PostGIS in Action, Second Edition
by Regina O. Obe
Leo S. Hsu

Appendix C
# brief contents

## Part 1 Introduction to PostGIS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>What is a spatial database?</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Spatial data types</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>Spatial reference system considerations</td>
<td>60</td>
</tr>
<tr>
<td>4</td>
<td>Working with real data</td>
<td>82</td>
</tr>
<tr>
<td>5</td>
<td>Using PostGIS on the desktop</td>
<td>113</td>
</tr>
<tr>
<td>6</td>
<td>Geometry and geography functions</td>
<td>140</td>
</tr>
<tr>
<td>7</td>
<td>Raster functions</td>
<td>175</td>
</tr>
<tr>
<td>8</td>
<td>PostGIS TIGER geocoder</td>
<td>203</td>
</tr>
<tr>
<td>9</td>
<td>Geometry relationships</td>
<td>218</td>
</tr>
</tbody>
</table>

## Part 2 Putting PostGIS to Work

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Proximity analysis</td>
<td>247</td>
</tr>
<tr>
<td>11</td>
<td>Geometry and geography processing</td>
<td>261</td>
</tr>
<tr>
<td>12</td>
<td>Raster processing</td>
<td>287</td>
</tr>
<tr>
<td>13</td>
<td>Building and using topologies</td>
<td>309</td>
</tr>
<tr>
<td>14</td>
<td>Organizing spatial data</td>
<td>337</td>
</tr>
<tr>
<td>15</td>
<td>Query performance tuning</td>
<td>369</td>
</tr>
</tbody>
</table>
PART 3 USING POSTGIS WITH OTHER TOOLS .........................407

16  ■ Extending PostGIS with pgRouting and procedural
    languages       409
17  ■ Using PostGIS in web applications     442
PostgreSQL supports almost the whole ANSI SQL 92, 1999 standard logic, as well as many of the SQL:2003, SQL:2006, SQL:2008 constructs, and some of the SQL:2011 constructs. In this appendix, we’ll cover some of these as well as some PostgreSQL-specific SQL language extensions. Because we’ll remain fairly focused on standard functionality, the content in this appendix is also applicable to other standards-compliant relational databases.

C.1 information_schema

The information_schema is a catalog introduced in SQL 92 and enhanced in each subsequent version of the spec. Although it’s a standard, sadly most commercial and open source databases don’t completely support it. We know that the following common databases do: PostgreSQL (7.3+), MySQL 5+, and Microsoft SQL Server 2000+. Oracle and IBM do via user-supported contributions.

The most useful views in this schema are tables, columns, and views; they provide a catalog of all the tables, columns, and views in your database. To get a list of all non-system tables in PostgreSQL, you can run the following query (the information_schema.tables view in PostgreSQL will list only tables that you have access to):

```
SELECT table_schema, table_name, table_type
FROM information_schema.tables
WHERE table_schema NOT IN('pg_catalog', 'information_schema')
ORDER BY table_schema, table_name;
```

The preceding query will work equally well in MySQL (except that in MySQL schema means database and there’s only one information_schema shared across all MySQL databases in a MySQL cluster). MS SQL Server behaves more like PostgreSQL in that each information_schema is unique to each database, except that in SQL Server the system views and tables aren’t queryable from information_schema, whereas they are in PostgreSQL.

The columns view will give you a listing of all the columns in a particular table or set of tables. The following example lists all the columns found in a table called ch01.restaurants.
### Listing C.1  List all columns in ch01.restaurants

```
SELECT c.column_name, c.data_type, c.udt_name,
       c.ordinal_position AS ord_pos,
       c.column_default AS cdefault
FROM information_schema.columns AS c
WHERE table_schema = 'ch01' and table_name = 'restaurants'
ORDER BY c.column_name;
```

The results of this query look something like the following.

### Listing C.2  Results of query in listing C.1

<table>
<thead>
<tr>
<th>column_name</th>
<th>data_type</th>
<th>udt_name</th>
<th>ord_pos</th>
<th>cdefault</th>
</tr>
</thead>
<tbody>
<tr>
<td>franchise</td>
<td>character</td>
<td>bpchar</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>geom</td>
<td>USER-DEFINED</td>
<td>geometry</td>
<td>3</td>
<td>nextval('ch01...')</td>
</tr>
<tr>
<td>id</td>
<td>integer</td>
<td>int4</td>
<td>1</td>
<td>nextval('ch01.restaurants_gid_seq'::regclass)</td>
</tr>
</tbody>
</table>

One important way that PostgreSQL is different from databases such as SQL Server and MySQL Server that support `information_schema` is that it has an additional field called `udt_name` that denotes the PostgreSQL-specific data type. Because PostGIS geometry is an add-on module and not part of PostgreSQL, you’ll see the standard ANSI data type listed as `USER-DEFINED` and the `udt_name` field storing the fact that it’s a geometry.

The `information_schema.columns` view provides numerous other fields, so we encourage you to explore it. We consider these to be the most useful fields:

- **table_name** and **column_name**—These should be obvious.
- **data_type**—The ANSI standard data type name for this column.
- **udt_name**—The PostgreSQL-specific name. Except for user-defined types, you can use `data_type` or `udt_name` when creating these fields except in the case of series. Recall that we created `id` as a serial data type in chapter 1, and behind the scenes PostgreSQL created an integer column and a sequence object and set the default of this new column to the next value of the sequence object: `nextval('ch01.restaurants_gid_seq'::regclass)`.
- **ordinal_position**—This is the order in which the column appears in the table.
- **character_maximum_length**—With character fields, this tells you the maximum number of characters allowed in the field.
- **column_default**—The default value assigned to new records. This can be a constant or the result of a function.

The `tables` view lists both tables and views (virtual tables). The `views` view gives you the name and the view definition for each view you have access to. The view definition gives you the SQL that defines the view, and it’s very useful for scripting the definitions. In PostgreSQL, you can see how the `information_schema` views are defined,
though you may not be able to in other databases such as SQL Server, because the information_schema is excluded from this system view.

```sql
SELECT table_schema, table_name, view_definition,
    is_updatable, is_insertable_into
FROM information_schema.views
WHERE table_schema = 'information_schema';
```

In the preceding examples, we’ve demonstrated the common meta-tables you’d find in the ANSI information_schema. We also demonstrated the most fundamental of SQL statements. In the next section, we’ll look at the anatomy of an SQL statement and describe what each part means.

