Handling One-to-Many Relations - Grouping and Aggregation

- **No class/lab next Tuesday**
  - Today's lab is also Problem Set #1 - due two weeks from today
  - We won't cover all of today's lab exercises in this lab preparation talk - the rest will be covered this Thursday
- **Handling One-to-Many Relations**
  - Today's lab prep will review SQL queries from last Thursday's lecture notes
  - Key concept is use of GROUP BY statements to handle one-to-many relations
  - We also introduce another database (of URISA proceedings) to learn more about SQL queries and relational database design

- **SQL queries and GROUP BY expressions**
  - See the Sample [Parcel Database](#)
  - See [SQL Notes](#) and Oracle help
  - See [Lab I](#) examples
  - The basic SELECT statement to query one or more tables:

```sql
SELECT [DISTINCT] column_name1[, column_name2, ...]
    FROM table_name1[, table_name2, ...]
    WHERE search_condition1
    [AND search_condition2 ...]
    [OR search_condition3 ...]  
    [GROUP BY column_names]
    [ORDER BY column_names];
```

  - Note that the order of the clauses matters! The clauses, if included, must appear in the order shown! Oracle will report an error if you make a mistake, but the error message (e.g., "ORA-00933: SQL command not properly ended") may not be very informative.

* Kindly refer to Lecture Notes Section
Aggregation: GROUP BY, Group Functions

Simple GROUP BY Example From the Parcels Database

When a SQL statement contains a GROUP BY clause, the rows are selected using the criteria in the WHERE clause and are then aggregated into groups that share common values for the GROUP BY expressions. The HAVING clause may be used to eliminate groups after the aggregation has occurred.

These examples draw on the PARCELS sample database* that we have used previously.

1. List all the fires, including the date of the fire:

   ```sql
   SELECT PARCELID, FDATE
   FROM FIRES
   ORDER BY PARCELID, FDATE;
   ```

2. List the count of fires by parcel:

   ```sql
   SELECT PARCELID, COUNT(FDATE) AS FIRE_COUNT
   FROM FIRES
   GROUP BY PARCELID
   ORDER BY PARCELID;
   ```

   Groups are shown in color, but this query does not collapse groups into a single row.

<table>
<thead>
<tr>
<th>PARCELID</th>
<th>FDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>02-AUG-88</td>
</tr>
<tr>
<td>2</td>
<td>02-APR-89</td>
</tr>
<tr>
<td>3</td>
<td>26-JUL-89</td>
</tr>
<tr>
<td>3</td>
<td>26-JUL-90</td>
</tr>
<tr>
<td>7</td>
<td>01-AUG-87</td>
</tr>
<tr>
<td>20</td>
<td>02-JUL-89</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARCELID</th>
<th>FIRE_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
</tbody>
</table>

The Different Roles of the WHERE Clause and the HAVING Clause

The WHERE clause restricts the rows that are processed before any grouping occurs.

The HAVING clause is used with GROUP BY to limit the groups returned after grouping has occurred.

* Kindly refer to Lecture Notes Section
### 3. List all the fires that occurred on or after 1 August 1988:

```sql
SELECT PARCELID, FDATE
FROM FIRES
WHERE FDATE >= TO_DATE('01-AUG-1988', 'DD-MON-YYYY')
ORDER BY PARCELID, FDATE;
```

### 4. List the count of fires that occurred on or after 1 August 1988 by parcel:

```sql
SELECT PARCELID, COUNT(FDATE) FIRE_COUNT
FROM FIRES
WHERE FDATE >= TO_DATE('01-AUG-1988', 'DD-MON-YYYY')
GROUP BY PARCELID
ORDER BY PARCELID;
```

### 5. List the count of fires that occurred on or after 1 August 1988 by parcel for parcels that had more than one fire:

```sql
SELECT PARCELID, COUNT(FDATE) FIRE_COUNT
FROM FIRES
WHERE FDATE >= TO_DATE('01-AUG-1988', 'DD-MON-YYYY')
GROUP BY PARCELID
HAVING COUNT(FDATE) > 1
ORDER BY PARCELID;
```

Groups are shown in color, but this query does not actually perform grouping. Note that the fire at parcel 7 on 1 August 1987 has been excluded by the WHERE clause.

This query shows the result of grouping, but no HAVING clause is applied. Groups that satisfy the HAVING clause of Query 5 are shown in **bold**.

