

# Introduction to SQL for Evergreen administrators

---

<b>REVISION HISTORY</b>			
-------------------------	--	--	--

NUMBER	DATE	DESCRIPTION	NAME
0.9	February 2010		DS

---

## Contents

<b>1 Preface</b>	<b>1</b>
<b>2 Part 1: Introduction to SQL Databases</b>	<b>1</b>
2.1 Learning objectives	1
2.2 Introduction	1
2.3 Tables	1
2.4 Schemas	2
2.5 Columns	3
2.6 Constraints	4
2.6.1 Prevent NULL values	4
2.6.2 Primary key	4
2.6.3 Foreign keys	4
2.6.4 Check constraints	5
2.7 Deconstructing a table definition statement	5
2.8 Displaying a table definition using <code>psql</code>	5
<b>3 Part 2: Basic SQL queries</b>	<b>6</b>
3.1 Learning objectives	6
3.2 The SELECT statement	6
3.3 Selecting particular columns from a table	7
3.4 Sorting results with the ORDER BY clause	7
3.5 Filtering results with the WHERE clause	7
3.5.1 Comparison operators	8
Comparing two scalar values	8
3.6 NULL values	9
3.7 Text delimiter: <code>'</code>	9
3.8 Grouping and eliminating results with the GROUP BY and HAVING clauses	10
3.9 Eliminating duplicate results with the DISTINCT keyword	12
3.10 Paging through results with the LIMIT and OFFSET clauses	12
<b>4 Part 3: Advanced SQL queries</b>	<b>13</b>
4.1 Learning objectives	13
4.2 Transforming column values with functions	13
4.2.1 Scalar functions	13
4.2.2 Aggregate functions	14
4.3 Sub-selects	15
4.4 Joins	15
4.4.1 Inner joins	15

---

---

4.4.2	Aliases	16
4.4.3	Outer joins	17
4.4.4	Self joins	18
4.5	Set operations	18
4.5.1	Union	19
4.5.2	Intersection	19
4.5.3	Difference	19
4.6	Views	20
4.7	Inheritance	21
<b>5</b>	<b>Part 4: Understanding query performance with EXPLAIN</b>	<b>21</b>
<b>6</b>	<b>Part 5: Inserting, updating, and deleting data</b>	<b>23</b>
6.1	Inserting data	23
6.2	Inserting data using a SELECT statement	24
6.3	Deleting rows	24
6.4	Updating rows	25
<b>7</b>	<b>Part 6: Query requests</b>	<b>25</b>
7.1	Monthly circulation stats by collection code / library	25
7.2	Monthly circulation stats by borrower stat / library	25
7.3	Monthly intralibrary loan stats by library	26
7.4	Monthly borrowers added by profile (adult, child, etc) / library	26
7.5	Borrower count by profile (adult, child, etc) / library	26
7.6	Monthly items added by collection / library	26
7.7	Hold purchase alert by library	27
7.8	Update borrower records with a different home library	27
<b>8</b>	<b>License</b>	<b>28</b>

---

## 1 Preface

This tutorial is intended primarily for people working with Evergreen who want to dig into the database that powers Evergreen, and who have little familiarity with SQL or the PostgreSQL database in particular. If you are already comfortable with SQL, this tutorial will help point you to the implementation quirks and features of PostgreSQL. If you are already familiar with both SQL and PostgreSQL, this tutorial will help you navigate the Evergreen schema.

## 2 Part 1: Introduction to SQL Databases

### 2.1 Learning objectives

- Understand how tables organize data into columns and rows
- Understand how schemas organize tables and other database objects
- Understand the properties of the most common data types used in Evergreen
- Understand table constraints, including unique constraints, referential constraints, NULL constraints, and check constraints
- Display a table definition using the `psql` command line tool

### 2.2 Introduction

Over time, the SQL database has become the standard method of storing, retrieving, and processing raw data for applications. Ranging from embedded databases such as SQLite and Apache Derby, to enterprise databases such as Oracle and IBM DB2, any SQL database offers basic advantages to application developers such as standard interfaces (Structured Query Language (SQL), Java Database Connectivity (JDBC), Open Database Connectivity (ODBC), Perl Database Independent Interface (DBI)), a standard conceptual model of data (tables, fields, relationships, constraints, etc), performance in storing and retrieving data, concurrent access, etc.

Evergreen is built on PostgreSQL, an open source SQL database that began as POSTGRES at the University of California at Berkeley in 1986 as a research project led by Professor Michael Stonebraker. A SQL interface was added to a fork of the original POSTGRES Berkeley code in 1994, and in 1996 the project was renamed PostgreSQL.

### 2.3 Tables

The table is the cornerstone of a SQL database. Conceptually, a database table is similar to a single sheet in a spreadsheet: every table has one or more columns, with each row in the table containing values for each column. Each column in a table defines an attribute corresponding to a particular data type.

We'll insert a row into a table, then display the resulting contents. Don't worry if the INSERT statement is completely unfamiliar, we'll talk more about the syntax of the insert statement later.

```
evergreen=# INSERT INTO actor.usr_note (usr, creator, pub, title, value)
          VALUES (1, 1, TRUE, 'Who is this guy?', 'He''s the administrator!');

evergreen=# select id, usr, creator, pub, title, value from actor.usr_note;
 id | usr | creator | pub |      title      |      value
-----+-----+-----+-----+-----+-----
  1 |  1 |       1 | t   | Who is this guy? | He's the administrator!
(1 rows)
```

PostgreSQL supports table inheritance, which lets you define tables that inherit the column definitions of a given parent table. A search of the data in the parent table includes the data in the child tables. Evergreen uses table inheritance: for example, the `action.circulation` table is a child of the `money.billable_xact` table, and the `money.*_payment` tables all inherit from the `money.payment` parent table.

## 2.4 Schemas

PostgreSQL, like most SQL databases, supports the use of schema names to group collections of tables and other database objects together. You might think of schemas as namespaces if you're a programmer; or you might think of the schema / table / column relationship like the area code / exchange / local number structure of a telephone number.

Table 1: Examples: database object names

Full name	Schema name	Table name	Field name
actor.usr_note.title	actor	usr_note	title
biblio.record_entry.marc	biblio	record_entry	marc

The default schema name in PostgreSQL is `public`, so if you do not specify a schema name when creating or accessing a database object, PostgreSQL will use the `public` schema. As a result, you might not find the object that you're looking for if you don't use the appropriate schema.

```
evergreen=# CREATE TABLE foobar (foo TEXT, bar TEXT);
CREATE TABLE
evergreen=# \d foobar
      Table "public.foobar"
  Column | Type | Modifiers
-----+-----+-----
   foo   | text |
   bar   | text |
```

```
evergreen=# SELECT * FROM usr_note;
ERROR:  relation "usr_note" does not exist
LINE 1: SELECT * FROM usr_note;
                        ^
```

Evergreen uses schemas to organize all of its tables with mostly intuitive, if short, schema names. Here's the current (as of 2010-01-03) list of schemas used by Evergreen:

Table 2: Evergreen schema names

Schema name	Description
acq	Acquisitions
action	Circulation actions
action_trigger	Event mechanisms
actor	Evergreen users and organization units
asset	Call numbers and copies
auditor	Track history of changes to selected tables
authority	Authority records
biblio	Bibliographic records
booking	Resource bookings
config	Evergreen configurable options
container	Buckets for records, call numbers, copies, and users
extend_reporter	Extra views for report definitions
metabib	Metadata about bibliographic records
money	Fines and bills
offline	Offline transactions
permission	User permissions
query	Stored SQL statements

Table 2: (continued)

Schema name	Description
reporter	Report definitions
search	Search functions
serial	Serial MFHD records
stats	Convenient views of circulation and asset statistics
vandelay	MARC batch importer and exporter

**Note**

The term *schema* has two meanings in the world of SQL databases. We have discussed the schema as a conceptual grouping of tables and other database objects within a given namespace; for example, "the **actor** schema contains the tables and functions related to users and organizational units". Another common usage of *schema* is to refer to the entire data model for a given database; for example, "the Evergreen database schema".