## C.2 Querying data with SQL

The cornerstone of every relational database is the declarative language called *Structured Query Language* (SQL). Although each relational database has a slightly different syntax, the fundamentals are pretty much the same across all relational DBMSs.

One of the most common things done with SQL is to query relational data. SQL of this nature is often referred to as a Data Manipulation Language (DML) and consists of clauses specifically designed for this purpose. The other side of DML is updating data with SQL, which we’ll cover in section C.3.

### C.2.1 SELECT, FROM, WHERE, and ORDER BY clauses

For accessing data, you use a `SELECT` statement, usually accompanied with a `FROM` and a `WHERE` clause. The `SELECT` part of the statement restricts the columns that will be returned, the `FROM` clause determines where the data comes from, and `WHERE` restricts the number of records to be returned.

When returning constants or simple calculations that come from nowhere, the `FROM` clause isn’t needed in PostgreSQL, SQL Server, or MySQL, whereas in databases such as Oracle and IBM DB2, you need to select `FROM dual` or `sys.dual` or some other dummy table.

#### Basic SELECT

A basic select looks something like this:

```sql
SELECT gid, item_name, the_geom
    FROM feature_items
    WHERE item_name LIKE 'Queens%';
```

Keep in mind that PostgreSQL is by default case sensitive, and if you want to do a non-case-sensitive search, you’d do the following (or use the non-portable `ILIKE` PostgreSQL predicate):

```sql
SELECT gid, item_name, the_geom
    FROM feature_items
    WHERE upper(item_name) LIKE 'QUEENS%';
```
There’s no guaranteed order for results to be returned in, but sometimes you’ll care about order. The SQL ORDER BY clause satisfies this need for order.

Following is an example that lists all items starting with *Lion* and orders them by item_name:

```sql
SELECT DISTINCT item_name
FROM feature_items
WHERE upper(item_name) LIKE 'LION%'
ORDER BY upper(item_name);
```

For versions of PostgreSQL prior to 8.4, you should uppercase your ORDER BY field, but PostgreSQL 8.4 provides a new per-database collation feature that makes this not as necessary, depending on the collation order you’ve designated for your database.

**SELECT * IS NOT YOUR FRIEND**

Within a SELECT statement you can use the term *, which means “select all the fields in the FROM tables.” There is also the variant sometable.* if you want to select all fields from only one table and not all fields from the other tables in your FROM.

We highly recommend you stay away from this with production code. It’s useful for seeing all the columns of a table when you don’t have the table structure in front of you, but it can be a real performance drain, especially with tables that hold geometries.

If you have a table with a column that’s unconstrained by size, such as a large text field or geometry field, you’ll be pulling all that data across the wire and pulling from disk even when you don’t care about the contents of all the fields. It’s also dangerous to use * when you have multiple tables in a join that have the same column names, because the column names would be output, and whatever code you’re running might arbitrarily pick the wrong column, depending on the language you’re using.

**INDEXES**

The WHERE clause often relies on an index to improve row selection. If you have a large number of distinct groupings of records that have the same value in a field, it’s useful to put an index on that field. For a few distinct groupings of records by a column, the index is more harmful than helpful, because the planner will ignore it and do a faster table scan, and updating will incur a heavy performance penalty.

**ALIASING**

In listing C.1 using information_schema, we demonstrated the concept of aliasing. Aliasing is giving a table or a column a different name in your query than how it’s defined in the database.

Aliasing is done with an AS clause. For table aliases, AS is optional for most ANSI SQL standard databases including PostgreSQL. For column aliases, AS is optional for most ANSI SQL databases and PostgreSQL 8.4+ but required for PostgreSQL 8.3 and below.

Aliasing is indispensable when doing self-joins (where you join the same table twice) and you need to distinguish between the two, or where the two tables you have may have field names in common. The other use of aliases is to make your code easier to read and to reduce typing by shortening long table and field names.
WHY USE AS WHEN YOU DON’T NEED TO

Although AS is an optional clause, we like to always put it in for clarity, but it’s really a matter of preference. To demonstrate, which of the following is more understandable?

```
SELECT b.somefield a FROM sometable b;
```

or

```
SELECT b.somefield AS a FROM sometable AS b;
```

### C.2.2 Using subselects

The SQL language has built-in support for subselects. Much of the expressiveness and complexity of SQL consists of keeping subselects straight and knowing when and when not to use them. For PostgreSQL, most valid SELECT . . . clauses can be used as subselects, and when used in a FROM clause, the subselect must be aliased. For some databases, such as SQL Server, there are some minor limitations; for example, SQL Server doesn’t allow an ORDER BY in a subselect without a TOP clause.

A subselect statement is a full SELECT . . . FROM . . . statement that appears within another SQL statement. It can appear in the following locations of an overall SQL statement:

- **In a UNION, INTERSECT, or EXCEPT**—You’ll learn about these shortly.
- **In a FROM clause**—When used in a FROM, the subselect acts as a virtual table, and it needs to have an alias name to define how it will be called in other parts of the query. Also, in versions prior to PostgreSQL 9.3, the subselect can’t reference other FROM table fields as part of its definition. Some databases allow you to do this under certain conditions, such as SQL Server’s 2005+ CROSS/OUTER APPLY and the LATERAL clause in PostgreSQL 9.3+. Note that for PostgreSQL 9.3+, the LATERAL clause term is optional, and you can simply use a subselect that references other tables in the FROM. We’ll demonstrate LATERALS later in this appendix.
- **In the definition of a calculated column**—When used in this context, the subselect can return only one column and one row. This pretty much applies to all databases, but PostgreSQL has a somewhat unique feature because of the way it implements rows. Each table has a data type that contains all the columns of its row. As such, a table can be used as a data type of a column. In addition, a row, even one coming from a subselect, is a composite type. This allows you to get away with returning a multicolumn row as a field expression. This isn’t a feature you’ll commonly find in other databases, so we won’t cover it in this appendix. You can, however, return multiple rows as an array if they contain only one column by using ARRAY in PostgreSQL. This will return the column as an array of that type. We demonstrate this in various parts of the book. Again, this is a feature that’s fairly unique to PostgreSQL and is very handy for spatial queries.
- **In the WHERE part of another SQL query**—Subselects can be used in IN, NOT IN, and EXISTS clauses.
In a WITH clause—A SELECT query in a WITH clause is loosely defined as a subselect but is not strictly thought of that way. Note that the WITH clause is available only in PostgreSQL 8.4+. You’ll also find it in Oracle, SQL Server 2005+, IBM DB2, and Firebird. You won’t find it in MySQL.

