Introduction to SQL for Evergreen administrators

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
2.9	February 2012		DS

Contents

1 Preface **1**

2 Part 1: Introduction to SQL Databases **1**

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 `psql` . . . 5

3 Part 2: Basic SQL queries **6**

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: ' . . . 10
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 . . . 13

4 Part 3: Advanced SQL queries **14**

4.1 Learning objectives . . . 14
4.2 Transforming column values with functions . . . 14
4.2.1 Scalar functions . . . 14
4.2.2 Aggregate functions . . . 15
4.2.3 HSTORE columns . . . 16
4.3 Subqueries . . . 16
4.4 Joins . . . 17

4.4.1 Inner joins . . . 17
4.4.2 Aliases . . . 18
4.4.3 Outer joins . . . 18
4.4.4 Self joins . . . 20
4.4.5 WITH clauses . . . 20
4.5 Set operations . . . 21
4.5.1 Union . . . 22
4.5.2 Intersection . . . 22
4.5.3 Difference . . . 23
4.6 Views . . . 24
4.7 Inheritance . . . 24

5 Part 4: Understanding query performance with EXPLAIN 25

6 Part 5: Inserting, updating, and deleting data 27
6.1 Inserting data . . . 27
6.2 Inserting data using a SELECT statement . . . 27
6.3 Inserting bulk data using a COPY statement . . . 28
6.4 Deleting rows . . . 28
6.5 Updating rows . . . 29

7 Part 6: The active database 29
7.1 Functions (revisited) . . . 29
7.2 Triggers . . . 30
7.3 Rules . . . 30

8 Part 6: Issuing batch updates for bib records 31

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.

actor.usr_note database table

```
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.

Example: Creating a table without a specific schema

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

Example: Trying to access a unqualified table outside of the public schema

```
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 2012-01-23) 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

Table 2: (continued)

Schema name	Description
offline	Offline transactions
permission	User permissions
query	Stored SQL statements
reporter	Report definitions
search	Search functions
serial	Serial MFHD records
staging	User records prior to approval
stats	Convenient views of circulation and asset statistics
unapi	Metadata formats
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 (####.##).
HSTORE	Key-value pairs	

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 keys 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
asset.copy	asset.copy.id	asset.copy.call_number	asset.call_number.id
asset.call_number	asset.call_number.id	asset.call_number.record	biblio.record_entry.id
biblio.record_entry	biblio.record_entry.id		

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 (9.1.2)
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:

```
SEL ECT [columns(s)]
 FROM [table(s)]
  [WHERE condition(s)]
  [GRO U P BY columns(s)]
  [HAV ING grouping-condition(s)]
  [ORD E R BY column(s)]
  [LIMIT maximum-results]
  [OFF SET 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*.

Trouble with OR

```
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!
```

Grouped OR'ed conditions

```
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 ~ y` (x matches the regular expression *y*; regular expressions are extremely powerful but would require another tutorial entirely!)
- `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)

IN clause examples

```
SELECT usr
  FROM actor.usr_address
  WHERE state = 'Ohio'
    AND city IN ('Columbus', 'Lancaster', 'Springfield')
;

SELECT id, usrname
  FROM actor.usr
  WHERE family_name = 'Bandy'
    AND id IN (
      SELECT usr
        FROM actor.usr_address
        WHERE state = 'Ohio'
          AND city in ('Columbus', 'Lancaster', 'Springfield')
    )
;
```

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.

Retrieving rows where a column is not NULL

```
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.

Retrieving rows where a column 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    |                   | 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:

Changing the way NULL values are displayed in psql

```
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:

Escaping ' in TEXT values

```
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.

Returning unique results of a single column with GROUP BY

```
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:

Returning unique results of multiple columns with GROUP BY

```
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.*

Counting unique column combinations with GROUP BY

```
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.

Using the WHERE clause with GROUP BY

```
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.

GROUP BY restricted by a HAVING clause

```
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.

Returning unique results of multiple columns with DISTINCT

```
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:

Using the LIMIT clause to restrict results

```
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 prevously executed.

Using the OFFSET clause to return a specific subset of rows

```
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
- Be able to query and retrieve values from HSTORE columns
- Know how to retrieve values from across multiple tables
- Understand the difference between inner joins, outer joins, unions, and intersections
- Know how to use the WITH clause to simplify a single query
- 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:

Using the UPPER() scalar function to convert text values to upper-case

```
-- 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

> **Note**
> Evergreen uses its own custom function, `evergreen.lowercase()`, to convert text to lower-case because it uses Perl's `lc()` function to do a better job with Unicode text.