## 2.5 Columns

Each column definition consists of:

- a data type
- (optionally) a default value to be used whenever a row is inserted that does not contain a specific value
- (optionally) one or more constraints on the values beyond data type

Although PostgreSQL supports dozens of data types, Evergreen makes our life easier by only using a handful.

Table 3: PostgreSQL data types used by Evergreen

Type name	Description	Limits
INTEGER	Medium integer	-2147483648 to +2147483647
BIGINT	Large integer	-9223372036854775808 to 9223372036854775807
SERIAL	Sequential integer	1 to 2147483647
BIGSERIAL	Large sequential integer	1 to 9223372036854775807
TEXT	Variable length character data	Unlimited length
BOOL	Boolean	TRUE or FALSE
TIMESTAMP WITH TIME ZONE	Timestamp	4713 BC to 294276 AD
TIME	Time	Expressed in HH:MM:SS
NUMERIC(precision, scale)	Decimal	Up to 1000 digits of precision. In Evergreen mostly used for money values, with a precision of 6 and a scale of 2 (####.##).

Full details about these data types are available from the [data types section of the PostgreSQL manual](#).

## 2.6 Constraints

### 2.6.1 Prevent NULL values

A column definition may include the constraint `NOT NULL` to prevent NULL values. In PostgreSQL, a NULL value is not the equivalent of zero or false or an empty string; it is an explicit non-value with special properties. We'll talk more about how to work with NULL values when we get to queries.

### 2.6.2 Primary key

Every table can have at most one primary key. A primary key consists of one or more columns which together uniquely identify each row in a table. If you attempt to insert a row into a table that would create a duplicate or NULL primary key entry, the database rejects the row and returns an error.

Natural primary keys are drawn from the intrinsic properties of the data being modelled. For example, some potential natural primary keys for a table that contains people would be:

Table 4: Example: Some potential natural primary keys for a table of people

Natural key	Pros	Cons
First name, last name, address	No two people with the same name would ever live at the same address, right?	Lots of columns force data duplication in referencing tables
SSN or driver's license	These are guaranteed to be unique	Lots of people don't have an SSN or a driver's license

To avoid problems with natural keys, many applications instead define surrogate primary keys. A surrogate primary key is a column with an autoincrementing integer value added to a table definition that ensures uniqueness.

Evergreen uses surrogate keys (a column named `id` with a `SERIAL` data type) for most of its tables.

### 2.6.3 Foreign keys

Every table can contain zero or more foreign keys: one or more columns that refer to the primary key of another table.

For example, let's consider Evergreen's modelling of the basic relationship between copies, call numbers, and bibliographic records. Bibliographic records contained in the `biblio.record_entry` table can have call numbers attached to them. Call numbers are contained in the `asset.call_number` table, and they can have copies attached to them. Copies are contained in the `asset.copy` table.

Table 5: Example: Evergreen's copy / call number / bibliographic record relationships

Table	Primary key	Column with a foreign key	Points to
<code>asset.copy</code>	<code>asset.copy.id</code>	<code>asset.copy.call_number</code>	<code>asset.call_number.id</code>
<code>asset.call_number</code>	<code>asset.call_number.id</code>	<code>asset.call_number.record</code>	<code>biblio.record_entry.id</code>
<code>biblio.record_entry</code>	<code>biblio.record_entry.id</code>		



### 2.6.4 Check constraints

PostgreSQL enables you to define rules to ensure that the value to be inserted or updated meets certain conditions. For example, you can ensure that an incoming integer value is within a specific range, or that a ZIP code matches a particular pattern.

## 2.7 Deconstructing a table definition statement

The `actor.org_address` table is a simple table in the Evergreen schema that we can use as a concrete example of many of the properties of databases that we have discussed so far.

```
CREATE TABLE actor.org_address (
  id          SERIAL PRIMARY KEY,           -- ❶
  valid       BOOL  NOT NULL DEFAULT TRUE, -- ❷
  address_type TEXT NOT NULL DEFAULT 'MAILING', -- ❸
  org_unit    INT   NOT NULL REFERENCES actor.org_unit (id) -- ❹
              DEFERRABLE INITIALLY DEFERRED,
  street1     TEXT  NOT NULL,
  street2     TEXT,                          -- ❺
  city        TEXT  NOT NULL,
  county      TEXT,
  state       TEXT  NOT NULL,
  country     TEXT  NOT NULL,
  post_code   TEXT  NOT NULL
);
```

- ❶ The column named `id` is defined with a special data type of `SERIAL`; if given no value when a row is inserted into a table, the database automatically generates the next sequential integer value for the column. `SERIAL` is a popular data type for a primary key because it is guaranteed to be unique - and indeed, the constraint for this column identifies it as the `PRIMARY KEY`.
- ❷ The data type `BOOL` defines a boolean value: `TRUE` or `FALSE` are the only acceptable values for the column. The constraint `NOT NULL` instructs the database to prevent the column from ever containing a `NULL` value. The column property `DEFAULT TRUE` instructs the database to automatically set the value of the column to `TRUE` if no value is provided.
- ❸ The data type `TEXT` defines a text column of practically unlimited length. As with the previous column, there is a `NOT NULL` constraint, and a default value of `'MAILING'` will result if no other value is supplied.
- ❹ The `REFERENCES actor.org_unit (id)` clause indicates that this column has a foreign key relationship to the `actor.org_unit` table, and that the value of this column in every row in this table must have a corresponding value in the `id` column in the referenced table (`actor.org_unit`).
- ❺ The column named `street2` demonstrates that not all columns have constraints beyond data type. In this case, the column is allowed to be `NULL` or to contain a `TEXT` value.

## 2.8 Displaying a table definition using `psql`

The `psql` command-line interface is the preferred method for accessing PostgreSQL databases. It offers features like tab-completion, readline support for recalling previous commands, flexible input and output formats, and is accessible via a standard SSH session.

If you press the `Tab` key once after typing one or more characters of the database object name, `psql` automatically completes the name if there are no other matches. If there are other matches for your current input, nothing happens until you press the `Tab` key a second time, at which point `psql` displays all of the matches for your current input.

To display the definition of a database object such as a table, issue the command `\d _object-name_`. For example, to display the definition of the `actor.usr_note` table:

```

$ psql evergreen # ❶
psql (8.4.1)
Type "help" for help.

evergreen=# \d actor.usr_note # ❷
          Table "actor.usr_note"
   Column |          Type          | Modifiers
-----+-----+-----
 id      | bigint                 | not null default nextval('actor.usr_note_id_seq'::regclass)
 usr     | bigint                 | not null
 creator | bigint                 | not null
 create_date | timestamp with time zone | default now()
 pub     | boolean                | not null default false
 title   | text                   | not null
 value   | text                   | not null
Indexes:
    "usr_note_pkey" PRIMARY KEY, btree (id)
    "actor_usr_note_creator_idx" btree (creator)
    "actor_usr_note_usr_idx" btree (usr)
Foreign-key constraints:
    "usr_note_creator_fkey" FOREIGN KEY (creator) REFERENCES actor.usr(id) ON DELETE CASCADE DEFERRABLE INITIALLY DEFERRED
    "usr_note_usr_fkey" FOREIGN KEY (usr) REFERENCES actor.usr(id) ON DELETE CASCADE DEFERRABLE INITIALLY DEFERRED

evergreen=# \q # ❸
$

```

- ❶ This is the most basic connection to a PostgreSQL database. You can use a number of other flags to specify user name, hostname, port, and other options.
- ❷ The `\d` command displays the definition of a database object.
- ❸ The `\q` command quits the `psql` session and returns you to the shell prompt.

