customer C such that ...
there is no item I...
which is not ordered by C

How does it compare with RA/Datalog?
Aggregate Operators

- These functions operate on the multiset of values of a column of a relation, and return a value:
  - **AVG**: average value
  - **MIN**: minimum value
  - **MAX**: maximum value
  - **SUM**: sum of values
  - **COUNT**: number of values

- The following versions eliminate duplicates before applying the operation to attribute A:
  - COUNT ( DISTINCT A)
  - SUM ( DISTINCT A)
  - AVG ( DISTINCT A)
Aggregate Operators: Examples

SELECT COUNT (*) FROM Customer

SELECT count
FROM Customer C
WHERE C.rating = (SELECT MAX(C2.rating)
                   FROM Customer C2)

SELECT AVG (salary)
FROM Customer
WHERE rating = 10

SELECT AVG (DISTINCT salary)
FROM Customer
WHERE rating = 10

SELECT COUNT (DISTINCT rating)
FROM Customer
WHERE salary BETWEEN 50 AND 100
Aggregate Operators: Examples (cont)

Find name and salary of the richest customer(s)

- The first query is wrong! WHY?
- Second query is fine: can use value = subquery only if subquery returns single value.
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
SELECT   cname, MAX (salary)
FROM     Customer

SELECT   cname, salary
FROM     Customer
WHERE    salary =
          (SELECT  MAX (salary)
           FROM    Customer)

SELECT   cname, salary
FROM     Customer
WHERE    (SELECT  MAX (salary)
          FROM    Customer) = salary
```
GROUP BY and HAVING

- Often, we want to divide tuples into groups and apply aggregate operations to each group.

Example: Find the average salary of the customers in each rating level.

  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

    For $i = 1, 2, \ldots, 10$: 
    
    ```sql
    SELECT AVG(salary)
    FROM Customer
    WHERE rating = i
    ```

- Problem:
  - We don’t know how many rating levels exist, and what the rating values for these levels are!

- Solution:
  - Use “GROUP BY” and/or “HAVING” clauses
GROUP BY and HAVING (cont)

### SQL Syntax

<table>
<thead>
<tr>
<th>SELECT</th>
<th>[DISTINCT] target-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>relation-list</td>
</tr>
<tr>
<td>WHERE</td>
<td>qualification</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>grouping-list</td>
</tr>
<tr>
<td>HAVING</td>
<td>group-qualification</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>target-list</td>
</tr>
</tbody>
</table>

- **The *target-list* contains**
  *(i)* attribute names
  *(ii)* terms with aggregate operations (e.g., `AVG (salary)`).

- **Attributes in (i) must also be in *grouping-list***.
  - each answer tuple corresponds to a **group**,
  - a **group** is a set of tuples that have the same value for all attributes in *grouping-list*
  - selected attributes in (i) must have a single value per group.

- **Attributes in *group-qualification* are either in *grouping-list*** or are arguments to an aggregate operator.
Conceptual Evaluation of a Query

1. compute the cross-product of relation-list
2. keep only tuples that satisfy qualification
3. partition selected tuples into groups by the value of attributes in grouping-list
4. keep only the groups that satisfy group-qualification (expressions in group-qualification must have a single value per group!)
5. keep only the fields that are in target-list
6. generate one answer tuple per qualifying group.
GROUP BY Example

Example1: For each item, find the salary of the poorest customer who has ordered this item:

```
SELECT iid, MIN(salary)
FROM Customer C, Order R
WHERE C.cid = R.cid
GROUP BY iid
```
GROUP BY Example: Default Evaluation

**Customer**

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>rating</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>J. Justin</td>
<td>7</td>
<td>70</td>
</tr>
<tr>
<td>35</td>
<td>G. Grumpy</td>
<td>8</td>
<td>90</td>
</tr>
<tr>
<td>50</td>
<td>R. Rusty</td>
<td>10</td>
<td>80</td>
</tr>
</tbody>
</table>

**Order**

<table>
<thead>
<tr>
<th>cid</th>
<th>iid</th>
<th>day</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>102</td>
<td>10/10/06</td>
<td>2</td>
</tr>
<tr>
<td>50</td>
<td>105</td>
<td>11/12/06</td>
<td>5</td>
</tr>
<tr>
<td>40</td>
<td>102</td>
<td>25/06/07</td>
<td>3</td>
</tr>
<tr>
<td>35</td>
<td>102</td>
<td>30/06/07</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ \sigma_{C.cid=R.cid} \text{ (Customer X Order)} \]

<table>
<thead>
<tr>
<th>(cid)</th>
<th>cname</th>
<th>rating</th>
<th>salary</th>
<th>(cid)</th>
<th>iid</th>
<th>day</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>J. Justin</td>
<td>7</td>
<td>70</td>
<td>40</td>
<td>102</td>
<td>10/10/06</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>J. Justin</td>
<td>7</td>
<td>70</td>
<td>40</td>
<td>102</td>
<td>25/06/07</td>
<td>3</td>
</tr>
<tr>
<td>35</td>
<td>G. Grumpy</td>
<td>8</td>
<td>90</td>
<td>35</td>
<td>102</td>
<td>30/06/07</td>
<td>5</td>
</tr>
<tr>
<td>50</td>
<td>R. Rusty</td>
<td>10</td>
<td>80</td>
<td>50</td>
<td>105</td>
<td>11/12/06</td>
<td>5</td>
</tr>
</tbody>
</table>
GROUP BY Example (cont’)

σ_{C.cid=R.cid} (Customer X Order) and grouped by iid

<table>
<thead>
<tr>
<th>(cid)</th>
<th>cname</th>
<th>rating</th>
<th>salary</th>
<th>(cid)</th>
<th>iid</th>
<th>day</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>J. Justin</td>
<td>7</td>
<td>70</td>
<td>40</td>
<td>102</td>
<td>10/10/06</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>J. Justin</td>
<td>7</td>
<td>70</td>
<td>40</td>
<td>102</td>
<td>25/06/07</td>
<td>3</td>
</tr>
<tr>
<td>35</td>
<td>G. Grumpy</td>
<td>8</td>
<td>90</td>
<td>35</td>
<td>102</td>
<td>30/06/07</td>
<td>5</td>
</tr>
<tr>
<td>50</td>
<td>R. Rusty</td>
<td>10</td>
<td>80</td>
<td>50</td>
<td>105</td>
<td>11/12/06</td>
<td>5</td>
</tr>
</tbody>
</table>

Result

<table>
<thead>
<tr>
<th>iid</th>
<th>Min(salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>70</td>
</tr>
<tr>
<td>105</td>
<td>80</td>
</tr>
</tbody>
</table>
GROUP BY and HAVING Example

**Example2:** For each item that has more than 2 orders, find the salary of the poorest customer who has ordered this item:

```
SELECT     iid, MIN (salary)
FROM       Customer C, Order R
WHERE      C.cid= R.cid
GROUP BY   iid
HAVING     COUNT (*) > 2
```
Grouping Examples (contd.)

For each laptop item, find the number of (distinct) customers who ordered this item

```
SELECT  I.iid,  COUNT (DISTINCT R.cid) AS scount
FROM    Item I, Order R
WHERE   R.iid=I.iid AND I.type='laptop'
GROUP BY I.iid
```

- Grouping over a join of two relations.
- What do we get if we:
  (a) remove `I.type='laptop'` from the `WHERE` clause, and then
  (b) add a `HAVING` clause with this dropped condition?
- What if we replace
  ```
  COUNT (DISTINCT R.cid)
  ```
  with
  ```
  COUNT (*)
  ```
More Grouping Examples

For each rating that has at least 2 customers whose salary is at least 50K, find the average salary of these customers for that rating.

