The Zoning Variance Database & More
Advanced
SQL Query Construction Techniques

Outline

- The Zoning Variance Database
- More About Queries (SELECT Statements)
  - Review: SELECT Statement Syntax
  - Ordering Rows Returned by a Query
  - Order of Operations with 'AND' and 'OR'
  - SELECT DISTINCT
  - Cartesian Products
  - Null Values
  - Outer Joins
  - Inner vs. Outer Joins with GROUP BY
- Views and Other Table-Like Database Objects
- The Data Dictionary
- Data Manipulation Language
  - INSERT: Add Rows to a Table
  - DELETE: Delete Rows from a Table
  - UPDATE: Modify Data in a Table
- Transaction Control
- Data Definition Language
  - Create and Drop Objects
  - Access Privileges for Objects
- SQL Query Construction Techniques
  - Method 1: Creating Intermediate Tables
  - Method 2: Use a View Instead of a Table
  - Method 3: Use a Subquery
  - Method 4: Use a More Efficient Subquery
The Zoning Variance Database

<table>
<thead>
<tr>
<th>Zoning Variances*</th>
<th>Schema of ZONING table (and listing of related lookup tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980 Census data (by Boston NSA)*</td>
<td>Schema of 1980 Boston Census data (and related lookup tables)</td>
</tr>
<tr>
<td>Schema of Decision, Use, NSA, Neighbrhd Lookup Tables*</td>
<td>Schema of Lookup tables (second half of Census data web page)</td>
</tr>
<tr>
<td>Sub-Neighborhood lookup table*</td>
<td>The NSA and NEIGHBRHD tables (bottom of Zoning Variance web page)</td>
</tr>
<tr>
<td>SQL examples using zoning variances*</td>
<td>Annotated SQL queries of ZONING table</td>
</tr>
<tr>
<td>Grouping zoning applicants via 'lookup' tables*</td>
<td>Annotated SQL queries illustrating use of lookup tables to categorize ownership of properties seeking zoning variances</td>
</tr>
<tr>
<td>Zoning Variance Database Evolution Chart*</td>
<td>Stages of evolution of the ZONING variance database</td>
</tr>
</tbody>
</table>

More About Queries (SELECT Statements)

Review: SELECT Statement Syntax

Basic Syntax:

```
SELECT expr1, expr2, expr3, ...
    FROM object1, object2, ...
    WHERE conditions
GROUP BY expr4, expr5, expr6, ...
    HAVING conditions
ORDER BY expr7, expr8, expr9, ...
```

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
Ordering Rows Returned by a Query

First of all, almost every SQL statement you write should specify the way the rows will be sorted. That means you should include an ORDER BY clause in nearly every SQL SELECT statement. While examples are in the notes, we haven't emphasized how to sort in descending order. For this you can use the 'DESC' keyword after the expression you want to sort that way. (SQL also has an 'ASC' keyword for ascending order. Since it is the default, it can be omitted.)

The syntax looks like this:

```sql
SELECT ... 
ORDER BY expr1 [ [ ASC | DESC ] , expr2 [ ASC | DESC ] ... ]
```

For example, let's sort the parcels by land use in ascending order and the square footage in descending order:

```sql
SELECT parcelid, landuse, sqft 
FROM parcels
ORDER BY landuse ASC, sqft DESC;
```

<table>
<thead>
<tr>
<th>PARCELID</th>
<th>LAN</th>
<th>SQFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td>34800</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>10900</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>14000</td>
</tr>
<tr>
<td>14</td>
<td>A</td>
<td>10000</td>
</tr>
<tr>
<td>16</td>
<td>A</td>
<td>9600</td>
</tr>
<tr>
<td>18</td>
<td>A</td>
<td>9500</td>
</tr>
<tr>
<td>11</td>
<td>C</td>
<td>210000</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>100000</td>
</tr>
<tr>
<td>19</td>
<td>C</td>
<td>40000</td>
</tr>
<tr>
<td>8</td>
<td>C</td>
<td>24800</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>20000</td>
</tr>
<tr>
<td>4</td>
<td>CL</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CM</td>
<td>2100</td>
</tr>
<tr>
<td>20</td>
<td>E</td>
<td>50000</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>25000</td>
</tr>
<tr>
<td>9</td>
<td>R1</td>
<td>1800</td>
</tr>
<tr>
<td>17</td>
<td>R1</td>
<td>1500</td>
</tr>
<tr>
<td>13</td>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>R3</td>
<td>5500</td>
</tr>
<tr>
<td>12</td>
<td>R3</td>
<td>5300</td>
</tr>
</tbody>
</table>
Notice that the query can mix the ASC and DESC keywords in a single ORDER BY clause.