## 3 Part 2: Basic SQL queries

### 3.1 Learning objectives

- Understand how to select specific columns from one table
- Understand how to filter results with the WHERE clause
- Understand how to specify the order of results using the ORDER BY clause
- Understand how to group and eliminate results with the GROUP BY and HAVING clauses
- Understand how to restrict the number of results using the LIMIT clause

### 3.2 The SELECT statement

The SELECT statement is the basic tool for retrieving information from a database. The syntax for most SELECT statements is:

```
SELECT [columns(s)]  
FROM [table(s)]  
  [WHERE condition(s)]  
  [GROUP BY columns(s)]  
  [HAVING grouping-condition(s)]  
  [ORDER BY column(s)]  
  [LIMIT maximum-results]  
  [OFFSET start-at-result-#]  
;
```

For example, to select all of the columns for each row in the `actor.usr_address` table, issue the following query:

```
SELECT *  
  FROM actor.usr_address  
;
```

### 3.3 Selecting particular columns from a table

`SELECT *` returns all columns from all of the tables included in your query. However, quite often you will want to return only a subset of the possible columns. You can retrieve specific columns by listing the names of the columns you want after the `SELECT` keyword. Separate each column name with a comma.

For example, to select just the city, county, and state from the `actor.usr_address` table, issue the following query:

```
SELECT city, county, state  
  FROM actor.usr_address  
;
```

### 3.4 Sorting results with the ORDER BY clause

By default, a `SELECT` statement returns rows matching your query with no guarantee of any particular order in which they are returned. To force the rows to be returned in a particular order, use the `ORDER BY` clause to specify one or more columns to determine the sorting priority of the rows.

For example, to sort the rows returned from your `actor.usr_address` query by city, with county and then zip code as the tie breakers, issue the following query:

```
SELECT city, county, state  
  FROM actor.usr_address  
  ORDER BY city, county, post_code  
;
```

### 3.5 Filtering results with the WHERE clause

Thus far, your results have been returning all of the rows in the table. Normally, however, you would want to restrict the rows that are returned to the subset of rows that match one or more conditions of your search. The `WHERE` clause enables you to specify a set of conditions that filter your query results. Each condition in the `WHERE` clause is an SQL expression that returns a boolean (true or false) value.

For example, to restrict the results returned from your `actor.usr_address` query to only those rows containing a state value of *Connecticut*, issue the following query:

```
SELECT city, county, state  
  FROM actor.usr_address  
  WHERE state = 'Connecticut'  
  ORDER BY city, county, post_code  
;
```

You can include more conditions in the `WHERE` clause with the `OR` and `AND` operators. For example, to further restrict the results returned from your `actor.usr_address` query to only those rows where the state column contains a value of *Connecticut* and the city column contains a value of *Hartford*, issue the following query:

```
SELECT city, county, state
FROM actor.usr_address
WHERE state = 'Connecticut'
      AND city = 'Hartford'
ORDER BY city, county, post_code
;
```

---

**Note**

To return rows where the state is *Connecticut* and the city is *Hartford* or *New Haven*, you must use parentheses to explicitly group the city value conditions together, or else the database will evaluate the `OR city = 'New Haven'` clause entirely on its own and match all rows where the city column is *New Haven*, even though the state might not be *Connecticut*.

---

```
SELECT city, county, state
FROM actor.usr_address
WHERE state = 'Connecticut'
      AND city = 'Hartford' OR city = 'New Haven'
ORDER BY city, county, post_code
;
```

```
-- Can return unwanted rows because the OR is not grouped!
```

```
SELECT city, county, state
FROM actor.usr_address
WHERE state = 'Connecticut'
      AND (city = 'Hartford' OR city = 'New Haven')
ORDER BY city, county, post_code
;
```

```
-- The parentheses ensure that the OR is applied to the cities, and the
-- state in either case must be 'Connecticut'
```

### 3.5.1 Comparison operators

Here is a partial list of comparison operators that are commonly used in `WHERE` clauses:

#### Comparing two scalar values

- `x = y` (equal to)
  - `x != y` (not equal to)
  - `x < y` (less than)
  - `x > y` (greater than)
  - `x LIKE y` (TEXT value `x` matches a subset of TEXT `y`, where `y` is a string that can contain `%` as a wildcard for 0 or more characters, and `_` as a wildcard for a single character. For example, `WHERE 'all you can eat fish and chips and a big stick' LIKE '%fish%stick'` would return `TRUE`)
  - `x ILIKE y` (like `LIKE`, but the comparison ignores upper-case / lower-case)
  - `x IN y` (`x` is in the list of values `y`, where `y` can be a list or a `SELECT` statement that returns a list)
-

### 3.6 NULL values

SQL databases have a special way of representing the value of a column that has no value: NULL. A NULL value is not equal to zero, and is not an empty string; it is equal to nothing, not even another NULL, because it has no value that can be compared.

To return rows from a table where a given column is not NULL, use the `IS NOT NULL` comparison operator.

```
SELECT id, first_given_name, family_name
FROM actor.usr
WHERE second_given_name IS NOT NULL
;
```

Similarly, to return rows from a table where a given column is NULL, use the `IS NULL` comparison operator.

```
SELECT id, first_given_name, second_given_name, family_name
FROM actor.usr
WHERE second_given_name IS NULL
;
```

id	first_given_name	second_given_name	family_name
1	Administrator		System Account

(1 row)

Notice that the NULL value in the output is displayed as empty space, indistinguishable from an empty string; this is the default display method in `psql`. You can change the behaviour of `psql` using the `pset` command:

```
evergreen=# \pset null '(null)'
Null display is '(null)'.

SELECT id, first_given_name, second_given_name, family_name
FROM actor.usr
WHERE second_given_name IS NULL
;
```

id	first_given_name	second_given_name	family_name
1	Administrator	(null)	System Account

(1 row)

Database queries within programming languages such as Perl and C have special methods of checking for NULL values in returned results.

### 3.7 Text delimiter: '

You might have noticed that we have been using the `'` character to delimit TEXT values and values such as dates and times that are TEXT values. Sometimes, however, your TEXT value itself contains a `'` character, such as the word `you're`. To prevent the database from prematurely ending the TEXT value at the first `'` character and returning a syntax error, use another `'` character to escape the following `'` character.