One caution with subselects in WITH clauses when used in PostgreSQL is that they’re always materialized. With materialization is still the case in PostgreSQL 9.4, but it isn’t necessarily the case with other databases that support the construct. Materialization is both good and bad. It’s good because you can use it as a kind of HINT to the planner to materialize, but bad because if you just consider it to be syntactic sugar, you could slow down your queries significantly.

What is a correlated subquery?

A correlated subquery is a subquery that uses fields from the outer query (the next level above the subquery) to define the subquery. Correlated subqueries are often used in column expressions and WHERE clauses. They’re generally slower than non-correlated subqueries because they have to be calculated for each unique combination of fields and have a dependency on the outer query.

The following listing shows some examples of subselects in action. Don’t worry if you don’t completely comprehend them, because some require an understanding of topics that we’ll cover shortly.

Listing C.3 Subselects used in a table alias

```sql
SELECT s.state, r.cnt_residents, c.land_area
FROM states As s LEFT JOIN
    (SELECT state, COUNT(res_id) As cnt_residents
     FROM residents
     GROUP BY state) AS r ON s.state = r.state
LEFT JOIN (SELECT state, SUM(ST_Area(the_geom)) As land_area
     FROM counties
     GROUP BY state) As c
    ON s.state = c.state;
```

This statement uses a subselect to define the derived table we alias as r. This is the common use case.

The same statement is demonstrated in the following listing using the PostgreSQL 8.4 WITH clause. The WITH clause, sometimes referred to as a common table expression (CTE), is an advanced ANSI SQL feature that you’ll find in SQL Server, IBM DB2, Oracle, and Firebird, to name a few.

Listing C.4 The same statement written using the WITH clause

```sql
WITH r A S
    (SELECT state, COUNT(res_id) As cnt_residents
     FROM residents
     GROUP BY state),
```
Querying data with SQL

(c AS (SELECT state, SUM(ST_Area(the_geom)) As land_area
     FROM counties
     GROUP BY state)
SELECT s.state, r.cnt_residents, c.land_area
FROM states As s LEFT JOIN
r ON s.state = r.state
LEFT JOIN c
     ON s.state = c.state;

Now let’s look at the same query written using a correlated subquery.

Listing C.5  The same statement written using a correlated subquery

SELECT s.state,
     (SELECT COUNT(res_id)
         FROM residents
         WHERE residents.state = s.state) AS cnt_residents
     , (SELECT SUM(ST_Area(the_geom))
         FROM counties
         WHERE counties.state = s.state) AS land_area
FROM states As s ;

Although you can use any of these variations to get the same results, the strategies used by the planner are very different. Depending on what you’re doing, one can be much faster than another. If you expect large numbers of returned rows, you should avoid the correlated subquery approach, although in certain cases it can be necessary to use a correlated subquery, such as when you need to prevent duplication of counts. You also don’t want to use a CTE for expressions that would result in many rows that you won’t always return. This is because the CTE will materialize the subexpression and will also not be able to employ indexes on the underlying table for subsequent uses in later expressions.

C.2.3  JOINS

PostgreSQL supports all the standard JOINs and sets defined in the ANSI SQL standards.

A JOIN is a clause that relates two tables, usually by a primary and a foreign key, although the join condition can be arbitrary. In spatial queries, you’ll find that the JOIN is often based on a proximity condition rather than on keys. The clauses LEFT JOIN, INNER JOIN, CROSS JOIN, RIGHT JOIN, FULL JOIN, and NATURAL JOIN exist in the ANSI SQL specifications, and PostgreSQL supports all of these. SQL Server supports them as well, but MySQL lacks FULL JOIN support. Oracle supports them as well.

LEFT JOIN

A LEFT JOIN returns all records from the first table (M) and only records in the second table (N) that match records in the first table (M). The maximum number of records returned by a LEFT JOIN is \( m \times n \) rows, where \( m \) is the number of rows in M
and $n$ is the number of rows in $N$. The number of columns is the number of columns selected from $M$ plus the number of columns selected from $N$.

Generally speaking, if your $M$ table has a primary key that’s the joining field, you can expect the minimum number of rows returned to be $m$ and the maximum to be $m + m \times n$.

NULL placeholders are put in table $N$’s columns where there’s no match in the $M$ table. You can see a diagram of a LEFT JOIN in figure C.1.

Let’s look at some examples of LEFT JOINS:

```sql
SELECT c.city_name, a.airport_code, a.airport_name, a.runlength
FROM city AS c
LEFT JOIN airports AS a ON a.city_code = c.city_code;
```

This query will list both cities that have airports and cities that don’t have airports, based on `city_code`. We assume `city_code` to be the city table’s primary key and a foreign key in the airports table. If the LEFT JOIN were changed to an INNER JOIN, only cities with airports would be listed. With a LEFT JOIN, cities that have no airports will get a NULL placeholder for the airport fields.

One trick commonly used with LEFT JOINS is to return only unmatched rows by taking advantage of the fact that a LEFT JOIN will return NULL placeholders where there’s no match. When doing this, make sure the field you’re joining with is guaranteed to be filled in when there are matches; otherwise, you’ll get spurious results. For example, a good candidate would be the primary key of a table. Here’s an example of such a trick:

```sql
SELECT c.city_name
FROM city AS c
LEFT JOIN airports AS a ON a.city_code = c.city_code
WHERE a.airport_code IS NULL;
```

In this example, you’re returning all cities with no matching airports. You’re making the assumption that the `airport_code` is never NULL in the airports table. If it were ever NULL, this wouldn’t work.

**INNER JOIN**

An INNER JOIN returns only records that are in both the $M$ and $N$ tables, as shown in figure C.2. The maximum number of records you can expect from an inner join is $(m \times n)$. Generally speaking, if your $M$ table has a primary key, and that field is used to
join with a field in table N, you can expect the maximum number of rows to be $n$. A classic example is customers joined with orders. If a customer has only five orders, the number of rows you’ll get back with that customer ID and name is five.