- 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. Commonly used aggregate functions include:

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

4.2.3 HSTORE columns

The HSTORE data type stores key-value pairs within a single column. You need to use special functions and operators to create, update, and query HSTORE columns. In Evergreen, the `metarecord.record_attrs` table includes an HSTORE column `attrs` for attributes from the leader and fixed fields like cataloging format, audience, and bibliographic level.

To create or update the value for a given key, use the concatenation operator (`||`) to concatenate the name of the HSTORE column with the new HSTORE key-value pair. For example:

Updating an HSTORE column

```
UPDATE metabib.record_attr
  SET attrs = attrs || hstore('bib_level', 's')
  WHERE id = 10
;
```

To retrieve a value from an HSTORE column for a given key, use the `->` operator to specify the name of the key against the HSTORE column. For example, to retrieve the bibliographic level and audience from the `metabib.record_attr` table:

Retrieving values from an HSTORE column

```
SELECT id, attrs->'audience' AS audience, attrs->'bib_level' AS bib_level
  FROM metabib.record_attr
  LIMIT 5
;

 id | audience | bib_level
----+----------+-----------
  1 | 0        | m
  2 | f        | m
  3 |          | m
  4 |          | m
  5 |          | m
(5 rows)
```

4.3 Subqueries

A subquery 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 subquery to restrict the copies returned by the main SELECT statement to only those locations that have an `opac_visible` value of `TRUE`:

Subquery example

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

Subqueries 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 subqueries within subqueries, you should consider tackling the problem with joins or WITH clauses 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.

A simple inner join

```
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:

Borrower Count by Profile (Adult, Child, etc)/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:

Borrower Count by Profile (using aliases)

```
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.

Base tables for the OUTER JOIN examples

```
-- user address table
SELECT id, city, county, state
  FROM actor.usr_address
  WHERE state = 'GA'
;

 id  |    city     | county | state
-----+-------------+--------+-------
 116 | Atlanta     | Fulton | GA
 118 | Thomasville | Thomas | GA
 148 | Lincolnton  |        | GA
(3 rows)

-- org_unit address table
SELECT id, city, county, state
  FROM actor.org_address
  WHERE state = 'GA'
;

 id |   city    | county | state
----+-----------+--------+-------
  1 | Anywhere  |        | GA
  2 | Anywhere  |        | GA
  3 | Anywhere  |        | GA
  4 | Anywhere  |        | GA
  5 | Anywhere  |        | GA
  6 | Anywhere  |        | GA
  7 | Anywhere  |        | GA
  8 | Anywhere  |        | GA
  9 | Anywhere  |        | GA
(9 rows)
```

Example of a LEFT OUTER JOIN

```
-- No org_unit addresses match 'OH', so we get all of the user
-- rows that match 'OH' and NULL values for the org_unit rows
SELECT aua.city, aua.state, aoa.org_unit
  FROM actor.usr_address aua
    LEFT OUTER JOIN actor.org_address aoa
      ON aua.state = aoa.state
  WHERE aua.state = 'OH'
  ORDER BY aua.city
;

     city      | state | org_unit
---------------+-------+----------
 Canton        | OH    |
 Cedarville    | OH    |
 Delaware      | OH    |
 Fort jennings | OH    |
 Gomer         | OH    |
```

```
 Hallsville    | OH    |
 Mc clure      | OH    |
 Southington   | OH    |
 Streetsboro   | OH    |
 Waverly       | OH    |
(10 rows)
```

Example of a RIGHT OUTER JOIN

```
-- Only 3 user rows match state = 'GA', but all of the org_unit
-- rows match, so we get each org_unit matched against each user
SELECT aua.city, aua.state, aoa.org_unit
  FROM actor.usr_address aua
    RIGHT OUTER JOIN actor.org_address aoa
      ON aoa.state = aua.state
  WHERE aua.state = 'GA'
  ORDER BY aoa.org_unit
;

    city     | state | org_unit
-------------+-------+----------
 Atlanta     | GA    |        1
 Thomasville | GA    |        1
 Lincolnton  | GA    |        1
 Atlanta     | GA    |        2
 Thomasville | GA    |        2
 Lincolnton  | GA    |        2
 Atlanta     | GA    |        3
...
 Lincolnton  | GA    |        9
(27 rows)
```

Example of a FULL OUTER JOIN

```
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.

4.4.5 WITH clauses

WITH clauses (more formally referred to as *common table expressions*, or *CTEs*) enable you to define one or more subqueries as a kind of temporary table that you can then reference in your main SELECT statement. Along with flexibility, by defining the subqueries of your SELECT statement up-front with meaningful names, your query is often easier to read. However, because PostgreSQL has to create the temporary tables before executing the main body of the query, the CTE may result in performance problems for some classes of queries. In those cases, subqueries or joins against views may be a better option.

In the following example, we use a WITH clause to define a subquery, referenced as *funds*, as a local acquisitions practice for this Evergreen site records the fund code for each record as a three-digit value in the `912 $a` subfield. As we are interested in the results of only the last 200 purchases, we use an ORDER BY clause to sort the results of the subquery by the record number and a LIMIT clause within the subquery to give us a maximum of 200 results. That leaves us with a very simple main SELECT statement to summarize the results.

WITH clause example