For example, to change the last name of a user in the `actor.usr` table to `L'estat`, issue the following SQL:

```
UPDATE actor.usr
SET family_name = 'L'estat'
WHERE profile IN (
SELECT id
FROM permission.grp_tree
WHERE name = 'Vampire'
)
;
```

When you retrieve the row from the database, the value is displayed with just a single ' character:

```
SELECT id, family_name
FROM actor.usr
WHERE family_name = 'L'estat'
;

id | family_name
-----+-----
 1 | L'estat
(1 row)
```

### 3.8 Grouping and eliminating results with the GROUP BY and HAVING clauses

The GROUP BY clause returns a unique set of results for the desired columns. This is most often used in conjunction with an aggregate function to present results for a range of values in a single query, rather than requiring you to issue one query per target value.

```
SELECT grp
FROM permission.grp_perm_map
GROUP BY grp
ORDER BY grp;

grp
-----+
 1
 2
 3
 4
 5
 6
 7
10
(8 rows)
```

While GROUP BY can be useful for a single column, it is more often used to return the distinct results across multiple columns. For example, the following query shows us which groups have permissions at each depth in the library hierarchy:

```
SELECT grp, depth
FROM permission.grp_perm_map
GROUP BY grp, depth
ORDER BY depth, grp;

grp | depth
-----+-----
 1 | 0
 2 | 0
 3 | 0
 4 | 0
 5 | 0
10 | 0
 3 | 1
 4 | 1
 5 | 1
 6 | 1
 7 | 1
10 | 1
 3 | 2
 4 | 2
10 | 2
(15 rows)
```

Extending this further, you can use the `COUNT ()` aggregate function to also return the number of times each unique combination of `grp` and `depth` appears in the table. *Yes, this is a sneak peek at the use of aggregate functions! Keeners.*

```
SELECT grp, depth, COUNT(grp)
FROM permission.grp_perm_map
GROUP BY grp, depth
ORDER BY depth, grp;
```

grp	depth	count
1	0	6
2	0	2
3	0	45
4	0	3
5	0	5
10	0	1
3	1	3
4	1	4
5	1	1
6	1	9
7	1	5
10	1	10
3	2	24
4	2	8
10	2	7

(15 rows)

You can use the `WHERE` clause to restrict the returned results before grouping is applied to the results. The following query restricts the results to those rows that have a depth of 0.

```
SELECT grp, COUNT(grp)
FROM permission.grp_perm_map
WHERE depth = 0
GROUP BY grp
ORDER BY 2 DESC
;
```

grp	count
3	45
1	6
5	5
4	3
2	2
10	1

(6 rows)

To restrict results after grouping has been applied to the rows, use the `HAVING` clause; this is typically used to restrict results based on a comparison to the value returned by an aggregate function. For example, the following query restricts the returned rows to those that have more than 5 occurrences of the same value for `grp` in the table.

```
SELECT grp, COUNT(grp)
FROM permission.grp_perm_map
GROUP BY grp
HAVING COUNT(grp) > 5
;
```

grp	count
6	9
4	15
5	6

```
 1 |      6
 3 |     72
10 |     18
(6 rows)
```

### 3.9 Eliminating duplicate results with the DISTINCT keyword

GROUP BY is one way of eliminating duplicate results from the rows returned by your query. The purpose of the DISTINCT keyword is to remove duplicate rows from the results of your query. However, it works, and it is easy - so if you just want a quick list of the unique set of values for a column or set of columns, the DISTINCT keyword might be appropriate.

On the other hand, if you are getting duplicate rows back when you don't expect them, then applying the DISTINCT keyword might be a sign that you are papering over a real problem.

```
SELECT DISTINCT grp, depth
FROM permission.grp_perm_map
ORDER BY depth, grp
```

```
;
```

```
grp | depth
-----+-----
 1 |      0
 2 |      0
 3 |      0
 4 |      0
 5 |      0
10 |      0
 3 |      1
 4 |      1
 5 |      1
 6 |      1
 7 |      1
10 |      1
 3 |      2
 4 |      2
10 |      2
(15 rows)
```

### 3.10 Paging through results with the LIMIT and OFFSET clauses

The LIMIT clause restricts the total number of rows returned from your query and is useful if you just want to list a subset of a large number of rows. For example, in the following query we list the five most frequently used circulation modifiers:

```
SELECT circ_modifier, COUNT(circ_modifier)
FROM asset.copy
GROUP BY circ_modifier
ORDER BY 2 DESC
LIMIT 5
```

```
;
```

```
circ_modifier | count
-----+-----
CIRC          | 741995
BOOK          | 636199
SER           | 265906
DOC           | 191598
LAW MONO     | 126627
(5 rows)
```



When you use the `LIMIT` clause to restrict the total number of rows returned by your query, you can also use the `OFFSET` clause to determine which subset of the rows will be returned. The use of the `OFFSET` clause assumes that you've used the `ORDER BY` clause to impose order on the results.

In the following example, we use the `OFFSET` clause to get results 6 through 10 from the same query that we previously executed.

```
SELECT circ_modifier, COUNT(circ_modifier)
FROM asset.copy
GROUP BY circ_modifier
ORDER BY 2 DESC
LIMIT 5
OFFSET 5
;
```

circ_modifier	count
LAW SERIAL	102758
DOCUMENTS	86215
BOOK_WEB	63786
MFORM SER	39917
REF	34380

(5 rows)

## 4 Part 3: Advanced SQL queries

### 4.1 Learning objectives

- Understand the difference between scalar functions and aggregate functions
- Know how to use functions to transform column values for comparison and return values
- Know how to retrieve values from across multiple tables
- Understand the difference between inner joins, outer joins, unions, and intersections
- Know how to create a view to simplify frequently used queries

Thus far you've been working with a single table at a time - and you have been able accomplish a great deal. However, relational databases by their nature spread data across multiple tables, so it is important to learn how to bring that data back together in your queries. In addition, real data in the wild often requires taming by transforming it to different states so that you can, for example, compare values more efficiently, or reduce the number of results, or find the maximum value in a set of results.

### 4.2 Transforming column values with functions

PostgreSQL includes many built-in functions for manipulating column data. You can also create your own functions (and Evergreen does make use of many custom functions). There are two types of functions used in databases: scalar functions and aggregate functions.

#### 4.2.1 Scalar functions

Scalar functions transform each value of the target column. If your query would return 50 values for a column in a given query, and you modify your query to apply a scalar function to the values returned for that column, it will still return 50 values. For example, the `UPPER()` function, used to convert text values to upper-case, modifies the results in the following set of queries:

```
-- First, without the UPPER() function for comparison
SELECT shortname, name
  FROM actor.org_unit
 WHERE id < 4
;

shortname |      name
-----+-----
CONS      | Example Consortium
SYS1      | Example System 1
SYS2      | Example System 2
(3 rows)

-- Now apply the UPPER() function to the name column
SELECT shortname, UPPER(name)
  FROM actor.org_unit
 WHERE id < 4
;

shortname |      upper
-----+-----
CONS      | EXAMPLE CONSORTIUM
SYS1      | EXAMPLE SYSTEM 1
SYS2      | EXAMPLE SYSTEM 2
(3 rows)
```

There are so many scalar functions in PostgreSQL that we cannot cover them all here, but we can list some of the most commonly used functions:

- `||` - concatenates two text values together
- `COALESCE()` - returns the first non-NULL value from the list of arguments
- `LOWER()` - returns a text value converted to lower-case
- `REPLACE()` - returns a text value after replacing all occurrences of a given text value with a different text value
- `REGEXP_REPLACE()` - returns a text value after being transformed by a regular expression
- `UPPER()` - returns a text value converted to upper-case

For a complete list of scalar functions, see [the PostgreSQL function documentation](#).

#### 4.2.2 Aggregate functions

Aggregate functions return a single value computed from the the complete set of values returned for the specified column.

- `AVG()`
- `COUNT()`
- `MAX()`
- `MIN()`
- `SUM()`

### 4.3 Sub-selects

A sub-select is the technique of using the results of one query to feed into another query. You can, for example, return a set of values from one column in a SELECT statement to be used to satisfy the IN() condition of another SELECT statement; or you could return the MAX() value of a column in a SELECT statement to match the = condition of another SELECT statement.