Following is an example of an INNER JOIN:

```
SELECT c.city_name, a.airport_code, a.airport_name, a.runlength
FROM city AS c
INNER JOIN airports a ON a.city_code = c.city_code;
```

In this example, you list only cities that have airports and only the airports in them. If you had a spatial database, you could do a JOIN using a spatial function such as `ST_Intersects` or `ST_DWithin` and you could find airports in proximity to a city or in a city region.

### RIGHT JOIN

A RIGHT JOIN returns all records in the N table and only records in the M table that match records in N, as shown in figure C.3. In practice, RIGHT JOINS are rarely used because a RIGHT can always be replaced with a LEFT, and most people find reading join clauses from left to right easier to comprehend. The RIGHT JOIN’s behavior is a mirror image of the LEFT JOIN, flipping the table order in the clause.

### FULL JOIN

The FULL JOIN, shown in figure C.4, returns all records in M and N and uses NULLs as placeholders in fields where there’s no matching data. There’s a lot of debate about the usefulness of FULL JOINS. In practice they’re rarely used, and some people are of the opinion that they should never be used because they can always be simulated with
a UNION [ALL]. Although we rarely use FULL JOINs, in some cases we find them clearer to use than a UNION [ALL].

The number of columns returned by a FULL JOIN is the same as for a LEFT, RIGHT, or INNER join; the minimum number of rows returned is max(m, n) and the maximum is (max(m, n) + m × n – min(m, n)).

**FULL JOINs on spatial relationships—forget about it**

While in theory it’s possible to do a FULL JOIN using spatial functions like ST_DWithin or ST_Intersects, in practice this isn’t currently supported, even as of PostgreSQL 9.4 and PostGIS 2.1.

**CROSS JOIN**

The CROSS JOIN is the cross product of two tables, where every record in the M table is joined with every record in the N table, as illustrated in figure C.5. The result of a CROSS JOIN without a WHERE clause is m × n rows. It’s sometimes referred to as a Cartesian product.

Here’s an example of a good use for a CROSS JOIN. This statement calculates the total price of a product, including state tax, for each state:

```sql
SELECT p.product_name, s.state,
       p.base_price * (1 + s.tax) As total_price
FROM products AS p
  CROSS JOIN state AS s;
```

It can also be written as follows:

```sql
SELECT p.product_name, s.state,
       p.base_price * (1 + s.tax) As total_price
FROM products AS p, state AS s
```
Note that table1 INNER JOIN table2 ON (table1.field1 = table2.field2) can be written with the table1 CROSS JOIN table2 or table1,table2 syntax and then followed with a WHERE table1.field1 = table2.field2 part, but we prefer the more explicit INNER JOIN because it’s less prone to mistakes.

When doing an INNER JOIN with CROSS JOIN syntax, you put the join fields in the WHERE clause. Primary keys and foreign keys are often put in the INNER JOIN ON clause, but in practice you can put any joining field in there. There’s no absolute rule about it. The distinction becomes important when doing LEFT JOINS, as you saw with the LEFT JOIN orphan trick.

**NATURAL JOIN**

A NATURAL JOIN is like an INNER JOIN without an ON clause. It’s supported by many ANSI-compliant databases. The NATURAL JOIN automagically joins same-named columns between tables, so there’s no need for an ON clause.

**JUST SAY NO TO THE NATURAL JOIN**

We highly suggest you stay away from using NATURAL JOINS. It’s a lazy and dangerous way of doing joins that will come back to bite you when you add fields with the same names as fields in other tables that are totally unrelated. We feel so strongly about not using NATURAL JOINS that we won’t even demonstrate their use. So when you see one in use, instead of thinking cool, just say no.

**CHAINING JOINS**

The other thing with JOINS is that you can chain them almost ad infinitum. You can also combine multiple JOIN types, but when joining different types, either make sure you have all your INNER JOINS before the LEFTs or put parentheses around them to control their order.

Here’s an example of JOIN chaining:

```sql
SELECT c.last_name, c.first_name, r.rental_id, p.amount, p.payment_date
FROM customer As C
    INNER JOIN rental As r ON C.customer_id = r.customer_id
    LEFT JOIN payment As p
        ON (p.customer_id = r.customer_id
            AND p.rental_id = r.rental_id);
```

This example is from the PostgreSQL pagila database. The pagila database is a favorite for demonstrating new features of PostgreSQL. You can download it from “PostgreSQL sample databases” (www.postgresql.org/ftp/projects/pgFoundry/dbsamples).

In the preceding example, you find all the customers who have had rentals, and you list the rental fields as well (note that the INNER JOIN excludes all customers who haven’t made rentals). You then pull the payments they’ve made for each rental. You’ll get NULLs if no payment was made but the rental exists.

**C.2.4 Sets**

The set predicates UNION [ALL], EXCEPT, INTERSECT, like a JOIN, can contain multiple tables or subqueries. What distinguishes the set class of predicates from a JOIN is that
they chain together SQL statements that can normally stand by themselves to return a single data set. The set class defines the kind of chaining behavior. Keep in mind that when we talk about sets here, we’re not talking about the SET clause you’ll find in UPDATE statements.

PostgreSQL supports all three set predicates, though many databases support only UNION [ALL].

One other distinguishing thing about sets is that the number of columns in each SELECT has to be the same, and the data types in each column should be the same too, or autocast to the same data type in a non-ambiguous way.

**SPATIAL PARALLELS**

One thing that confuses new spatial database users is the parallels between the two terminologies. In general SQL lingua franca, you have UNION, INTERSECT, and EXCEPT, which talk about table rows. When you add space to the mix, you have parallel terminology for geometries: ST_Union (which is like a UNION), ST_Collect (which is like a UNION ALL), ST_Intersection (which is like INTERSECT), and ST_Difference (which is like EXCEPT) serve the same purposes for geometries.

**UNION AND UNION ALL**

The most common type of set includes the UNION and UNION ALL sets, illustrated in figure C.6. Most relational databases have at least one of these, and most have both. A UNION takes two SELECT statements and returns a DISTINCT set of these, which means no two records will be exactly the same. A UNION ALL, on the other hand, always returns \( n + m \) rows, where \( n \) is the number of rows in table N and \( m \) is the number of rows in table M.