```
WITH funds AS (
  SELECT record, value
  FROM metabib.full_rec
  WHERE tag = '912'
    AND subfield = 'a'
    AND value ~ '^\d\d\d$'
  ORDER BY record DESC
  LIMIT 200
)
SELECT value, COUNT(value)
FROM funds
GROUP BY value
ORDER BY value;

 value | count
-------+-------
 132   |     4
 151   |    14
 152   |     2
 190   |     2
 202   |     3
 212   |     1
...
```

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.

Base tables for the set operation examples

```
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.

In the following example, we use `UNION` operators to bring together all the possible permissions for each user in the system. A user can inherit permissions from their user profile, from the permission groups to which they belong, and can be granted individual permissions, so we join three result sets using two `UNION` operators. In addition, for the sake of convenience we define the entire result set as a *view* so that we can subsequently refer to the result set without having to repeat the entire statement.

Example of a UNION set operation

```
CREATE VIEW all_users_all_perms AS
(
  SELECT au.usrname, 'Individually granted'::text AS perm_source, ppl.code,
    pupm.depth, pupm.grantable
  FROM actor.usr au
    INNER JOIN permission.usr_perm_map pupm ON pupm.usr = au.id
    INNER JOIN permission.perm_list ppl ON ppl.id = pupm.perm
)
UNION
(
  SELECT au.usrname, 'Group membership - '::text || pgt.name AS perm_source,
    ppl.code, pgpm.depth, pgpm.grantable
  FROM actor.usr au
    INNER JOIN permission.usr_grp_map pugm ON pugm.usr = au.id
    INNER JOIN permission.grp_tree pgt ON pgt.id = pugm.grp
    INNER JOIN permission.grp_perm_map pgpm ON pgpm.grp = pgt.id
    INNER JOIN permission.perm_list ppl ON ppl.id = pgpm.perm
)
UNION
(
  SELECT au.usrname, 'User profile - '::text || pgt.name AS perm_source,
    ppl.code, pgpm.depth, pgpm.grantable
  FROM actor.usr au
    INNER JOIN permission.grp_tree pgt ON pgt.id = au.profile
    INNER JOIN permission.grp_perm_map pgpm ON pgpm.grp = pgt.id
    INNER JOIN permission.perm_list ppl ON ppl.id = pgpm.perm
);
```

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.

In the following example, we use the `INTERSECT` operator to show which permissions two users have in common, based on the `all_users_all_perms` view that we created in the previous example.

Example of an INTERSECT set operation

```
(
  SELECT code, depth, grantable
  FROM all_perms_for_all_users
  WHERE usrname = 'bm1alopez'
)
INTERSECT
(
  SELECT code, depth, grantable
  FROM all_perms_for_all_users
  WHERE usrname = 'bm1epeterson'
);
```

```
                    code                     | depth | grantable
---------------------------------------------+-------+-----------
 ADMIN_BOOKING_RESOURCE_ATTR_MAP             |     1 | t
 CREATE_PATRON_STAT_CAT                      |     1 | t
 SET_CIRC_CLAIMS_RETURNED                    |     1 | t
 DELETE_PATRON_STAT_CAT_ENTRY                |     1 | t
 ADMIN_BOOKING_RESERVATION_ATTR_MAP          |     1 | t
 ABORT_TRANSIT_ON_MISSING                    |     0 | t
 CREATE_USER_GROUP_LINK                      |     1 | t
 ADMIN_COPY_LOCATION_ORDER                   |     1 | t
 UPDATE_PICKUP_LIB_FROM_TRANSIT              |     1 | t
...
```

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.

In the following example, we use the `EXCEPT` operator to show which permissions the first user has that the second user does not, based on the `all_users_all_perms` view that we created in the previous example.