**Order of Operations with 'AND' and 'OR'**

The Boolean operators 'AND' and 'OR' can do unexpected things when you combine them in a query and you're not careful. Suppose we want to find parcels that have a square footage of 5300 or 10000, and, of those, find the ones with land use code 'A'.

We might (incorrectly) write a query like this:

```
SELECT parcelid, landuse, sqft
FROM parcels
WHERE sqft = 5300
  OR sqft = 10000
  AND landuse = 'A'
ORDER BY landuse ASC, sqft DESC;
```

<table>
<thead>
<tr>
<th>PARCELID</th>
<th>LAN</th>
<th>SQFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>A</td>
<td>10000</td>
</tr>
<tr>
<td>12</td>
<td>R3</td>
<td>5300</td>
</tr>
</tbody>
</table>

Notice that it returned a row with LANDUSE = 'R3'. That's not what we wanted! The problem is that the 'AND' and 'OR' operators, when mixed, are not processed in the sequence written, but rather follow an order of operations much as in algebra (exponentation before everything, then multiplication and division before addition and subtraction). In Boolean logic, 'AND' is like multiplication and 'OR' is like addition, and Oracle orders their processing accordingly. Hence, the query above is actually equivalent to this one:

```
SELECT parcelid, landuse, sqft
FROM parcels
WHERE sqft = 5300
  OR ( sqft = 10000
       AND landuse = 'A')
ORDER BY landuse ASC, sqft DESC;
```

<table>
<thead>
<tr>
<th>PARCELID</th>
<th>LAN</th>
<th>SQFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>A</td>
<td>10000</td>
</tr>
<tr>
<td>12</td>
<td>R3</td>
<td>5300</td>
</tr>
</tbody>
</table>
Since the order of operations can surprise you at inconvenient times, you should *always* use parentheses to force the correct order whenever you mix 'AND' and 'OR' in a WHERE clause. Here is the correct way to write the query:

```sql
SELECT parcelid, landuse, sqft
FROM parcels
WHERE ( SQFT = 10000
    OR SQFT = 5300)
    AND LANDUSE = 'A'
ORDER BY landuse ASC, sqft DESC;
```

```
PARCELID LAN SQFT
---------- --- -----------
   14    A    10000
```

**SELECT DISTINCT**

Normally, a query may return duplicate rows. For example, if we query the FIRES table to list the parcels that had fires, we'll find that the parcels that had more than one fire (parcels 2 and 3) show up multiple times:

```sql
SELECT parcelid
FROM fires
ORDER BY parcelid;
```

```
PARCELID
--------
   2
   2
   3
   3
   7
  20
```

6 rows selected.

If we don't want to see the duplicates, we can add the keyword DISTINCT right after SELECT:

```sql
SELECT DISTINCT parcelid
FROM fires
ORDER BY parcelid;
```

```
PARCELID
--------
   2
   3
   7
  20
```

4 rows selected.
Now parcels 2 and 3 show up only once. You only use the DISTINCT once, right after SELECT, to apply to the entire row. You do not apply it to each column. Hence, this query is valid:

```
SELECT DISTINCT p.onum, o.oname
FROM parcels p, owners o
WHERE p.onum = o.ownernum
ORDER BY p.onum;
```

```
ONUM ONAME
-------- ---------------
  9  PATRICK KING
 10   METHUINON TRUST
 11  FERNANDO MARTINEZ
 18  JOHN MCCORMACK
 29   FRANK O'BRIEN
 32  GERALD RAPPAPORT
 38   BAY STATE, INC.
 55  THOMAS KELLIHER
 89  JOSEPH NOONAN
100   MGH, INC.
200  VANDELAY INDUSTRIES
```

11 rows selected.