For example, in the following query we use a sub-select to restrict the copies returned by the main SELECT statement to only those locations that have an `opac_visible` value of TRUE:

```
SELECT call_number
FROM asset.copy
WHERE deleted IS FALSE
AND location IN (
  SELECT id
  FROM asset.copy_location
  WHERE opac_visible IS TRUE
)
;
```

Sub-selects can be an approachable way to breaking down a problem that requires matching values between different tables, and often result in a clearly expressed solution to a problem. However, if you start writing sub-selects within sub-selects, you should consider tackling the problem with joins instead.

### 4.4 Joins

Joins enable you to access the values from multiple tables in your query results and comparison operators. For example, joins are what enable you to relate a bibliographic record to a barcoded copy via the `biblio.record_entry`, `asset.call_number`, and `asset.copy` tables. In this section, we discuss the most common kind of join—the inner join—as well as the less common outer join and some set operations which can compare and contrast the values returned by separate queries.

When we talk about joins, we are going to talk about the left-hand table and the right-hand table that participate in the join. Every join brings together just two tables - but you can use an unlimited (for our purposes) number of joins in a single SQL statement. Each time you use a join, you effectively create a new table, so when you add a second join clause to a statement, table 1 and table 2 (which were the left-hand table and the right-hand table for the first join) now act as a merged left-hand table and the new table in the second join clause is the right-hand table.

Clear as mud? Okay, let's look at some examples.

#### 4.4.1 Inner joins

An inner join returns all of the columns from the left-hand table in the join with all of the columns from the right-hand table in the joins that match a condition in the ON clause. Typically, you use the = operator to match the foreign key of the left-hand table with the primary key of the right-hand table to follow the natural relationship between the tables.

In the following example, we return all of columns from the `actor.usr` and `actor.org_unit` tables, joined on the relationship between the user's home library and the library's ID. Notice in the results that some columns, like `id` and `mailing_address`, appear twice; this is because both the `actor.usr` and `actor.org_unit` tables include columns with these names. This is also why we have to fully qualify the column names in our queries with the schema and table names.

```
SELECT *
FROM actor.usr
INNER JOIN actor.org_unit ON actor.usr.home_ou = actor.org_unit.id
WHERE actor.org_unit.shortname = 'CONS'
;
```

```
-[ RECORD 1 ]-----+-----
id              | 1
card            | 1
profile         | 1
usrname        | admin
```

```

email |
...
mailing_address |
billing_address |
home_ou | 1
...
claims_never_checked_out_count | 0
id | 1
parent_ou |
ou_type | 1
ill_address | 1
holds_address | 1
mailing_address | 1
billing_address | 1
shortname | CONS
name | Example Consortium
email |
phone |
opac_visible | t
fiscal_calendar | 1

```

Of course, you do not have to return every column from the joined tables; you can (and should) continue to specify only the columns that you want to return. In the following example, we count the number of borrowers for every user profile in a given library by joining the `permission.grp_tree` table where profiles are defined against the `actor.usr` table, and then joining the `actor.org_unit` table to give us access to the user's home library:

```

SELECT permission.grp_tree.name, actor.org_unit.name, COUNT(permission.grp_tree.name)
FROM actor.usr
    INNER JOIN permission.grp_tree
        ON actor.usr.profile = permission.grp_tree.id
    INNER JOIN actor.org_unit
        ON actor.org_unit.id = actor.usr.home_ou
WHERE actor.usr.deleted IS FALSE
GROUP BY permission.grp_tree.name, actor.org_unit.name
ORDER BY actor.org_unit.name, permission.grp_tree.name
;

```

```

name | name | count
-----+-----+-----
Users | Example Consortium | 1
(1 row)

```

#### 4.4.2 Aliases

So far we have been fully-qualifying all of our table names and column names to prevent any confusion. This quickly gets tiring with lengthy qualified table names like `permission.grp_tree`, so the SQL syntax enables us to assign aliases to table names and column names. When you define an alias for a table name, you can access its column throughout the rest of the statement by simply appending the column name to the alias with a period; for example, if you assign the alias `au` to the `actor.usr` table, you can access the `actor.usr.id` column through the alias as `au.id`.

The formal syntax for declaring an alias for a column is to follow the column name in the result columns clause with `AS alias`. To declare an alias for a table name, follow the table name in the `FROM` clause (including any `JOIN` statements) with `AS alias`. However, the `AS` keyword is optional for tables (and columns as of PostgreSQL 8.4), and in practice most SQL statements leave it out. For example, we can write the previous `INNER JOIN` statement example using aliases instead of fully-qualified identifiers:

```

SELECT pgt.name AS "Profile", aou.name AS "Library", COUNT(pgt.name) AS "Count"
FROM actor.usr au
    INNER JOIN permission.grp_tree pgt
        ON au.profile = pgt.id
    INNER JOIN actor.org_unit aou

```

```

    ON aou.id = au.home_ou
WHERE au.deleted IS FALSE
GROUP BY pgt.name, aou.name
ORDER BY aou.name, pgt.name
;

```

Profile	Library	Count
Users	Example Consortium	1

(1 row)

A nice side effect of declaring an alias for your columns is that the alias is used as the column header in the results table. The previous version of the query, which didn't use aliased column names, had two columns named name; this version of the query with aliases results in a clearer categorization.

#### 4.4.3 Outer joins

An outer join returns all of the rows from one or both of the tables participating in the join.

- For a LEFT OUTER JOIN, the join returns all of the rows from the left-hand table and the rows matching the join condition from the right-hand table, with NULL values for the rows with no match in the right-hand table.
- A RIGHT OUTER JOIN behaves in the same way as a LEFT OUTER JOIN, with the exception that all rows are returned from the right-hand table participating in the join.
- For a FULL OUTER JOIN, the join returns all the rows from both the left-hand and right-hand tables, with NULL values for the rows with no match in either the left-hand or right-hand table.

```
SELECT * FROM aaa;
```

id	stuff
1	one
2	two
3	three
4	four
5	five

(5 rows)

```
SELECT * FROM bbb;
```

id	stuff	foo
1	one	oneone
2	two	twotwo
5	five	fivefive
6	six	sixsix

(4 rows)

```
SELECT * FROM aaa
LEFT OUTER JOIN bbb ON aaa.id = bbb.id
```

```
;
```

id	stuff	id	stuff	foo
1	one	1	one	oneone
2	two	2	two	twotwo
3	three			
4	four			
5	five	5	five	fivefive

(5 rows)

```
SELECT * FROM aaa
  RIGHT OUTER JOIN bbb ON aaa.id = bbb.id
;
id | stuff | id | stuff |  foo
-----+-----+-----+-----+-----
 1 | one   | 1 | one   | oneone
 2 | two   | 2 | two   | twotwo
 5 | five  | 5 | five  | fivefive
   |      | 6 | six   | sixsix
(4 rows)
```

```
SELECT * FROM aaa
  FULL OUTER JOIN bbb ON aaa.id = bbb.id
;
id | stuff | id | stuff |  foo
-----+-----+-----+-----+-----
 1 | one   | 1 | one   | oneone
 2 | two   | 2 | two   | twotwo
 3 | three |   |      |
 4 | four  |   |      |
 5 | five  | 5 | five  | fivefive
   |      | 6 | six   | sixsix
(6 rows)
```

#### 4.4.4 Self joins

It is possible to join a table to itself. You can, in fact you must, use aliases to disambiguate the references to the table.