```
SELECT rating, AVG(salary)
FROM Customer
WHERE salary >= 50
GROUP BY rating
HAVING COUNT(*) > 1
```

- Only rating can appear alone in the SELECT and/or HAVING clauses.
- 2nd column of result is unnamed. (Use AS to name it.)

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>rating</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>J. Justin</td>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>31</td>
<td>R. Rubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>71</td>
<td>Z. Zorba</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>H. Hasty</td>
<td>7</td>
<td>80</td>
</tr>
<tr>
<td>29</td>
<td>B. Brutus</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>58</td>
<td>R. Rusty</td>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>

Answer

<table>
<thead>
<tr>
<th>rating</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>80</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>

Relation: Answer
Grouping Examples (cont’)

For each rating that has at least 2 customers (of any salary), find the average salary among the customers with that rating whose salary is at least 50K.

```
SELECT  C.rating,  AVG (C.salary)
FROM    Customer C
WHERE   C.salary  >=  50
GROUP BY C.rating
HAVING  1  <  (SELECT  COUNT (*)
               FROM    Customer C2
               WHERE   C2.rating=C.rating)
```

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with at least 2 customers with salary at least 50K!
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) > 1
Grouping Examples (contd.)

Find those ratings for which their average salary is the minimum over all ratings

```sql
SELECT  C.rating
FROM     Customer C
WHERE  C.salary = (SELECT  MIN (AVG (C2.salary))  FROM Customer C2)
```

WRONG! Aggregate operations cannot be nested!

Correct solution (in SQL/92 and SQL/99)

```sql
SELECT  Temp.rating, Temp.avgsalary
FROM     (SELECT  C.rating, AVG (C.salary) AS avgsalary
           FROM  Customer C
           GROUP BY  C.rating) AS Temp
WHERE  Temp.avgsalary = (SELECT  MIN (Temp.avgsalary)
                          FROM  Temp)
```
Null Values

- Tuples may have a null value, denoted by \textit{null}, for some of their attributes.
- Value \textit{null} signifies an unknown value or that a value does not exist.
- The predicate \texttt{IS NULL ( IS NOT NULL )} can be used to check for null values.
  - E.g., \textit{Find the names of the customers whose salary is not known.}
    
    ```
    SELECT cname
    FROM Customer
    WHERE salary IS NULL
    ```

- The result of any arithmetic expression involving \textit{null} is \textit{null}.
  - E.g., $5 + \text{null}$ returns \textit{null}.
### Null Values and Three-Valued Logic

To deal with null values we need a three-valued logic using the truth value *unknown*:

<table>
<thead>
<tr>
<th>OR</th>
<th>False</th>
<th>Unknown</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>F</td>
<td>U</td>
<td>T</td>
</tr>
<tr>
<td>Unknown</td>
<td>U</td>
<td>U</td>
<td>T</td>
</tr>
<tr>
<td>True</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Classical Logic is 2-valued.

Lattice of truth values in Kleene’s 3-valued logic.
Null Values and Three-Valued Logic

- To deal with null values we need a three-valued logic using the truth value *unknown*:

<table>
<thead>
<tr>
<th>AND</th>
<th>False</th>
<th>Unknown</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Unknown</td>
<td>F</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>True</td>
<td>F</td>
<td>U</td>
<td>T</td>
</tr>
</tbody>
</table>

Classical Logic is 2-valued.

Lattice of truth values in Kleene’s 3-valued logic.
Null Values and Three-Valued Logic

To deal with null values we need a three-valued logic using the truth value *unknown*:

<table>
<thead>
<tr>
<th></th>
<th>False</th>
<th>Unknown</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>T</td>
<td>U</td>
<td>F</td>
</tr>
</tbody>
</table>

Classical Logic is 2-valued.

Lattice of truth values in Kleene’s 3-valued logic.
Null Values and Three-Valued Logic

In particular, note:

- **OR:** \( (\text{unknown or true}) = \text{true} \), \( (\text{unknown or false}) = \text{unknown} \)
  \( (\text{unknown or unknown}) = \text{unknown} \)
- **AND:** \( (\text{true and unknown}) = \text{unknown} \), \( (\text{false and unknown}) = \text{false} \)
  \( (\text{unknown and unknown}) = \text{unknown} \)
- **NOT:** \( (\text{not unknown}) = \text{unknown} \)
- “\( P \) is unknown” evaluates to true if predicate \( P \) evaluates to \text{unknown}

Any comparison with **null** returns **unknown**

- E.g. \( 5 < \text{null or null <> null or null = null} \)

Result of **where** clause predicate is treated as **false** if predicate evaluates to **unknown**

All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.
Summary of “impact” of null values

- Represent “unknown” or “inapplicable”.
- Make arithmetic expressions evaluate to “unknown” (U).
- Make logical expressions evaluate to T, U, of F.
- SQL treats truth value U in where clause as F.
- SQL ignores tuples with nulls on aggregated attr when aggregating using any function save **Count**.
- SQL counts all tuples regardless of presence of nulls.
- SQL lets you test if a value is null via IS NULL and IS NOT NULL.
Database Modification – Insertion

- Can insert a single tuple using:
  
  ```sql
  INSERT INTO Student 
  VALUES (53688, 'Smith', '222 W.15th ave', 333-4444, MATH)
  ```

- or