Final result, after groups that fail the HAVING clause have been eliminated.

<table>
<thead>
<tr>
<th>PARCELID</th>
<th>FDATE</th>
<th>FIRE_COUNT</th>
<th>PARCELID</th>
<th>FIRE_COUNT</th>
<th>PARCELID</th>
<th>FIRE_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>02-AUG-88</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>02-APR-89</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>26-JUL-89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>26-JUL-90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>02-JUL-89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Rules for GROUP BY Queries

- In a GROUP BY query, all expressions in the SELECT list not containing group functions (SUM, AVG, COUNT, etc.) must appear in the GROUP BY clause.

The following query is invalid because it includes a column (FDATE) that is not in the GROUP BY clause. This query will fail with the Oracle error "ORA-00979: not a GROUP BY expression":

```sql
SELECT PARCELID, FDATE, COUNT(FDATE) FIRE_COUNT
FROM FIRES
GROUP BY PARCELID
ORDER BY PARCELID, FDATE;
```

Think for a minute why this query does not make sense. For each distinct value of PARCELID, there may be multiple values of FDATE. For example, the parcel with PARCELID = 2 had fires on both 2 Aug. 1988 and 2 Apr. 1989. When we group by PARCELID alone, the results of the query will have at most one row for each value of PARCELID. If we include FDATE in the SELECT list, which FDATE should Oracle pick for PARCELID = 2? The answer is undefined, and that is why the query is invalid. Oracle is unable to pick a single value of an unaggregated item to represent a group.

To fix this query, we must ensure that the SELECT list and the GROUP BY clause contain the same expressions (excluding expressions that use group functions such as COUNT(FDATE)). We have two choices. First, we can remove FDATE from the SELECT list (and the ORDER BY clause):

```sql
SELECT PARCELID, COUNT(FDATE) FIRE_COUNT
FROM FIRES
GROUP BY PARCELID
ORDER BY PARCELID;
```

Second, we can add FDATE to the GROUP BY clause:
SELECT PARCELID, FDATE, COUNT(FDATE) FIRE_COUNT
FROM FIRES
GROUP BY PARCELID, FDATE
ORDER BY PARCELID, FDATE;

Be careful when picking this second option! Adding a column to the GROUP BY clause may change the meaning of your groups, as in this example. Notice that all the FIRE_COUNT values for this last query are 1. That's because by adding FDATE to the GROUP BY clause we have effectively made each group a single row from the FIRES table—not very interesting!

- All GROUP BY expressions should appear in the SELECT list.

(How else will you know what group is being shown?)

To find the names of the owners of exactly one parcel, we can use this query:

SELECT OWNERS.ONAME, COUNT(*) PARCEL_COUNT
FROM PARCELS, OWNERS
WHERE PARCELS.ONUM = OWNERS.OWNERNUM
GROUP BY OWNERS.ONAME
HAVING COUNT(*) = 1;

The query below is valid, but uninformative. Who are the single-parcel owners?

SELECT COUNT(*) PARCEL_COUNT
FROM PARCELS, OWNERS
WHERE PARCELS.ONUM = OWNERS.OWNERNUM
GROUP BY OWNERS.ONAME
HAVING COUNT(*) = 1;

Showing the count of parcels when we've restricted the parcel count to 1 is not very interesting. We can simply leave the count out of the SELECT list. Note that you can use a HAVING condition without including the group function in the SELECT list:

SELECT OWNERS.ONAME
FROM PARCELS, OWNERS
WHERE PARCELS.ONUM = OWNERS.OWNERNUM
GROUP BY OWNERS.ONAME
HAVING COUNT(*) = 1;
GROUP BY expressions can be more elaborate than simple column references.

Expressions such as

\[
\begin{align*}
&\text{COL1 + COL2} \quad \text{(the sum of two columns)} \\
&\text{COL1 || COL2 || COL3} \quad \text{(three columns concatenated together)}
\end{align*}
\]

are also valid in the GROUP BY clause. As noted above, the expression should appear both in the GROUP BY clause and in the SELECT list. A column alias, while valid in the SELECT list, may not be used in the GROUP BY clause. You must repeat the entire expression from the SELECT list in the GROUP BY clause. Note, however, that a column alias is valid in the ORDER BY clause.