## 4.5 Set operations

Relational databases are effectively just an efficient mechanism for manipulating sets of values; they are implementations of set theory. There are three operators for sets (tables) in which each set must have the same number of columns with compatible data types: the union, intersection, and difference operators.

```
SELECT * FROM aaa;
```

```
id | stuff
-----+-----
 1 | one
 2 | two
 3 | three
 4 | four
 5 | five
(5 rows)
```

```
SELECT * FROM bbb;
```

```
id | stuff |  foo
-----+-----+-----
 1 | one   | oneone
 2 | two   | twotwo
 5 | five  | fivefive
 6 | six   | sixsix
(4 rows)
```

### 4.5.1 Union

The UNION operator returns the distinct set of rows that are members of either or both of the left-hand and right-hand tables. The UNION operator does not return any duplicate rows. To return duplicate rows, use the UNION ALL operator.

```
-- The parentheses are not required, but are intended to help
-- illustrate the sets participating in the set operation
(
  SELECT id, stuff
    FROM aaa
)
UNION
(
  SELECT id, stuff
    FROM bbb
)
ORDER BY 1
;
```

id	stuff
1	one
2	two
3	three
4	four
5	five
6	six

(6 rows)

### 4.5.2 Intersection

The INTERSECT operator returns the distinct set of rows that are common to both the left-hand and right-hand tables. To return duplicate rows, use the INTERSECT ALL operator.

```
(
  SELECT id, stuff
    FROM aaa
)
INTERSECT
(
  SELECT id, stuff
    FROM bbb
)
ORDER BY 1
;
```

id	stuff
1	one
2	two
5	five

(3 rows)

### 4.5.3 Difference

The EXCEPT operator returns the rows in the left-hand table that do not exist in the right-hand table. You are effectively subtracting the common rows from the left-hand table.

```
(
  SELECT id, stuff
    FROM aaa
)
EXCEPT
(
  SELECT id, stuff
    FROM bbb
)
ORDER BY 1
;

id | stuff
----+-----
 3 | three
 4 | four
(2 rows)

-- Order matters: switch the left-hand and right-hand tables
-- and you get a different result
(
  SELECT id, stuff
    FROM bbb
)
EXCEPT
(
  SELECT id, stuff
    FROM aaa
)
ORDER BY 1
;

id | stuff
----+-----
 6 | six
(1 row)
```

## 4.6 Views

A view is a persistent `SELECT` statement that acts like a read-only table. To create a view, issue the `CREATE VIEW` statement, giving the view a name and a `SELECT` statement on which the view is built.

The following example creates a view based on our borrower profile count:

```
CREATE VIEW actor.borrower_profile_count AS
  SELECT pgt.name AS "Profile", aou.name AS "Library", COUNT(pgt.name) AS "Count"
  FROM actor.usr au
    INNER JOIN permission.grp_tree pgt
      ON au.profile = pgt.id
    INNER JOIN actor.org_unit aou
      ON aou.id = au.home_ou
  WHERE au.deleted IS FALSE
  GROUP BY pgt.name, aou.name
  ORDER BY aou.name, pgt.name
;
```

When you subsequently select results from the view, you can apply additional `WHERE` clauses to filter the results, or `ORDER BY` clauses to change the order of the returned rows. In the following examples, we issue a simple `SELECT *` statement to show that the default results are returned in the same order from the view as the equivalent `SELECT` statement would be returned. Then we issue a `SELECT` statement with a `WHERE` clause to further filter the results.



```
SELECT * FROM actor.borrower_profile_count;
```

Profile	Library	Count
Faculty	University Library	208
Graduate	University Library	16
Patrons	University Library	62
...		

```
-- You can still filter your results with WHERE clauses
```

```
SELECT *
FROM actor.borrower_profile_count
WHERE "Profile" = 'Faculty';
```

Profile	Library	Count
Faculty	University Library	208
Faculty	College Library	64
Faculty	College Library 2	102
Faculty	University Library 2	776

(4 rows)

## 4.7 Inheritance

PostgreSQL supports table inheritance: that is, a child table inherits its base definition from a parent table, but can add additional columns to its own definition. The data from any child tables is visible in queries against the parent table.

Evergreen uses table inheritance in several areas: \* In the Vandelay MARC batch importer / exporter, Evergreen defines base tables for generic queues and queued records for which authority record and bibliographic record child tables \* Billable transactions are based on the `money.billable_xact` table; child tables include `action.circulation` for circulation transactions and `money.grocery` for general bills. \* Payments are based on the `money.payment` table; its child table is `money.bnm_payment` (for brick-and-mortar payments), which in turn has child tables of `money.forgive_payment`, `money.work_payment`, `money.credit_payment`, `money.goods_payment`, and `money.bnm_desk_payment`. The `money.bnm_desk_payment` table in turn has child tables of `money.cash_payment`, `money.check_payment`, and `money.credit_card_payment`. \* Transits are based on the `action.transit_copy` table, which has a child table of `action.hold_transit_copy` for transits initiated by holds. \* Generic acquisition line items are defined by the `acq.lineitem_attr_definition` table, which in turn has a number of child tables to define MARC attributes, generated attributes, user attributes, and provider attributes.

## 5 Part 4: Understanding query performance with EXPLAIN

Some queries run for a long, long time. This can be the result of a poorly written query—a query with a join condition that joins every row in the `biblio.record_entry` table with every row in the `metabib.full_rec` view would consume a massive amount of memory and disk space and CPU time—or a symptom of a schema that needs some additional indexes. PostgreSQL provides the `EXPLAIN` tool to estimate how long it will take to run a given query and show you the *query plan* (how it plans to retrieve the results from the database).

To generate the query plan without actually running the statement, simply prepend the `EXPLAIN` keyword to your query. In the following example, we generate the query plan for the poorly written query that would join every row in the `biblio.record_entry` table with every row in the `metabib.full_rec` view:

```
EXPLAIN SELECT *
FROM biblio.record_entry
FULL OUTER JOIN metabib.full_rec ON 1=1
;
```

QUERY PLAN

```
-----//
Merge Full Join (cost=0.00..4959156437783.60 rows=132415734100864 width=1379)
-> Seq Scan on record_entry (cost=0.00..400634.16 rows=2013416 width=1292)
-> Seq Scan on real_full_rec (cost=0.00..1640972.04 rows=65766704 width=87)
(3 rows)
```

This query plan shows that the query would return 132415734100864 rows, and it plans to accomplish what you asked for by sequentially scanning (*Seq Scan*) every row in each of the tables participating in the join.

In the following example, we have realized our mistake in joining every row of the left-hand table with every row in the right-hand table and take the saner approach of using an `INNER JOIN` where the join condition is on the record ID.

```
EXPLAIN SELECT *
FROM biblio.record_entry bre
INNER JOIN metabib.full_rec mfr ON mfr.record = bre.id;
QUERY PLAN
-----//
Hash Join (cost=750229.86..5829273.98 rows=65766704 width=1379)
Hash Cond: (real_full_rec.record = bre.id)
-> Seq Scan on real_full_rec (cost=0.00..1640972.04 rows=65766704 width=87)
-> Hash (cost=400634.16..400634.16 rows=2013416 width=1292)
-> Seq Scan on record_entry bre (cost=0.00..400634.16 rows=2013416 width=1292)
(5 rows)
```

This time, we will return 65766704 rows - still way too many rows. We forgot to include a `WHERE` clause to limit the results to something meaningful. In the following example, we will limit the results to deleted records that were modified in the last month.

```
EXPLAIN SELECT *
FROM biblio.record_entry bre
INNER JOIN metabib.full_rec mfr ON mfr.record = bre.id
WHERE bre.deleted IS TRUE
AND DATE_TRUNC('MONTH', bre.edit_date) >
DATE_TRUNC ('MONTH', NOW() - '1 MONTH'::INTERVAL)
;
QUERY PLAN
-----//
Hash Join (cost=5058.86..2306218.81 rows=201669 width=1379)
Hash Cond: (real_full_rec.record = bre.id)
-> Seq Scan on real_full_rec (cost=0.00..1640972.04 rows=65766704 width=87)
-> Hash (cost=4981.69..4981.69 rows=6174 width=1292)
-> Index Scan using biblio_record_entry_deleted on record_entry bre
(cost=0.00..4981.69 rows=6174 width=1292)
Index Cond: (deleted = true)
Filter: ((deleted IS TRUE) AND (date_trunc('MONTH'::text, edit_date)
> date_trunc('MONTH'::text, (now() - '1 mon'::interval))))
(7 rows)
```

We can see that the number of rows returned is now only 201669; that's something we can work with. Also, the overall cost of the query is 2306218, compared to 4959156437783 in the original query. The `Index Scan` tells us that the query planner will use the index that was defined on the `deleted` column to avoid having to check every row in the `biblio.record_entry` table.