  ```sql
  INSERT INTO Student (sid, name, address, phone, major) 
  VALUES (53688, 'Smith', '222 W.15th ave', 333-4444, MATH)
  ```

- Add a tuple to student with null address and phone:

  ```sql
  INSERT INTO Student (sid, name, address, phone, major) 
  VALUES (33388, 'Chan', null, null, CPSC)
  ```

What if there was a “not null” constraint on address or phone?
Can add values selected from another table?

Add an order for customer 222 for every laptop item with date 1/1/07 and quantity 5

```
INSERT INTO Order
SELECT 222, iid, '1/1/07', 5
FROM Item
WHERE type = 'laptop'
```

The *select-from-where statement* is fully evaluated before any of its results are inserted.

So, statements like

```
INSERT INTO table1 SELECT ...... FROM table1
```

are ok.

How would you say add orders for a specific item (iid) from every customer rated at 6 or above?
Database Modification – Deletion

- Note that only whole tuples are deleted.
- Can delete all tuples satisfying some condition (e.g., name = Smith):
  
  \[
  \text{DELETE FROM Student} \\
  \text{WHERE name = 'Smith'}
  \]

- Delete all Customers whose salary is above the average sailor salary:
  
  \[
  \text{DELETE FROM Customer} \\
  \text{WHERE salary > (SELECT avg(salary) FROM Customer)}
  \]

Do you see any problem with this?
Database Modification – Updates

- Increase the rating of all customers by 2 (should not be more than 10)

- Need to write two updates:
  - UPDATE Customer
    SET rating = 10
    WHERE rating >= 8
  
  - UPDATE Customer
    SET rating = rating + 2
    WHERE rating < 8

- Is the order important?

- How would you raise by 10% the salary of every customer rated at 8 or above?
Views

- Provide a mechanism to hide certain data from certain users. To create a view we use the command:

  \[
  \text{CREATE VIEW } \text{vname AS } \langle\text{query expression}\rangle
  \]

  where:
  - \langle\text{query expression}\rangle\text{ is any legal SQL expression}
  - \text{vname \text{ is the view name}}

- Example:

  \[
  \text{CREATE VIEW } \text{LDO}rder \text{ AS }
  \text{SELECT } \text{cid, iid, type, date }
  \text{FROM Item I, Order R }
  \text{WHERE I.iid = R.iid AND (type = 'laptop' OR type = 'desktop')}
  \]
With Clause

- Allows views to be defined locally to a query, rather than globally.
- Example:

```sql
WITH LDOder(cid, iid, type, date) AS
    SELECT cid, iid, type, date
    FROM Item I, Order R
    WHERE I.iid = R.iid AND (type = 'laptop' OR
    type = 'desktop')