However, the following incorrect query, with two DISTINCT keywords, generates the cryptic error message "ORA-00936: missing expression":

```
SELECT DISTINCT p.onum, DISTINCT o.oname
FROM parcels p, owners o
WHERE p.onum = o.ownernum
ORDER BY p.onum;
```

Note also that you can use DISTINCT with the group functions (e.g., COUNT, AVG, STDDEV) to get these functions to consider only distinct values within a group:

```
SELECT COUNT(wpb) wpb_ct,
       COUNT(DISTINCT wpb) wpb_ct_distinct,
       COUNT(*) row_ct
FROM parcels;
```

```
WPB_CT WPB_CT_DISTINCT ROW_CT
------- --------------- ------
  20         20        20
```

1 row selected.

Finally, note that COUNT(DISTINCT *) does not work. This makes sense if you think about it. Why do you think this is the case?

### Cartesian Products

What's wrong with this query?

```
SELECT p.parcelid, p.onum, o.ownernum, o.oname
FROM parcels p, owners o;
```

This query returns 200 rows!
There are only 20 rows in PARCEL and 10 rows in OWNERS, so what's going on? The problem here is that this query has no WHERE clause that specifies how the PARCEL table relates to the OWNERS table. Without this information, Oracle does not know how to match a row in PARCEL to a corresponding row in OWNERS. What does it do instead? It matches every row in PARCEL to every row in OWNERS. Hence, we end up with:

(20 rows in PARCEL) matched to (10 rows in OWNERS) = 20 x 10 = 200 rows returned

This kind of unconstrained join is called a Cartesian product. This result is desirable only under rare circumstances. If you have queries that are returning a suspiciously large number of rows, you have probably unwittingly requested a Cartesian product. Note that for tables of even modest size, the number of rows returned by a Cartesian product can be explosive. If you generate a Cartesian product of one table with 1,000 rows with another table with 2,000 rows, your query will return 1,000 x 2,000 = 2,000,000 rows! That's right--two million rows! Hence, you should very careful to avoid unintentional Cartesian products.

To fix this query, we need to specify how the owner numbers stored in the PARCEL table should be matched to owner numbers in the OWNERS table. In PARCEL, the owner numbers are stored in ONUM, while in OWNERS they are stored in OWNERNUM. Here is the repaired query with the appropriate join condition in the WHERE clause:

```
SELECT p.parcelid, p.onum, o.ownernum, o.oname
FROM parcels p, owners o
WHERE p.onum = o.ownernum;
```

This query returns 20 rows, which is definitely an improvement.

Note that a Cartesian product can easily be hidden in a query that requires multiple joins. Suppose we want to find all the papers, with associated keywords, written by authors with the last name WALKER. We could try this query:

```
COLUMN keyword FORMAT A20 TRUNC
COLUMN title FORMAT A25 TRUNC

SELECT a.lastname, a.fnamemi, k.keyword, t.title
FROM keywords k, match m, titles t, authors a
WHERE m.code = k.code
    AND t.paper = a.paper
    AND a.lastname = 'WALKER'
ORDER BY a.lastname, a.fnamemi, k.keyword, t.title;
```
This query returns a whopping 6174 rows! What's wrong? All of the tables appear to be involved in a join condition. The problem is that while the MATCH and KEYWORDS tables are tied together, and the TITLES and AUTHORS tables are linked to each other, nothing links these two sets of tables together. Adding another join condition fixes it:

```
SELECT a.lastname, a.fnamemi, k.keyword, t.title
FROM keywords k, match m, titles t, authors a
WHERE m.code = k.code
  AND t.paper = a.paper
  AND m.paper = t.paper
  AND a.lastname = 'WALKER'
ORDER BY a.lastname, a.fnamemi, k.keyword, t.title;
```

This query returns a much more reasonable 14 rows. Note that we could have also specified "m.paper = a.paper" and it would have worked too. Why?