However, we are still running a sequential scan over the `metabib.real_full_rec` table (the table on which the `metabib.full_rec` view is based). Given that linking from the bibliographic records to the flattened MARC subfields is a fairly common operation, we could create a new index and see if that speeds up our query plan.

```
-- This index will take a long time to create on a large database
-- of bibliographic records
CREATE INDEX bib_record_idx ON metabib.real_full_rec (record);

EXPLAIN SELECT *
```

```
FROM biblio.record_entry bre
  INNER JOIN metabib.full_rec mfr ON mfr.record = bre.id
WHERE bre.deleted IS TRUE
  AND DATE_TRUNC('MONTH', bre.edit_date) >
      DATE_TRUNC ('MONTH', NOW() - '1 MONTH'::INTERVAL)
;

                                QUERY PLAN
-----
Nested Loop  (cost=0.00..1558330.46 rows=201669 width=1379)
->  Index Scan using biblio_record_entry_deleted on record_entry bre
    (cost=0.00..4981.69 rows=6174 width=1292)
    Index Cond: (deleted = true)
    Filter: ((deleted IS TRUE) AND (date_trunc('MONTH'::text, edit_date) >
      date_trunc('MONTH'::text, (now() - '1 mon'::interval))))
->  Index Scan using bib_record_idx on real_full_rec
    (cost=0.00..240.89 rows=850 width=87)
    Index Cond: (real_full_rec.record = bre.id)
(6 rows)
```

We can see that the resulting number of rows is still the same (201669), but the execution estimate has dropped to 1558330 because the query planner can use the new index (`bib_record_idx`) rather than scanning the entire table. Success!

---

**Note**

While indexes can significantly speed up read access to tables for common filtering conditions, every time a row is created or updated the corresponding indexes also need to be maintained - which can decrease the performance of writes to the database. Be careful to keep the balance of read performance versus write performance in mind if you plan to create custom indexes in your Evergreen database.

---

## 6 Part 5: Inserting, updating, and deleting data

### 6.1 Inserting data

To insert one or more rows into a table, use the `INSERT` statement to identify the target table and list the columns in the table for which you are going to provide values for each row. If you do not list one or more columns contained in the table, the database will automatically supply a `NULL` value for those columns. The values for each row follow the `VALUES` clause and are grouped in parentheses and delimited by commas. Each row, in turn, is delimited by commas (*this multiple row syntax requires PostgreSQL 8.2 or higher*).

For example, to insert two rows into the `permission.usr_grp_map` table:

```
INSERT INTO permission.usr_grp_map (usr, grp)
VALUES (2, 10), (2, 4)
;
```

Of course, as with the rest of SQL, you can replace individual column values with one or more use sub-selects:

```
INSERT INTO permission.usr_grp_map (usr, grp)
VALUES (
  (SELECT id FROM actor.usr
   WHERE family_name = 'Scott' AND first_given_name = 'Daniel'),
  (SELECT id FROM permission.grp_tree
   WHERE name = 'Local System Administrator')
), (
  (SELECT id FROM actor.usr
   WHERE family_name = 'Scott' AND first_given_name = 'Daniel'),
  (SELECT id FROM permission.grp_tree
```

```
WHERE name = 'Circulator')
)
;
```

## 6.2 Inserting data using a SELECT statement

Sometimes you want to insert a bulk set of data into a new table based on a query result. Rather than a `VALUES` clause, you can use a `SELECT` statement to insert one or more rows matching the column definitions. This is a good time to point out that you can include explicit values, instead of just column identifiers, in the return columns of the `SELECT` statement. The explicit values are returned in every row of the result set.

In the following example, we insert 6 rows into the `permission.usr_grp_map` table; each row will have a `usr` column value of 1, with varying values for the `grp` column value based on the `id` column values returned from `permission.grp_tree`:

```
INSERT INTO permission.usr_grp_map (usr, grp)
SELECT 1, id
FROM permission.grp_tree
WHERE id > 2
;

INSERT 0 6
```

## 6.3 Deleting rows

Deleting data from a table is normally fairly easy. To delete rows from a table, issue a `DELETE` statement identifying the table from which you want to delete rows and a `WHERE` clause identifying the row or rows that should be deleted.

In the following example, we delete all of the rows from the `permission.grp_perm_map` table where the permission maps to `UPDATE_ORG_UNIT_CLOSING` and the group is anything other than administrators:

```
DELETE FROM permission.grp_perm_map
WHERE grp IN (
SELECT id
FROM permission.grp_tree
WHERE name != 'Local System Administrator'
) AND perm = (
SELECT id
FROM permission.perm_list
WHERE code = 'UPDATE_ORG_UNIT_CLOSING'
)
;
```

---

### Note

There are two main reasons that a `DELETE` statement may not actually delete rows from a table, even when the rows meet the conditional clause.

1. If the row contains a value that is the target of a relational constraint, for example, if another table has a foreign key pointing at your target table, you will be prevented from deleting a row with a value corresponding to a row in the dependent table.
  2. If the table has a rule that substitutes a different action for a `DELETE` statement, the deletion will not take place. In Evergreen it is common for a table to have a rule that substitutes the action of setting a `deleted` column to `TRUE`. For example, if a book is discarded, deleting the row representing the copy from the `asset.copy` table would severely affect circulation statistics, bills, borrowing histories, and their corresponding tables in the database that have foreign keys pointing at the `asset.copy` table (`action.circulation` and `money.billing` and its children respectively). Instead, the `deleted` column value is set to `TRUE` and Evergreen's application logic skips over these rows in most cases.
-

## 6.4 Updating rows

To update rows in a table, issue an `UPDATE` statement identifying the table you want to update, the column or columns that you want to set with their respective new values, and (optionally) a `WHERE` clause identifying the row or rows that should be updated.

Following is the syntax for the `UPDATE` statement:

```
UPDATE [table-name]
SET [column] TO [new-value]
WHERE [condition]
;
```

## 7 Part 6: Query requests

The following queries were requested by Bibliomation, but might be reusable by other libraries.