A union can have multiple chains, each separated by a UNION ALL or a UNION. ORDER BY can appear only once and must be at the end of the chain. ORDER BY is often denoted by numbers, where the number denotes the column number to order by.

A union is generally used to put together results from different tables. The example in the following listing will list all water and land features greater than 500 units in area and all architecture monuments greater than 1000 dollars, and will order the results by item name.

```sql
Listing C.6 Combining water and land features

SELECT water_name As label_name, the_geom,  
   ST_Area(the_geom) As feat_area  
FROM water_features  
WHERE ST_Area(the_geom) > 10000  
UNION ALL  
SELECT feat_name As label_name, the_geom,
```

**Figure C.6 UNION ALL versus UNION.**
The thick-bordered light-gray box is M, and the thin-bordered dark-gray box is N. On the left, the UNION ALL shared regions are duplicated; in the UNION on the right, only one of the shared regions is kept, resulting in a distinct set.
ST_Area(the_geom) As feat_area
FROM land_features
WHERE ST_Area(feat_geometry) > 500
UNION ALL
SELECT arch_name As label_name, the_geom,
    ST_Area(the_geom) As feat_area
FROM architecture
WHERE price > 1000
ORDER BY 1,3;

This example will pull data from three tables (water_features, land_features, and architecture) and return a single data set ordered by the name of the feature and then the area of the feature.

**UNION IS OFTEN MISTAKENLY USED**

The plain UNION statement is often mistakenly used because it’s the default option when ALL isn’t specified. As stated, UNION does an implicit DISTINCT on the data set, which makes it slower than a UNION ALL. It also has another side effect of losing geometry records that have the same bounding boxes. In general, you’ll want to use a UNION ALL except when deduping data.

**INTERSECT**

INTERSECT is used to join multiple queries, similar to UNION. It’s defined in the ANSI SQL standard, but not all databases support it; for example, MySQL doesn’t support it, and neither does SQL Server 2000, although SQL Server 2005 and above do.

INTERSECT returns only the set of records that is common between the two result sets, as shown in figure C.7. It’s different from INNER JOIN in that it isn’t multiplicative and in that both queries must have the same number of columns. In figure C.7, the shaded area represents what’s returned by an SQL INTERSECT.

Later we’ll look at a spatial intersection involving an intersection of geometries rather than an intersection of row spaces.

INTERSECT is rarely used, and there are a few reasons for that:

- Many relational databases don’t support it.
- It tends to be slower than doing the same trick with an INNER JOIN. In PostgreSQL 8.4, the speed of INTERSECTS has been improved, but in prior versions it wasn’t that great.
- In some cases, it looks convoluted when you’re talking about the same table.
In some cases, though, `INTERSECT` does make your code clearer, such as when you have two disparate tables or when you chain more than two queries. The following listing demonstrates `INTERSECT` and the equivalent query using `INNER JOIN`.

**Listing C.7  INTERSECT compared to INNER JOIN**

```sql
SELECT feature_id
FROM water_features
WHERE ST_Area(geom) > 500
INTERSECT
SELECT feature_id
FROM protected_areas
WHERE induction_year > 2000;

SELECT wf.feature_id
FROM water_features As wf
INNER JOIN
protected_areas As pa ON wf.feature_id = pa.feature_id
WHERE ST_Area(wf.geom) > 500
AND pa.induction_year > 2000;
```

The first query uses the `INTERSECT` approach to list all water features greater than 500 square units that are also designated as protected areas inducted after the year 2000 1.

The second approach demonstrates the same query, but using an `INNER JOIN` instead of an `INTERSECT` 2. Note that if the `feature_id` field isn’t unique in both tables, the `INNER JOIN` runs the chance of multiplying records. To overcome that, you can change the `SELECT` to `SELECT DISTINCT`.

The next example demonstrates chaining `INTERSECT` clauses:

```sql
SELECT r
FROM generate_series(1,3) AS r
INTERSECT
SELECT n
FROM generate_series(3,8) AS n
INTERSECT
SELECT s
FROM generate_series(2,3) AS s;
```

Keep in mind that you can mix and match with `UNION` and `EXCEPT` as well. The order of precedence is from top query down, unless you have subselect parenthetical expressions.

**EXCEPT**

An `EXCEPT` chains queries together such that the final result contains only records in A that aren’t in B. The number and type of columns in each chained query must be the same, similar to `UNION` and `INTERSECT`. The shaded section in figure C.8 represents the result of the final query.
EXCEPT is rarely used, but it does come in handy when chaining multiple clauses:

```sql
SELECT r
  FROM generate_series(1,3) AS r
EXCEPT
SELECT n
  FROM generate_series(3,8) AS n
INTERSECT
SELECT s
  FROM generate_series(2,3) AS s;
```

### C.2.5 Using SQL aggregates

Aggregate functions roll a group of records into one record. In PostgreSQL, the standard SUM, MAX, MIN, AVG, COUNT, and various statistical aggregates are available out of the box. PostGIS adds many more for geometry, raster, and topology types, of which ST_Collect, ST_Union, ST_MakeLine, and ST_Extent are the most commonly used. In this section, we’ll focus on using aggregates. How you use aggregates is pretty much the same regardless of whether they’re spatial or not.

Aggregates in SQL generally have the following parts:

- **SELECT and FROM**—This is where you select the fields and where you pull data from. You also include the aggregated functions in the select field list.
- **SOMEAGGREGATE(DISTINCT somefield)**—On rare occasions, you’ll use the DISTINCT clause within an aggregate function to denote that you use only a distinct set of values to aggregate. This is commonly done with the COUNT aggregate to count a unique name only once.
- **WHERE**—Non-aggregate filter; this gets applied before the HAVING part.
- **HAVING**—Similar to WHERE, but used when applying filtering on the already aggregated data.
- **GROUP BY**—All fields in the SELECT that are non-aggregated and function calls must appear here (pre-PostgreSQL 9.1).
- **SOMEAGGREGATE(somefield ORDER BY someotherfield1, ... someotherfieldn)**—This is a feature new in PostgreSQL 9.0. It’s most useful for aggregates that return output composed of subelements, and it controls the order in which these are returned. In common PostgreSQL usage, you’ll often see this used with string_agg or array_agg. In the PostGIS world, you’ll see this used commonly with the ST_MakeLine aggregate function where you need to control the order of the line’s points by time or some other column. The ORDER BY construct can be combined with DISTINCT in the form SOMEAGGREGATE(DISTINCT somefield ORDER BY somefield). Note that in the DISTINCT form, similar to a regular SELECT DISTINCT, the ORDER BY columns need to be inputs to the function.