A Cartesian product can easily be hidden by a GROUP BY, since it will aggregate all the spurious rows, and you will not see the rows that made up the groups. Here's a variation on the earlier, broken example, now with a GROUP BY:

```
SELECT a.lastname, a.fnamemi, count(k.keyword) keywords, count(t.title) titles
FROM keywords k, match m, titles t, authors a
WHERE m.code = k.code
  AND t.paper = a.paper
  AND a.lastname = 'WALKER'
GROUP BY a.lastname, a.fnamemi
ORDER BY a.lastname, a.fnamemi;
```

Because this query returns only counts, it's not obvious that the query is defective--unless you have an idea about what results are reasonable! Always scrutinize your results!

**NULL Values**

NULL is a special value that means "missing" or "unknown." It can be placed in a column of any type, unless the column has been declared NOT NULL, in which case NULL values are forbidden. A NULL is not the same as zero or any other value.

Special rules apply when NULL values are involved:

- A row containing NULL value in a column will never match another row in a join, not even another one containing NULL.
• Remember that in logical conditions (e.g., col1 = col2, col3 > col4, col5 < 0), **NULL does not equal NULL**: logical expressions containing NULL will always evaluate to **UNKNOWN**, which is similar (but not identical) to FALSE.

• Never use NULL with one of the Boolean operators (col1 = NULL, col2 <> NULL, col3 < NULL, col4 <= NULL, col5 > NULL, col6 >= NULL). All of these will probably not perform as intended. Always use the IS operator when testing for NULL (col7 IS NULL, col8 IS NOT NULL).

• Group functions on a column **ignore** NULL values.

For more information on NULL values and how Oracle treats them, consult the Oracle documentation on NULL.

**Outer Joins**

Take a look at this query:

```sql
SELECT p.parcelid, f.fdate
FROM parcels p, fires f
WHERE p.parcelid = f.parcelid
ORDER BY p.parcelid, f.fdate;
```

The query above returned 6 rows, but there are 20 parcels. Fourteen parcels seem to be missing. Where did they go? The answer is that Oracle will only list the parcels that occur in both tables, PARCELS and FIRES.

How do we get around this problem if we want to see all 20 rows in the parcel table, whether they match a record in FIRES or not? The answer is an **outer join**. The standard join is also known as an "inner join," meaning that the default behavior of not matching NULL values occurs. In an outer join, we explicitly tell Oracle that we want it to display NULL values that would otherwise be excluded by adding the (+) outer join operator to each column in the WHERE clause where the additional NULLs should appear.

```sql
SELECT p.parcelid, f.fdate
FROM parcels p, fires f
WHERE p.parcelid = f.parcelid (+)
ORDER BY p.parcelid, f.fdate;
```

Note that the position of the outer join operator (+) is significant! This query will run but return a different result:

```sql
SELECT p.parcelid, f.fdate
FROM parcels p, fires f
WHERE p.parcelid (+) = f.parcelid
ORDER BY p.parcelid, f.fdate;
```
Some additional class notes on outer joins* are available. If you wish, you can peruse the Oracle 8i documentation on outer joins.

**Inner vs. Outer Joins with GROUP BY**

List all the parcels that had a fire, including the address and date of the fire (this requires an inner join):

```sql
SELECT p.parcelid, p.add1, p.add2, f.fdate
FROM parcels p, fires f
WHERE p.parcelid = f.parcelid
ORDER BY p.parcelid, f.fdate;
```

Repeat the same query, except list all the parcels, whether they had a fire or not (this requires an outer join):

```sql
SELECT p.parcelid, p.add1, p.add2, f.fdate
FROM parcels p, fires f
WHERE p.parcelid = f.parcelid (+)
ORDER BY p.parcelid, f.fdate;
```

List the count of fires for the parcels that had a fire (this requires an inner join with grouping):

```sql
SELECT p.parcelid, COUNT(fdate) fire_count
FROM parcels p, fires f
WHERE p.parcelid = f.parcelid
GROUP BY p.PARCELID
ORDER BY COUNT(fdate) DESC, p.parcelid;
```

Note that the query above lists the parcels in descending order of the count of fires. When specifying a group function or other expression in the ORDER BY clause, you must use the full expression, even if you defined a column alias for it in the SELECT list. In this case, we must use COUNT(FDATE) rather than FIRE_COUNT in the ORDER BY clause. The DESC keyword after COUNT(FDATE) indicates that we want the fire counts shown in descending order, rather than the default ascending order. You need to apply the DESC keyword to every expression in the ORDER BY clause that you want in descending order.