SELECT cid, cname
FROM Customer C, LDOder R
WHERE C.cid = R.cid AND date > '1/1/07'
```
SQL’s Join Operators

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the from clause.
- Join condition - defines which tuples in the two relations match, and what attributes are present in the result of the join.
- Join type - defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

<table>
<thead>
<tr>
<th>Join Types</th>
<th>Join Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>inner join</td>
<td>natural</td>
</tr>
<tr>
<td>left outer join</td>
<td>on &lt;predicate&gt;</td>
</tr>
<tr>
<td>right outer join</td>
<td></td>
</tr>
<tr>
<td>full outer join</td>
<td></td>
</tr>
</tbody>
</table>

Example: 

```sql
SELECT S.name, E.dept, E.cno
FROM Student S NATURAL LEFT OUTER JOIN Enrolled E
```
Integrity Constraints (Review)

- 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., *cid* is a key), or prevent inconsistencies (e.g., *cname* has to be a string, *salary* must be < 200)

- **Types of IC’s:**
  - domain constraints,
  - primary key constraints,
  - foreign key constraints,
  - general constraints
General Constraints

- Create with a CHECK clause.
- Constraints can be named.
- Can use subqueries to express constraint.

```sql
CREATE TABLE Customer
  cid  INTEGER,
cname CHAR(10),
rating  INTEGER,
salary  REAL,
PRIMARY KEY (cid),
CHECK (rating >= 1
      AND rating <= 10);
```

```sql
CREATE TABLE Order1
  (cid  INTEGER,
iid  INTEGER,
day  DATE,
qty  REAL,
PRIMARY KEY (cid, iid, day),
CHECK (Printer <>
      (SELECT I.type
         FROM Item I
         WHERE I.iid=iid)));
```

Check constraints are checked when tuples are inserted or modified.
Domain Constraints

- User can create a new domain and set constrains for it.
- Example:
  ```sql
  CREATE DOMAIN agedomain INTEGER DEFAULT 21
  CHECK ( VALUE >= 1 AND VALUE <= 110 )
  ```

- New systems support distinct types.
Constraints Over Multiple Relations

- Cannot be defined in one table.
- Are defined as ASSERTIONs which are not associated with any one table.
- **Example:** Every student has taken at least one course.

```sql
CREATE ASSERTION totalEnrolment
CHECK
(NOT EXISTS ((SELECT sid FROM student)
EXCEPT
(SELECT sid FROM Enrolled)));
```

What would this IC look like in Datalog-like syntax?
Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  - Transactions are started implicitly and terminated by one of
    - commit work: makes all updates of the transaction permanent in the database
    - rollback work: undoes all updates performed by the transaction.

Example

- Transfer of money from account A to account B involves two steps:
  - deduct from A and add to B
- If one step succeeds and the other fails, database is in an inconsistent state
- Therefore, either both steps should succeed or neither should

If any step of a transaction fails, all work done by the transaction can be undone by rollback work.

Rollback of incomplete transactions is done automatically, in case of system failures.
Transactions (Contd.)

- In most database systems, each SQL statement that executes successfully is automatically committed.  
  - Each transaction consists of only a single statement
- Automatic commit can usually be turned off, but how to do so depends on the database system

- Another option in SQL:1999: enclose statements within
  
  ```sql
  begin atomic
  ...
  end
  ```
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.

- Consists of a data definition, data manipulation and query language.

- Many alternative ways to write a query; optimizer looks for most efficient evaluation plan.
  - Holy Grail: users don’t have to care about efficiency, and relegate finding an efficient plan to QOzer.
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.
Summary (Contd.)

- NULL for unknown field values brings many complications
- SQL allows specification of rich integrity constraints (and triggers)
- Embedded SQL allows execution within a host language; cursor mechanism allows retrieval of one record at a time
- APIs such as ODBC and JDBC introduce a layer of abstraction between application and DBMS