### 7.1 Monthly circulation stats by collection code / library

```
SELECT COUNT(acirc.id) AS "COUNT", aou.name AS "Library", acl.name AS "Copy Location"
FROM asset.copy ac
INNER JOIN asset.copy_location acl ON ac.location = acl.id
INNER JOIN action.circulation acirc ON acirc.target_copy = ac.id
INNER JOIN actor.org_unit aou ON acirc.circ_lib = aou.id
WHERE DATE_TRUNC('MONTH', acirc.create_time) = DATE_TRUNC('MONTH', NOW() - INTERVAL '3
month')
AND acirc.desk_renewal IS FALSE
AND acirc.opac_renewal IS FALSE
AND acirc.phone_renewal IS FALSE
GROUP BY aou.name, acl.name
ORDER BY aou.name, acl.name, 1
;
```

### 7.2 Monthly circulation stats by borrower stat / library

```
SELECT COUNT(acirc.id) AS "COUNT", aou.name AS "Library", asceum.stat_cat_entry AS "
Borrower Stat"
FROM action.circulation acirc
INNER JOIN actor.org_unit aou ON acirc.circ_lib = aou.id
INNER JOIN actor.stat_cat_entry_usr_map asceum ON asceum.target_usr = acirc.usr
INNER JOIN actor.stat_cat astat ON asceum.stat_cat = astat.id
WHERE DATE_TRUNC('MONTH', acirc.create_time) = DATE_TRUNC('MONTH', NOW() - INTERVAL '3
month')
AND astat.name = 'Preferred language'
AND acirc.desk_renewal IS FALSE
AND acirc.opac_renewal IS FALSE
AND acirc.phone_renewal IS FALSE
GROUP BY aou.name, asceum.stat_cat_entry
ORDER BY aou.name, asceum.stat_cat_entry, 1
;
```

### 7.3 Monthly intralibrary loan stats by library

```
SELECT aou.name AS "Library", COUNT(acirc.id)
FROM action.circulation acirc
  INNER JOIN actor.org_unit aou ON acirc.circ_lib = aou.id
  INNER JOIN asset.copy ac ON acirc.target_copy = ac.id
  INNER JOIN asset.call_number acn ON ac.call_number = acn.id
WHERE acirc.circ_lib != acn.owning_lib
  AND DATE_TRUNC('MONTH', acirc.create_time) = DATE_TRUNC('MONTH', NOW() - INTERVAL '3 ←
  month')
  AND acirc.desk_renewal IS FALSE
  AND acirc.opac_renewal IS FALSE
  AND acirc.phone_renewal IS FALSE
GROUP BY aou.name
ORDER BY aou.name, 2
;
```

### 7.4 Monthly borrowers added by profile (adult, child, etc) / library

```
SELECT pgt.name AS "Profile", aou.name AS "Library", COUNT(pgt.name) AS "Count"
FROM actor.usr au
  INNER JOIN permission.grp_tree pgt
    ON au.profile = pgt.id
  INNER JOIN actor.org_unit aou
    ON aou.id = au.home_ou
WHERE au.deleted IS FALSE
  AND DATE_TRUNC('MONTH', au.create_date) = DATE_TRUNC('MONTH', NOW() - '3 months':: ←
  interval)
GROUP BY pgt.name, aou.name
ORDER BY aou.name, pgt.name
;
```

### 7.5 Borrower count by profile (adult, child, etc) / library

```
SELECT pgt.name AS "Profile", aou.name AS "Library", COUNT(pgt.name) AS "Count"
FROM actor.usr au
  INNER JOIN permission.grp_tree pgt
    ON au.profile = pgt.id
  INNER JOIN actor.org_unit aou
    ON aou.id = au.home_ou
WHERE au.deleted IS FALSE
GROUP BY pgt.name, aou.name
ORDER BY aou.name, pgt.name
;
```

### 7.6 Monthly items added by collection / library

We define a "collection" as a shelving location in Evergreen.

```
SELECT aou.name AS "Library", acl.name, COUNT(ac.barcode)
FROM actor.org_unit aou
  INNER JOIN asset.call_number acn ON acn.owning_lib = aou.id
  INNER JOIN asset.copy ac ON ac.call_number = acn.id
  INNER JOIN asset.copy_location acl ON ac.location = acl.id
WHERE ac.deleted IS FALSE
  AND acn.deleted IS FALSE
```

```
    AND DATE_TRUNC('MONTH', ac.create_date) = DATE_TRUNC('MONTH', NOW() - '1 month':: ←
        interval)
    GROUP BY aou.name, acl.name
    ORDER BY aou.name, acl.name
;
```

## 7.7 Hold purchase alert by library

in the following set of queries, we bring together the active title, volume, and copy holds and display those that have more than a certain number of holds per title. The goal is to UNION ALL the three queries, then group by the bibliographic record ID and display the title / author information for those records that have more than a given threshold of holds.

```
-- Title holds
SELECT all_holds.bib_id, aou.name, rmsr.title, rmsr.author, COUNT(all_holds.bib_id)
FROM
  (
    (
      SELECT target, request_lib
      FROM action.hold_request
      WHERE hold_type = 'T'
      AND fulfillment_time IS NULL
      AND cancel_time IS NULL
    )
    UNION ALL
    -- Volume holds
    (
      SELECT bre.id, request_lib
      FROM action.hold_request ahr
      INNER JOIN asset.call_number acn ON ahr.target = acn.id
      INNER JOIN biblio.record_entry bre ON acn.record = bre.id
      WHERE ahr.hold_type = 'V'
      AND ahr.fulfillment_time IS NULL
      AND ahr.cancel_time IS NULL
    )
    UNION ALL
    -- Copy holds
    (
      SELECT bre.id, request_lib
      FROM action.hold_request ahr
      INNER JOIN asset.copy ac ON ahr.target = ac.id
      INNER JOIN asset.call_number acn ON ac.call_number = acn.id
      INNER JOIN biblio.record_entry bre ON acn.record = bre.id
      WHERE ahr.hold_type = 'C'
      AND ahr.fulfillment_time IS NULL
      AND ahr.cancel_time IS NULL
    )
  ) AS all_holds(bib_id, request_lib)
INNER JOIN reporter.materialized_simple_record rmsr
INNER JOIN actor.org_unit aou ON aou.id = all_holds.request_lib
ON rmsr.id = all_holds.bib_id
GROUP BY all_holds.bib_id, aou.name, rmsr.id, rmsr.title, rmsr.author
HAVING COUNT(all_holds.bib_id) > 2
ORDER BY aou.name
;
```

## 7.8 Update borrower records with a different home library

In this example, the library has opened a new branch in a growing area, and wants to reassign the home library for the patrons in the vicinity of the new branch to the new branch. To accomplish this, we create a staging table that holds a set of city names and

the corresponding branch shortname for the home library for each city.

Then we issue an UPDATE statement to set the home library for patrons with a physical address with a city that matches the city names in our staging table.

```
CREATE SCHEMA staging;
CREATE TABLE staging.city_home_ou_map (city TEXT, ou_shortname TEXT,
    FOREIGN KEY (ou_shortname) REFERENCES actor.org_unit (shortname));
INSERT INTO staging.city_home_ou_map (city, ou_shortname)
    VALUES ('Southbury', 'BR1'), ('Middlebury', 'BR2'), ('Hartford', 'BR3');
BEGIN;

UPDATE actor.usr au SET home_ou = COALESCE(
    (
        SELECT aou.id
        FROM actor.org_unit aou
            INNER JOIN staging.city_home_ou_map schom ON schom.ou_shortname = aou.shortname
            INNER JOIN actor.usr_address aua ON aua.city = schom.city
        WHERE au.id = aua.usr
        GROUP BY aou.id
    ), home_ou)
WHERE (
    SELECT aou.id
    FROM actor.org_unit aou
        INNER JOIN staging.city_home_ou_map schom ON schom.ou_shortname = aou.shortname
        INNER JOIN actor.usr_address aua ON aua.city = schom.city
    WHERE au.id = aua.usr
    GROUP BY aou.id
) IS NOT NULL;
```

## 8 License

This work is licensed under a [Creative Commons Attribution-Share Alike 2.5 Canada License](#).