Relational Database Management & Geospatial Data

- **SQL and the Sample Parcel Database**
  - Examining the schema: table structure and data types
  - Reviewing software setup and using SQL*Plus
  - The basic SELECT statement
  - Sample SQL Queries using ArcView and the Cambridge SALES89 table
  - Qualities of a Good Database Design
  - Why use a more elaborate database management system (DBMS)?
- **Database Management: Motivation and Fundamentals (from 11.520 lecture notes)**
  - The Web as an information repository
  - Often need more highly structured data repositories (and query tools)
  - Planner's perspective and GIS implications
  - Data types, parsing, & mix-n-match issues
- **The Relational Model**
  - All data are represented as tables
  - Querying one or more tables via the basic SELECT statement
  - Gets interesting when "rows" in each table have different meaning
  - One-to-Many Complications (same-house sales; multi-polygon towns;...)

- **SQL and the Sample Parcel Database**[*]
  - Examining the schema: table structure and data types
    - Why spin off an owner's table
    - Why have an owner number
    - Entity-relationship diagrams and normalization
    - Database design issues and tradeoffs
  - Reviewing software setup and using SQL*Plus
    - discussion of lab questions

[*] Kindly refer to the Lecture Notes section

        - Use of NULL: difference between IS NULL and = NULL
        - Spooling files, fixed width fonts, use of COLUMN...
        - Table prefixes - why use table alias, distinguishing table owner
            - distributed access to table column
    - Use of SQL Notes* and Oracle help
- The basic SELECT statement to query one or more tables:

```
   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.

- **Sample SQL Queries using the Parcel Database***
    - See Lab 1* examples
    - See examples and other help in SQL Notes*

- **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 $250,00

      SELECT address, date, realprice
      FROM sales89
      WHERE realprice > 250000 and date > "07/01/1989"

* Kindly refer to Lecture Notes Section

- **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:

  ```
  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):

  ```
  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
  ```

  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
  ```

- **Qualities of a Good Database Design**
  - Tables reflect real-world structure of the problem
  - Can represent all expected data over time
  - Avoids redundant storage of data items
  - Provides efficient access to data
  - Supports the maintenance and integrity of data over time
  - Clean, consistent, and easy to understand

        - *Note: These objectives are sometimes contradictory!*

    - **Why use a more elaborate database management system (DBMS)?**
        - Handling multi-table complexity (one-to-many, ...)
        - Ease of documenting/replicating queries/results
        - Performance
        - Security
        - Safe for multiple users
        - Sharing data among applications
        - Built-in data dictionary
- **Database Management: Motivation and Fundamentals (Repeat of outline from end of previous lecture)**

    - The Web as an information repository
        - A rich information source but a loosely structured collection of relatively unstructured data
        - Hard to find what you want without search engines and portals to index and structure the information and standardize the query process
        - Hard to utilize and extend knowledge on the Web without controlling/copying it (broken links, complex parsing/extraction, limited quality control, etc.)
    - Often need more highly structured data repositories (and query tools)
        - Desktop tools such as Excel, MS-Access, Filemaker, etc. handle personal database management needs (mailing lists, survey results, etc.)
        - Complex software often needed to manage multi-user access to 'persistent data'
        - Types of databases: single-user, corporate, engineering, science, image/video, geographic, ...
        - Issues: performance, metadata, user interface, data structure, concurrency, distributed, ...
        - Other 'big-system' issues: Security/reliability/integrity requirements (parcel ownership records, census data, major roads)
        - Other complications: transaction processing, data warehousing, online analytic processing, data mining, ...
        - Our focus: data structure issues and query capabilities

- Planner's perspective and GIS implications
    - Complex, semi-structured questions that involve one-of-a-kind analyses:
        - Which buildings in Boston have more than 1000 square feet of retail space and are located in neighborhoods with above-average incomes?
        - What level of trace gas exposures can be anticipated from EPA's Toxic Release Inventory sites?
        - Have 'move-to-opportunity' families had better job-retention experience than inner city residents who receive job training and housing assistance?
    - Recognize that planner needs are different from those of City Hall
    - (corporate) vs. Professional (end-user) needs/goals
        - city hall (enterprise) issues -- efficient data entry/retrieval/accuracy/security using tools that facilitate automation, maintenance, access control, and simple interfaces for edits, reports, common queries
        - planning professional issues -- startup/flexibility/modeling/integration/power using tools that can extract, merge, transform data; handle time series; and support complex queries
    - Structured vs. unstructured databases
        - Highly structured data - Census data parcel records, etc. with SQL query tools
        - Unstructured data - Web pages with search engines and 'free-format text retrieval' tools
    - GIS 'demos' are easy but spatial analysis is hard
        - No sweat if the data you want are already cleaned, parsed, and precisely suited to your question
        - Useful spatial analyses involves judicious mixing and matching data from official and local sources
        - Tapping into distributed, non-static databases can get complex for non automatable tasks
- Data types, parsing, & mix-n-match issues
    - Alphanumeric: Character strings; integers, floating point numbers, dates, binary codes, ...
    - Multi-dimensional: Images, maps, spatial objects, 3D models, video, math models, ...

        - Encoding/parsing addresses, zips, census tracts (77 Mass Ave, Cambridge, MA 02139)
        - Storage space, column headers (metadata), null/missing values

- **The Relational Model**
    - All data are represented as tables
        - Each table can be stored or viewed as one 'flat file'
        - Tables are comprised of rows and columns
        - Simple queries select particular rows and columns from a table
        - Each table has a **primary key,** a unique identifier constructed from one or more columns
        - A table is linked (joined) to another by including the other table's primary key. Such an included column is called a **foreign key**
        - More complex queries relate (join) multiple tables using primary/foreign keys
        - The results of any given query are just another table! (so complex queries can involve sub-queries)
        - One-to-many (and many-to-many) relations can be handled through the use of aggregation functions (sum, count, average, minimum, etc.)
    - Gets interesting when "rows" in each table have different meaning and joining tables involves one-to-many or many-to-many matches. Consider:
        - house sales in a 'sales' table
        - persons in the owner table
        - tax payments in a 'tax' table
        - counts and other statistics in a census tract table
    - Handling one-to-many and many-to-many relations can be useful but tricky:
        - Owners may have multiple properties; properties may sell more than once; etc.
        - How can you join the tables in order to determine all owners that have been in arrears on their taxes within two years of buying a property
        - Are new owners more likely to be in arrears on their taxes if the property is in a low (high) income census tract?

Other SQL Commands and Notes (Mostly for later lectures & labs)

- **Review: SELECT Statement Syntax**
- **SQL Miscellany**
- **Creating a table: CREATE TABLE**
- **Dropping a table: DROP TABLE**
- **Storing a Query: CREATE VIEW**
- **Joins: Multiple Table Queries**
- **Aggregration: GROUP BY, Group Functions**

SQL Miscellany

- DESCRIBE
- Numbers and strings
- AND and OR
- LIKE
- SUBSTR
- Expressions

Creating a Table

- CREATE TABLE table_name ...
- CREATE TABLE table_name AS SELECT ...

Dropping a Table

- DROP TABLE table_name;

Storing a Query: CREATE VIEW

- CREATE VIEW view_name AS SELECT ...

Joins: Multiple Table Queries

- See: SQL Help Notes*

---

* 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:

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

Groups are shown in color, but this query does not actually perform grouping.

PARCELID	FDATE
2	02-AUG-88
2	02-APR-89
3	26-JUL-89
3	26-JUL-90
7	01-AUG-87
20	02-JUL-89

2. List the count of fires by parcel:

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

Groups and summary functions have been calculated; notice that no FDATE values are shown.

PARCELID	FIRE_COUNT
2	2
3	2
7	1
20	1

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:

```
 SELECT PARCELID,
FDATE
   FROM FIRES
  WHERE FDATE >=

TO_DATE('01-AUG-
1988',

'DD-MON-YYYY')
ORDER BY PARCELID,
FDATE;
```

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.

PARCELID	FDATE
2	02-AUG-88
2	02-APR-89
3	26-JUL-89
3	26-JUL-90
20	02-JUL-89

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

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

*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**.*

PARCELID	FIRE_COUNT
2	**2**
3	**2**
20	1

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

```
 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;
```

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

PARCELID	FIRE_COUNT
2	2
3	2

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":

  ```
   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):

```
  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

  ```
  COL1 + COL2                  (the sum of two columns)
  COL1 || COL2 || COL3 (three columns concatenated together)
  ```

  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;

  CITY_STATE                 OWNERS
  ------------------------ ----------
  BOSTON, MA                      8
  BROOKLYN, NY                    1
  BURLINGTON, VT                  1
  NEW YORK, NY                    1
  ```

  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:

```
       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:

```
  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:

```
    SELECT IGNFACTOR, COUNT(FDATE) FIRE_COUNT,
           SUM(ESTLOSS) LOSSES
```

```
         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
BLDVAL        TAX
---------- ---------- ---------- ---------
- ----------
        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;
```

```
LAN    PARCELS     OWNERS DISTINCT_OWNERS
--- ---------- ---------- ---------------
             2          2               1
A            4          4               3
C            5          5               3
CL           1          1               1
CM           1          1               1
E            2          2               2
R1           2          2               2
R2           1          1               1
R3           2          2               2

9 rows selected.
```

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:

```
  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:

```
  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:

```
  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:

```
  SELECT PARCELID, COUNT(FDATE) FIRE_COUNT
    FROM FIRES
   WHERE ESTLOSS >= 40000
GROUP BY PARCELID
  HAVING COUNT(FDATE) > 1
ORDER BY PARCELID;
```