List the count of fires for all parcels, whether it experienced a fire or not (this requires an outer join with grouping):

```sql
SELECT p.parcelid, COUNT(fdate) fire_count
FROM parcels p, fires f
WHERE p.parcelid = f.parcelid (+)
GROUP BY p.parcelid
ORDER BY COUNT(fdate) DESC, p.parcelid;
```

* Kindly refer to Lecture Notes section
Note that the query above uses an **outer join** on the FIRES table -- indicated by the outer join symbol (+) -- to include the parcels that are not listed in the FIRES table in the count. The outer join symbol indicates where a NULL should replace a real value if none is available in the table. Note that we need to use the outer join symbol with both columns in FIRES that we are joining with PARCEL.

The query below runs but returns the _wrong_ result--some parcels that had no fires show up with one fire in the count. Why?

```
SELECT p.parcelid, COUNT(*) bogus_fire_count
FROM parcels p, fires f
WHERE p.parcelid = f.parcelid (+)
GROUP BY p.parcelid
ORDER BY COUNT(*) DESC, p.parcelid;
```

**Views and Other Table-Like Database Objects**

- **Views**
- **Synonyms**

A synonym is simply a second name for an existing object. These are particular convenient when a table is owned by another user (or, stated differently, stored in a different schema). You have been using synonyms all along for the objects in the PARCELS, URISA, and ZONING databases. To see the synonyms, use this query:

```
SELECT synonym_name, table_owner, table_name
FROM user_synonyms
ORDER BY synonym_name;
```

- **Others (use this link to find a discussion of most of the Oracle 8i schema objects)**

*Views* are the most useful of these alternate object types for our purposes. A view is a query that is stored in the database, then treated like a table. Unlike a table you create using CREATE TABLE ... AS SELECT, which creates a one-time snapshot of the data returned by the query, a view will reflect the current state of the tables in the underlying query. Hence, if the tables in the database are changing over time, the same query on a view may return different results at different times. Creating a view is similar to the CREATE TABLE ... AS SELECT statement; instead, use CREATE VIEW ... AS SELECT. For example:

```
CREATE VIEW parcel_owners
```
AS SELECT p.parcelid, o.oname, P.ADD1, P.ADD2, P.ZIP
FROM parcels p, owners o
WHERE p.onum = o.ownernum(+);

CREATE VIEW owner_sqft
AS SELECT o.oname, SUM(SQFT) TOTAL_SQFT
FROM parcels p, owners o
WHERE p.onum = o.ownernum
GROUP BY o.oname;

Note that the column alias TOTAL_SQFT in the example above is required because Oracle needs to know what to name the column in the view. Do not include an ORDER BY clause in the SELECT statement that you use to create the view.

Once the view is created, it can be treated for (almost) all intents and purposes as a true table. You can describe them to see their structure:

DESCRIBE parcel_owners
DESCRIBE owner_sqft

The Data Dictionary

Information describing all the Oracle objects in the database is stored in the Oracle data dictionary, which you can access through a large number of data dictionary views. We can query data dictionary views just like any view or table. You're already familiar with the view CAT which provides you with a catalog of objects that you own:

```
SELECT * FROM cat;
```

Many other such views are available. The USER_SYNONYMS view mentioned above is one of them. The data dictionary view USER_VIEWS includes information about the views we just created. To see its structure, we can use the DESCRIBE statement in SQL*Plus:

```
SQL> DESCRIBE user_views
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIEW_NAME</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>VARCHAR2(30)</td>
<td></td>
</tr>
<tr>
<td>TEXT_LENGTH</td>
<td></td>
</tr>
<tr>
<td>NUMBER</td>
<td></td>
</tr>
<tr>
<td>TEXT</td>
<td></td>
</tr>
<tr>
<td>LONG</td>
<td></td>
</tr>
</tbody>
</table>
To see the definition of the views we just created, we can use the following statements:

```sql
SET LONG 5000
SELECT view_name, text
FROM user_views
WHERE view_name IN ('PARCEL_OWNERS', 'OWNER_SQFT');
```

Note that the column TEXT has type "LONG". In order to ensure that SQL*Plus displays this LONG column properly, we used the "SET LONG 5000" statement before running the query on USER_VIEWS.

**Data Manipulation Language**

SELECT statements view or query the contents of tables. With Data Manipulation Language (DML) statements, we can alter the contents of the tables. DML statements include:

- **INSERT**
- **DELETE**
- **UPDATE**

**INSERT: Add Rows to a Table**

General syntax:

```sql
INSERT INTO table1 (col1, col2, ...) VALUES (value1, value2, ...)
```

or

```sql
INSERT INTO table1 (col1, col2, ...) VALUES (value1, value2, ...)
```
SELECT ...

Example: Add a row to the FIRES table

```
INSERT
    INTO fires (parcelid, fdate, ignfactor, estloss)
VALUES (12, '17-JAN-96', 2, 35000)
```

**DELETE: Delete Rows from a Table**

General syntax:
```
DELETE
    FROM table1
    WHERE conditions;
```
Example: Delete fires with losses less than $50000 from the database
```
DELETE
    FROM fires
    WHERE estloss < 50000;
```

**UPDATE: Modify Data in a Table**

General syntax:
```
UPDATE table1
    SET col1 = value1, col2 = value2, ...
    WHERE conditions;
```

or
```
UPDATE table1
    SET col1 =
        (SELECT ...)
    WHERE conditions;
```

Example: Change the building value of a particular parcel
```
UPDATE tax
    SET bldval = 200000
    WHERE parcelid = 11;
```
Note: You can also update a table using a subquery. This typically involves one or more correlated subqueries (see these examples). Correlated subqueries are beyond this overview.
Transaction Control

Transaction control statements allow several SQL statements to be grouped together into a unit (a transaction) that are either processed or rejected as a whole. These statements include:

- **COMMIT**: makes permanent all changes since the start of the session or the previous COMMIT
- **ROLLBACK**: reverses pending changes to the database
- **SAVEPOINT**: allows more refined control over COMMITs and ROLLBACKs
- **SET TRANSACTION**: allows more refined control over transaction progress

Data Definition Language

Data Definition Language (DDL) statements affect the structure and security provisions of the database, among other things.

Create and Drop Objects

Create new objects

```sql
CREATE TABLE table1 ... ;
CREATE VIEW view1 ... ;
CREATE_INDEX index1 ON table1 (col1, ... ) ;
```

Drop objects permanently

```sql
DROP TABLE table1;
DROP VIEW view1;
DROP_INDEX index1;
```

Modify existing objects

```sql
ALTER TABLE table1;
ALTER VIEW view1;
ALTER_INDEX index1;
```
Access Privileges for Objects

Specific privileges on tables can be given to individual users

```sql
GRANT SELECT
    ON table1
    TO user1;

GRANT SELECT, INSERT, UPDATE (col1)
    ON table1
    TO user1
    WITH GRANT OPTION;
```

Privileges can be revoked

```sql
REVOKE ALL
    ON table1
    FROM user1;
```

SQL Query Construction Techniques:

Building the Answer to a Complex Question

Method 1: Creating Intermediate Tables

In previous years, the introductory exercises encouraged creating temporary tables as shown here. We now deprecate that practice in favor of creating views, especially when trying to build a simple query into a more complex one. In some cases, however, creating your own tables makes sense. One example is the solution to the problems with the property owners' names in the real-world parcel database.

Adapted from Joe's URISA database examples*:

```sql
/* Find papers using keywords related to GIS and mapping: */
DROP TABLE gispapers;
DROP VIEW gispapers;