**WARNING** With geometries, what is DISTINCTed is the bounding box, so different geometries with the same bounding box will get thrown out.
GROUP BY FUNCTIONAL DEPENDENCY ENHANCEMENT

PostgreSQL 9.1 introduced the functional dependency feature, which means that if you’re already grouping by a primary key of a table, you can skip grouping by other fields in that table. This feature is defined in the ANSI SQL 99 standard. It saves some typing as well as makes it easier to port some MySQL apps.

FAST FACTS ABOUT AGGREGATE FUNCTIONS

There are some important things you should keep in mind when working with aggregate functions. Some of these facts are standard across all relational databases, some are specific to PostgreSQL, and some are a consequence of the way PostGIS implements = for geometries:

- For most aggregate functions, NULLs are ignored. This is important to know, because it allows you to do things such as \( \text{COUNT(geom)} \) AS \( \text{num\_has\_geoms} \) and \( \text{COUNT(neighborhood)} \) AS \( \text{num\_has\_neighborhoods} \) in the same SELECT statement.
- If you want to count all records, use a field that’s never NULL to count; for example, \( \text{COUNT(gid)} \) or a constant such as \( \text{COUNT(1)} \). You can also use \( \text{COUNT(*)} \). Prior to PostgreSQL 8.1, the \( \text{COUNT(*)} \) function was really slow, so long-time PostgreSQL users tend to avoid that syntax out of habit.
- When grouping by geometries, which is very rare, it’s the bounding box of the geometry that’s actually grouped on (although the first geometry with that bounding box is used for output), so be very careful and avoid grouping by geometry if possible, unless you have another field in the GROUP BY that’s distinct for each geometry, like the primary key of the table the geometry is coming from.

The following listing is an example that mixes aggregate SQL functions with spatial aggregates.

**Listing C.8 Combining standard SQL and spatial aggregates**

```sql
SELECT n.nei_name,
       SUM(ST_Length(roads.geom)) as total_road_length,
       ST_Extent(roads.geom) As total_extent,
       COUNT(DISTINCT roads.road_name) As count_of_roads
FROM neighborhoods As n
INNER JOIN roads ON
    ST_Intersects(neighborhoods.geom, roads.geom)
WHERE n.city = 'Boston'
GROUP BY n.nei_name
HAVING ST_Area(ST_Extent(roads.geom)) > 1000;
```

The query for each neighborhood specifies the total length of roads and the extent of all roads. It also includes a count of unique road names and counts only neighborhoods where the total area of the extent covered is greater than 1,000 square units.
C.2.6 Window functions and window aggregates

PostgreSQL 8.4 introduced the ANSI-standard window functions and aggregates, and PostgreSQL 9.0 improved on this by expanding the functionality of BETWEEN ROWS and RANGE. PostgreSQL 9.4 improves even more on this feature by adding percentile_cont and percentile_dist as well as the complementary WITHIN GROUP (ORDER BY somefield) to go along with these new functions.

Window functionality allows you to do useful things such as sequentially number results by some sort of ranking; calculate running subtotals based on a subset of the full set using the concept of a window frame; and, for PostGIS 1.4+, perform running geometry ST_Union and ST_MakeLine calls, which are perhaps solutions in search of a problem but nevertheless intriguing.

A window frame defines a subset of data within a subquery using the term PARTITION BY, and then within that window you can define orderings and sum results to achieve rolling totals and counts. Microsoft SQL Server, Oracle, and IBM also support this feature, with Oracle’s feature set being the strongest and SQL Server’s being weaker than that of IBM DB2 or PostgreSQL. Check out our brief summary comparing these databases on the Postgres Online Journal to get a sense of the differences: www.postgresonline.com/journal/archives/122-Window-Functions-Comparison.html

PostgreSQL also supports named window frames that can be reused by name.

The example in the following listing uses the ROW_NUMBER window function to number streets sequentially within one kilometer of a police station, ordered by their proximity to the police station.

Listing C.9 Find roads within 1 km of each police station and number sequentially

```sql
SELECT ROW_NUMBER() OVER ( PARTITION BY loc.pid
	ORDER BY ST_Distance(r.geom, loc.geom)
	, r.road_name) As row_num,
loc.pid, r.road_name,
ST_Distance(r.geom, loc.geom)/1000 As dist_km
FROM land As loc
LEFT JOIN road As r
ON ST_DWithin(r.geom, loc.geom, 1000)
WHERE loc.land_type = 'police station'
ORDER BY pid, row_num;
```

In this listing, you use the window function called ROW_NUMBER to number the results 1. The PARTITION BY clause forces numbering to restart for each unique parcel ID (identified by pid) that uniquely identifies a police station 2. The ORDER BY defines the ordering 3. In this case, you increment based on proximity to the police station. If two streets happen to be at the same distance, then one will arbitrarily be n and the other n+1. The ORDER BY includes road_name as a tie breaker.

Table C.1 shows a subset of the resulting table for two police stations.
C.2.7 LATERALs

New in PostgreSQL 9.3 is the LATERAL clause. The LATERAL clause is an ANSI SQL feature that allows you to specify tables, functions (particularly set-returning functions), or subqueries in the FROM clause that reference columns from preceding FROM items. A common use of the LATERAL clause in PostGIS work is to expand spatial objects into component parts. LATERAL is also used in conjunction with KNN gist operators <-> and <#> to overcome the constant reference geometry issue, and doing limit queries where, for example, for each record location you want to return the five closest items from another set of data. We demonstrated some examples of its use in distance searching in chapter 10, and in chapter 8 for batch geocoding.