Suppose we want to count the parcel owners based on the city and state they live in, formatting the city and state as "City, State" (e.g., "BOSTON, MA"). The query below is valid:

```
SELECT CITY || ', ' || STATE CITY_STATE,
       COUNT(*) OWNERS
FROM OWNERS
GROUP BY CITY || ', ' || STATE
ORDER BY CITY_STATE;
```

<table>
<thead>
<tr>
<th>CITY_STATE</th>
<th>OWNERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON, MA</td>
<td>8</td>
</tr>
<tr>
<td>BROOKLYN, NY</td>
<td>1</td>
</tr>
<tr>
<td>BURLINGTON, VT</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK, NY</td>
<td>1</td>
</tr>
</tbody>
</table>

The query below will fail with the Oracle error "ORA-00904: invalid column name" because "TOTAL_VAL" is merely a column alias, not a real column. (Curiously, column aliases are valid in the ORDER BY clause, as shown above.) Hence the actual expression must be included in the GROUP BY clause as above:

```
SELECT CITY || ', ' || STATE CITY_STATE,
       COUNT(*) OWNERS
FROM OWNERS
GROUP BY CITY || ', ' || STATE
ORDER BY CITY_STATE;
```
The HAVING clause restricts the groups that are returned.

Do not confuse it with the WHERE clause, which refers to the original rows before they are aggregated. While you can use the HAVING clause to screen out groups that the WHERE clause would have excluded, the WHERE is more efficient than the GROUP BY clause, because it operates while the rows are being retrieved and before they are aggregated, while the HAVING clause operates only after the rows have been retrieved and aggregated into groups.

The query below lists the count of fires by parcel, counting only fires with a loss of at least $40,000:

```sql
SELECT PARCELID, COUNT(FDATE) FIRE_COUNT
FROM FIRES
WHERE ESTLOSS >= 40000
GROUP BY PARCELID
ORDER BY PARCELID;
```

An incorrect way to attempt this query is to place the "ESTLOSS >= 40000" condition in the HAVING clause rather than the WHERE clause. The following query will fail with the Oracle error "ORA-00979: not a GROUP BY expression" because we are attempting to exclude a group using a column that is not part of the GROUP BY clause and hence out of context:

```sql
SELECT PARCELID, COUNT(FDATE) FIRE_COUNT
FROM FIRES
GROUP BY PARCELID
HAVING ESTLOSS >= 40000
ORDER BY PARCELID;
```

Suppose we want to look at losses for fires by ignition factor (IGNFACTOR), but want to ignore the case where IGNFACTOR is 2. We can write this query two ways.

First, we can use a WHERE clause:

```sql
SELECT IGNFACTOR, COUNT(FDATE) FIRE_COUNT,
SUM(ESTLOSS) LOSSES
FROM FIRES
GROUP BY IGNFACTOR;
```
FROM FIRES
WHERE IGNFACTOR <> 2
GROUP BY IGNFACTOR
ORDER BY IGNFACTOR;

Second, we can use a HAVING clause:

```
SELECT IGNFACTOR, COUNT(FDATE) FIRE_COUNT,
       SUM(ESTLOSS) LOSSES
FROM FIRES
GROUP BY IGNFACTOR
HAVING IGNFACTOR <> 2
ORDER BY IGNFACTOR;
```

Both of these queries return the same correct result, but the first version using WHERE is more efficient because it screens out rows when they are initially retrieved by the database engine, while the second version using HAVING takes effect only after all the rows have been retrieved and aggregated. This can make a big difference in performance in a large database.

- **Group functions ignore NULL values.**
  Remember, however, that COUNT(*) counts rows, not any particular column. Hence COUNT(*) is sometimes helpful when a particular column of interest contains NULLs. This can lead to different results if you are not careful. For example, the TAX table includes one row with no BLDVAL:

```
SELECT *
FROM TAX
WHERE BLDVAL IS NULL;
```

```
PARCELID  PRPTYPE  LANDVAL
---  ---  ----
   20  9
```

**N.B.:** To find rows with NULLs in them, you must use the syntax "expr IS NULL", as above. If you use "expr = NULL", the query will run but not return any rows. The other comparison operators (e.g., =, <>, ! =, >, <, >=, <=, LIKE) will never match NULL. For example, the query below executes but returns no rows:
SELECT *
FROM TAX
WHERE BLDVAL = NULL;

no rows selected
If we count records in the TAX table by row using COUNT(*), we get one result:

SELECT COUNT(*)
FROM TAX;

COUNT(*)
--------
9

but if we count the building values we get a different one:
SELECT COUNT(BLDVAL)
FROM TAX;

COUNT(BLDVAL)
--------
8

Values of ONUM in the PARCELS table are not unique. That explains why
COUNT(ONUM) and COUNT(DISTINCT ONUM) return different results in the
query below:

SELECT COUNT(ONUM) OWNERS,
       COUNT(DISTINCT ONUM) DISTINCT_OWNERS
FROM PARCELS;

OWNERS DISTINCT_OWNERS
-------- -----------
 20 11

In the last three queries, we were treating the entire set of rows as a single group
(i.e., we used a group function such as COUNT without a GROUP BY clause).
Now let's use a GROUP BY to see if ownership of properties in various land use
categories is concentrated among a few owners:

SELECT LANDUSE,
       COUNT(*) PARCELS,
       COUNT(ONUM) OWNERS,
       COUNT(DISTINCT ONUM) DISTINCT_OWNERS
FROM PARCELS
GROUP BY LANDUSE;
Which value do we want to use for the count of owners, OWNERS or DISTINCT OWNERS? Why?

**GROUP BY Examples**

Find the parcels that experienced more than one fire:

```sql
SELECT PARCELID, COUNT(FDATE) FIRE_COUNT
FROM FIRES
GROUP BY PARCELID
HAVING COUNT(FDATE) > 1
ORDER BY PARCELID;
```

List the fires with a loss of at least $40,000:

```sql
SELECT PARCELID, FDATE, ESTLOSS
FROM FIRES
WHERE ESTLOSS >= 40000
ORDER BY PARCELID, FDATE, ESTLOSS;
```

List the count of fires by parcel, counting only fires with a loss of at least $40,000:

```sql
SELECT PARCELID, COUNT(FDATE) FIRE_COUNT
FROM FIRES
WHERE ESTLOSS >= 40000
GROUP BY PARCELID
ORDER BY PARCELID;
```

Find the parcels that experienced more than one fire with a loss of at least $40,000:

```sql
SELECT PARCELID, COUNT(FDATE) FIRE_COUNT
FROM FIRES
WHERE ESTLOSS >= 40000
GROUP BY PARCELID
HAVING COUNT(FDATE) > 1
ORDER BY PARCELID;
```
Introducing the URISA PROCEEDINGS database:

- Database for URISA Proceedings papers
- Relational tables for Author, Title, Paper, Keyword relationships
- Handling multiple authors and multiple keywords
- Help pages and sample queries for the URISA database*

Sample SQL Queries using ArcView and the Cambridge SALES89 table

- **Example A**: Select address, date, realprice columns from sales89 table for houses that sold after July 1, 1989 for more than $250,000

  ```sql
  SELECT address, date, realprice
  FROM sales89
  WHERE realprice > 250000 and date > '07/01/1989'
  ```

- **Example B**: Count the number of 1989 sales associated with each address that is listed in the sales89 table and order the results by sale_count, then, address, and date:

  ```sql
  SELECT address, count(distinct date) sale_count
  FROM sales89
  WHERE realprice > 250000 and date > '07/01/1989'
  GROUP BY address, date
  ORDER BY count(distinct date), address, date
  ```

- **Example C**: For every sales89 sale in Cambridge, list the address, saledate, and sales price along with the percent of adults in the surrounding census block group who had less than a high school education. The sales89 and cambbgrp tables could be joined by a common column (if the sales89 table had a column listing the census block group) or by a spatial join (that used the geographic data to compute which block group contained each sale):

  ```sql
  SELECT s.address, s.date, s.realprice,
  100*(c.EDU1 + c.EDU2 / c.EDUTOTAL) low_ed_percent
  FROM cambbgrp c, sales89 s
  WHERE c.stcntrbg = s.stcntrbg
  ```

* Kindly refer to Lecture Notes Section
if the sales89 table included as a 'foreign key' the stcntrbg 'primary key' from the cambbgrp table, or:

```
SELECT s.address, s.date, s.realprice,  
100*(c.EDU1 + c.EDU2 / c.EDUTOTAL) low_ed_percent  
FROM cambbgrp c, sales89 s  
WHERE s.SpatialObject IS CONTAINED WITHIN  
c.SpatialObject
```