CREATE TABLE gispapers AS
    SELECT m.code, keyword, m.paper
```

* Kindly refer to Lecture Notes section
FROM keywords k, match m
WHERE m.code = k.code
    AND ( keyword LIKE '%GIS%'
    OR keyword LIKE '%GEOGRAPHIC INFORMATION%'
    OR keyword LIKE '%MAPPING%');

/* Counts of papers using these keywords */
SELECT m.code, k.keyword, count(distinct t.paper) papers
    FROM match m, titles t, keywords k
    WHERE m.paper = t.paper AND k.code = m.code
    AND m.code IN
        (SELECT code
            FROM gispapers)
GROUP BY m.code, k.keyword
ORDER BY m.code;

Method 2: Use a View Instead of a Table

We can create a view instead of a table using the identical SELECT statement, but substituting 'CREATE VIEW' for 'CREATE TABLE' in the example above. We have to drop the table first because a table and a view may not have the same name.

/* Find papers using keywords related to GIS and mapping: */
DROP TABLE gispapers;

CREATE VIEW gispapers
AS SELECT m.code, keyword, m.paper
    FROM keywords k, match m
    WHERE m.code = k.code
    AND ( keyword LIKE '%GIS%'
    OR keyword LIKE '%GEOGRAPHIC INFORMATION%'
    OR keyword LIKE '%MAPPING%');

/* Counts of papers using these keywords */
SELECT m.code, k.keyword, count(distinct t.paper) papers
    FROM match m, titles t, keywords
WHERE m.paper = t.paper AND k.code = m.code
AND m.code IN
(SELECT code
  FROM gispapers)
GROUP BY m.code, k.keyword
ORDER BY m.code;

Note that we can query from the view implementation of GISPAPERS the same as from
the table implementation. A view, however, is simply a stored query (which retains its
ties to the original table), while the table copies the data (and possibly wastes a lot of
space in the database). We can look at the text of the view with the following query
against the Oracle data dictionary:
-- Use SET LONG 5000 so that SQL*Plus will
display enough characters
-- from the view definition column for us to see
the entire query.

SET LONG 5000
SELECT text
FROM user_views
WHERE view_name = 'GISPAPERS';

Note that the name of the view must be in UPPERCASE letters and surrounded by 'single
quotation marks'.

Method 3: Use a Subquery

A view is just a stored query. We can embed this query in our original SQL statement so
that we can accomplish the entire task with one statement. The subquery below is the
same as the query in the CREATE TABLE and CREATE VIEW statements, except that
the columns keyword and m.paper have been removed because they are not needed in the
subquery. (If you included them, you would see the Oracle error "ORA-00913: too many
values".)

SELECT m.code, k.keyword, count(distinct
  t.paper) papers
FROM match m, titles t, keywords k
WHERE m.paper = t.paper AND k.code = m.code
AND m.code IN
(SELECT m.code
  FROM keywords k, match m
  WHERE m.code = k.code
  AND (keyword LIKE '%GIS%' OR keyword LIKE
  '%GEOGRAPHIC INFORMATION%'
    OR keyword LIKE '%MAPPING%'


Method 4: Use a More Efficient Subquery

The query above works, but is not as fast as it could be because Oracle may be returning many more rows in the subquery than needed. We can rewrite this query using the more efficient but less obvious EXISTS syntax.

```
SELECT m.code, k1.keyword, count(distinct t.paper) papers
FROM match m, titles t, keywords k1
WHERE m.paper = t.paper
AND k1.code = m.code
AND EXISTS
    (SELECT NULL
     FROM keywords k2
     WHERE m.code = k2.code
     AND (    keyword LIKE '%GIS%' 
     OR keyword LIKE '%GEOGRAPHIC INFORMATION%'
     OR keyword LIKE '%MAPPING%'))
GROUP BY m.code, k1.keyword
ORDER BY m.code;
```

Note that we use NULL in the subquery's SELECT list to indicate that we do not care about the contents returned by the subquery, but only if the subquery returns a row or not each time it is executed (once for each candidate row in the outer query).