The basic use of LATERAL in conjunction with KNN operators such as <-> is shown in the following example, which returns the closest three hospitals for each location:

```sql
SELECT loc.address, hosp.name
FROM loc CROSS JOIN
  LATERAL (SELECT name FROM hospitals
             ORDER BY hospitals.geom <-> loc.geom LIMIT 3) As hosp
```

As with all LATERAL constructs, the hosp subquery relies on column geom from the loc table, and each value of loc.geom begets a new subtable. loc.geom is used as a constant in the subquery, so it can utilize spatial indexes.

In cases where you aren’t sure if a subquery will return a result for each record in your preceding FROM table, you can combine LATERAL with a LEFT JOIN, as this next example demonstrates:

```sql
SELECT loc.address, hosp.name
FROM loc LEFT JOIN
  LATERAL (SELECT name FROM hospitals
            ORDER BY hospitals.geom <-> loc.geom LIMIT 3) As hosp ON true
```

In the next section, you’ll learn about another key component of SQL. SQL is good for querying data, but it’s also useful for updating and adding data.
C.3 UPDATE, INSERT, and DELETE

The other feature of a DML is the ability to update, delete, and insert data. SQL’s UPDATE, DELETE, and INSERT statements can combine the predicates you learned about for selecting data. They can perform cross updates between tables or formulate a virtual table (subquery) to insert data into a physical table. In the exercises that follow, we’ll demonstrate simple constructs as well as more complex ones.

C.3.1 UPDATE

You can use the SQL UPDATE statement to update existing data. You can update individual records or a batch of records based on some WHERE condition.

SIMPLE UPDATE

A simple UPDATE will update data to a static value based on a WHERE condition. Here’s a simple example of this:

UPDATE things
    SET status = 'active'
    WHERE last_update_date > (CURRENT_TIMESTAMP - '30 day'::interval);

UPDATE FROM OTHER TABLES

A simple UPDATE is one of the more common update statements used. In certain cases, however, you’ll need to read data from a separate table based on some sort of related criteria. In this case, you’ll need to use joins within your UPDATE statement.

Here’s a simple example that updates the region code of a point data set if the point falls within the region:

UPDATE things
    SET region_code = r.region_code
    FROM regions As r
    WHERE ST_Intersects(things.geom, r.geom);

UPDATE WITH SUBSELECTS

A subselect, as you learned earlier, is like a virtual table. It can be used in UPDATE statements the same way you use regular tables. In a regular UPDATE statement, even involving statements with table joins, you can’t update a table value with an aggregate such as the SUM of another table field. A way to get around this limitation of SQL is to use a subselect.

Following is an example that tallies the number of objects in a region:

UPDATE regions
    SET total_objects = ts.cnt
    FROM (SELECT t.region_code, COUNT(t.gid) As cnt
          FROM things AS t
          GROUP BY t.region_code) As ts
    WHERE regions.region_code = ts.region_code;

If you’re updating all rows in a table, it’s often more efficient to build the table from scratch and use an INSERT statement rather than an UPDATE statement. The reason for
this is that an UPDATE is really an INSERT and a DELETE. Because of the multi-version concurrency control (MVCC) implementation of PostgreSQL, PostgreSQL will remove the old row and replace it with the new row in the active heap.

Next you’ll learn how to perform INSERTs.

C.3.2 INSERT

Just like the UPDATE statement, you can have simple INSERTs that insert constants as well as more complex ones that read from other tables or aggregate data. We’ll demonstrate some of these constructs.

SIMPLE INSERT

The simple INSERT just inserts constants, and it comes in three basic forms.

The single-value constructor approach has existed in PostgreSQL since the 6.0 days and is pretty well supported across all relational databases. Here you insert a single point:

```
INSERT INTO points_of_interest(fe_name, geom)
VALUES ('Highland Golf Club',
    ST_SetSRID(ST_Point(-70.063656, 42.037715), 4269));
```

The next most popular approach is the multi-row VALUES constructor syntax introduced in SQL 92, which we demonstrate often in this book. This syntax was introduced in PostgreSQL 8.2 and IBM DB2, has been supported for a long time in MySQL (we think since 3+), and was introduced in SQL Server 2008. As of this writing, Oracle has yet to support this useful construct.

The multi-row VALUES constructor is useful for adding more than a single row or as a faster way of creating a derived table with just constants. Listing C.10 shows an example of a multi-row VALUES insert. It’s similar to the single insert. It starts with the word VALUES, and then each row is enclosed in parentheses and separated with a comma.

```
INSERT INTO poi(poi_name, poi_geom)
VALUES ('Park',
    ST_GeomFromText('POLYGON ((86980 67760, 43975 71292, 43420 56700, 91400 35280, 91680 72460, 89460 75500, 86980 67760))'),
    ('Zoo', ST_GeomFromText('POLYGON ((41715 67525, 61393 64101, 91505 49252, 91505 49252, 91400 35280, 41715 67525))');
```

The last kind of simple INSERT is one that uses the SELECT clause, as shown in listing C.11. In the simplest example, it doesn’t have a FROM. Some people prefer this syntax because it allows you to alias what the value is right next to the constant. It’s also a necessary syntax for the more complex kind of INSERT we’ll demonstrate in the next section.
UPDATE, INSERT, and DELETE

INSERT INTO poi(poi_name, geom)
SELECT 'Park' AS poi_name,
    ST_GeomFromText('POLYGON ((86980 67760, 43975 71292, 43420 56700, 91400 35280, 91680 72460, 89460 75500, 86980 67760))') As geom
UNION ALL
SELECT 'Zoo' As poi_name,
    ST_GeomFromText('POLYGON ((41715 67525, 61393 64101, 91505 49252, 91400 35280, 41715 67525))') As geom;

This is the standard way of inserting multiple rows into a table. It was the only way to do a multi-row INSERT before PostgreSQL 8.2. This syntax is supported by PostgreSQL (all versions), MySQL, and SQL Server. To use it in something like Oracle or IBM DB2, you need to include a FROM clause, like FROM dual or sys.dual.

ADVANCED INSERT

The advanced INSERT is not that advanced. You use this syntax to copy data from one table or query to another table. In the simplest case, you’re copying a filtered set of data from another table. It uses the SELECT syntax, usually with a FROM and sometimes accompanying joins.

This example inserts a subset of rows from one table into another:

```
INSERT INTO polygons_of_interest(fe_name, geom, interest_type)
SELECT pid, geom, 'less than 300 sqft' As interest_type
FROM parcels WHERE ST_Area(geom) < 300;
```

A slightly more advanced INSERT is one that joins several tables together. In this scenario, the SELECT FROM is just a standard SQL SELECT statement with joins, or one that consists of subselects. Listing C.12 is a somewhat complex case: given a table of polygon chain link edges, it constructs polygons and stuffs them into a new table of polygons.

```
Listing C.11  Simple value INSERT using SELECT instead of VALUES

Listing C.12  Construct polygons from line work and insert into polygon table

INSERT INTO polygons(polyid, geom)
SELECT polyid, ST_Multi(final.geom) As geom
FROM (SELECT pc.polyid,
    ST_BuildArea(ST_Collect(pc.geom)) As geom
FROM (SELECT p.right_poly as polyid, lw.geom
    FROM polychain p INNER JOIN linework lw ON
    lw.tlid = p.tlid
    WHERE (p.right_poly <> p.left_poly OR p.left_poly IS NULL)
    UNION ALL
    SELECT p.left_poly as polyid, lw.geom
    FROM polychain p INNER JOIN linework lw ON
    lw.tlid = p.tlid
    WHERE (p.right_poly <> p.left_poly OR p.right_poly IS NULL)
    ) AS pc
GROUP BY poly.polyid) As final;
```
**SELECT INTO and CREATE TABLE AS**

Another form of the `INSERT` statement is what we commonly refer to as a *bulk INSERT*. In this kind of `INSERT`, not only are you inserting data, but you’re also creating the table to hold the data in a single statement. PostgreSQL supports two basic forms of this:

- The standard `SELECT ... INTO` is supported by a lot of relational databases. We prefer this approach because it’s more cross-platform (it will work on SQL Server as well as MySQL, for example).
- The `CREATE TABLE ... AS SELECT ...` approach isn’t as well supported by other relational databases.

In both cases, any valid `SELECT` or `WITH` statement can be used. The following listing shows examples of the same statement written using `SELECT INTO` and `CREATE TABLE AS`.

```sql
Listing C.13 Examples of SELECT INTO and CREATE TABLE

SELECT t.region_code
, COUNT(t.gid) As cnt
  INTO thingy_summary
FROM things AS t
GROUP BY t.region_code;

CREATE TABLE thingy_summary AS
  SELECT t.region_code, COUNT(t.gid) As cnt
  FROM things AS t
  GROUP BY t.region_code;
```

### C.3.3 DELETE

When doing a `DELETE`, there are four basic forms: a simple delete that just involves one table, a medium delete that involves deleting data `USING` matching data in another table or subselect, and the `NOT IN` or `IN` approach, which utilizes a correlated or uncorrelated subquery. Finally, there’s the `TRUNCATE TABLE` approach, which is the fastest and deletes all data in a table, but only works if your table has no related foreign-key constraints in other tables.

**Simple DELETE**

A simple `DELETE` has no subselects but usually has a `WHERE` clause. All the data in a table is deleted and logged if you’re missing a `WHERE` clause.

Following is an example of a standard `DELETE`:

```sql
DELETE FROM streets WHERE fe_name LIKE 'Mass%';
```

**DELETE based on data in another table with USING**

PostgreSQL has a `USING` clause that can be used in a `DELETE`. The `USING` clause denotes tables that should be used for filtering only, and not deleted from. Tables or subqueries that appear in the `USING` clause can be used in the `WHERE` clause of the `DELETE`.

In this next example, you first delete all streets in the current streets table that appear in the `new_streets` data set in preparation for reload:

```sql
DELETE FROM streets USING new_streets WHERE streets.tlid = new_streets.tlid;
```
UPDATE, INSERT, and DELETE

DELETE WITH SUBSELECT IN WHERE

In PostgreSQL, as in many other relational databases, you can’t use a JOIN in the FROM clause to determine what to delete based on another table. You can, however, overcome this restriction by either using a subselect in the WHERE clause or using the PostgreSQL-specific USING clause. The WHERE approach is more cross-platform and the one we’ll demonstrate next. The subselect approach is useful for cases where you need to delete all data in your current table that’s also in the table you’re adding from, or you need to delete duplicate records.

The following example deletes duplicate records:

```sql
DELETE
FROM sometable
WHERE someuniquekey NOT IN
  (SELECT MAX(dup.someuniquekey)
    FROM sometable As dup
    WHERE dup.dupcolumn1 = sometable.dupcolumn1
    AND dup.column2 = sometable.dupcolumn2
    AND dup.column3 = sometable.dupcolumn3
    GROUP BY dup.dupcolumn1, dup.dupcolumn2, dup.dupcolumn3);
```

TRUNCATE TABLE

In cases where you want to delete all the data in a table, you can use the much faster TRUNCATE TABLE statement. TRUNCATE TABLE is considerably faster because it does much less transaction logging than a standard DELETE FROM, but it can be used only in tables that aren’t involved in foreign-key relationships. Here’s an example of it at work:

```sql
TRUNCATE TABLE streets;
```
Processing data tied to location and topology requires specialized know-how. PostGIS is a free spatial database extender for PostgreSQL, every bit as good as proprietary software. With it, you can easily create location-aware queries in just a few lines of SQL code and build the back end for a mapping, raster analysis, or routing application with minimal effort.

PostGIS in Action, Second Edition teaches you to solve real-world geodata problems. It first gives you a background in vector-, raster-, and topology-based GIS and then quickly moves into analyzing, viewing, and mapping data. You’ll learn how to optimize queries for maximum speed, simplify geometries for greater efficiency, and create custom functions for your own applications. You’ll also learn how to apply your existing GIS knowledge to PostGIS and integrate with other GIS tools.

What’s Inside

- An introduction to spatial databases
- Geometry, geography, raster, and topology spatial types, functions, and queries
- Applying PostGIS to real-world problems
- Extending PostGIS to web and desktop applications
- Updated for PostGIS 2.x and PostgreSQL 9.x

Familiarity with relational database and GIS concepts is helpful but not required.

Regina Obe and Leo Hsu are database consultants and authors. Regina is a member of the PostGIS core development team and the Project Steering Committee.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/PostGISinActionSecondEdition