Example of an EXCEPT set operation

```
(
  SELECT code, depth, grantable
  FROM all_perms_for_all_users
  WHERE usrname = 'bm1alopez'
)
EXCEPT
(
  SELECT code, depth, grantable
  FROM all_perms_for_all_users
  WHERE usrname = 'bm1hstone'
)
ORDER BY 1;
                    code                     | depth | grantable
---------------------------------------------+-------+-----------
 ABORT_TRANSIT_ON_LOST                       |     0 | t
 ABORT_TRANSIT_ON_MISSING                    |     0 | t
 ADMIN_BOOKING_RESERVATION                   |     1 | t
 ADMIN_BOOKING_RESERVATION_ATTR_MAP          |     1 | t
...

-- Order matters: switch the left-hand and right-hand tables
-- and you get a different result
(
  SELECT code, depth, grantable
  FROM all_perms_for_all_users
  WHERE usrname = 'bm1hstone'
)
EXCEPT
(
  SELECT code, depth, grantable
  FROM all_perms_for_all_users
  WHERE usrname = 'bm1alopez'
)
ORDER BY 1;
                code                 | depth | grantable
-------------------------------------+-------+-----------
 ACQ_INVOICE_REOPEN                  |     0 | t
 ACQ_XFER_MANUAL_DFUND_AMOUNT        |     0 | t
```

```
 ADMIN_ACQ_CANCEL_CAUSE             |     0 | t
 ADMIN_ACQ_CLAIM                    |     0 | t
...
```

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:

Creating a view

```
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.

Selecting results from a view

```
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:

Query plan for a terrible query

```
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.

Query plan for a less terrible query

```
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.

Query plan for a realistic query

```
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.

Query plan with optimized access via a new index

```
-- 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.

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

Inserting 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 subqueries:

Inserting rows using subqueries instead of integers

```
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`:

Inserting rows via a SELECT statement

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

INSERT 0 6
```

6.3 Inserting bulk data using a COPY statement

Given a large amount of data in a text file, you can use the `COPY` statement to quickly insert that data into a table. `COPY` statements are optimized for bulk loading and are faster than issuing the equivalent `INSERT` statements. By default, `COPY` expects tab-delimited files with one record per line, but you can specify different options. You can also copy data from STDIN, which can be useful for scripting data loads.

For example, given the following file of raw data in which ` → ` represents a `<TAB>` character, we can load some copy locations into Evergreen:

Example tab-separated file for copy locations

```
Storage -> 4
Newspaper room -> 5
```

Example command for loading data from a file

```
COPY asset.copy_location(name, owning_lib) FROM '/path/to/file.tsv';
```

As previously noted, you can load data from STDIN. In this case, you need to identify the end of the data in the file with `\.` appearing on a line by itself. This delimiter enables other commands to follow it, for example for a script to populate many different tables in a database.

Example command for loading data from STDIN

```
COPY asset.copy_location(name, owning_lib) FROM STDIN;
Storage -> 4
Newspaper room -> 5
\.
```

6.4 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:

Deleting rows from a table

```
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.

6.5 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:

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

7 Part 6: The active database

When you insert, update, or delete rows in a table, the data does not necessarily stay exactly as you specified. If you invoked functions on the data, it will have been modified according to the function specification. Further, the tables themselves might have triggers or rules defined on them that modify the incoming data or cause entirely different actions to happen.

7.1 Functions (revisited)

PostgreSQL supports user-defined functions—that is, scalar or aggregate functions written in one of a number of different languages—and Evergreen relies heavily on user-defined functions to support its business logic. Accordingly, to understand how the Evergreen database schema transforms input into the results you and your users experience, you need to understand a number of different functions written in SQL, plpgsql, and plperlu.

For just a few examples:

- `search.query_parser_fts`, Evergreen's core logic for finding visible bibliographic records that match a search query, is written in over 300 lines of plpgsql and takes 11 different arguments.
- `search_normalize` implements a modified version of the NACO normalization algorithm in 60 lines of plperlu, taking raw text from bibliographic records and converting it into more easily searchable text by converting ligatures into their ASCII equivalents, stripping punctuation, and other normalizations.
- `action.purge_circulations`, which removes transaction records that have closed to protect confidentiality of patrons while retaining non-identifying details for statistical purposes, is written in 90 lines of plpgsql.

7.2 Triggers

A trigger can be added to a table to specify a function that should be invoked when that table is modified by means of an INSERT, UPDATE, or DELETE statement. The trigger can fire before or after the statement, and can fire once per modified row, or once per statement, depending on the trigger definition. A trigger can effectively prevent the statement from happening entirely, modify the data that was to have been inserted or updated, and insert data into other tables. Tables can have multiple triggers defined on them, firing matching BEFORE triggers before AFTER triggers, with triggers within those categories firing in alphabetical order.

For example, the `biblio.record_entry` table has a number of triggers defined on it:

Triggers defined on biblio.record_entry

```
Triggers:
    a_marcxml_is_well_formed BEFORE INSERT OR UPDATE ON                    ❶
        biblio.record_entry FOR EACH ROW
        EXECUTE PROCEDURE biblio.check_marcxml_well_formed()
    a_opac_vis_mat_view_tgr AFTER INSERT OR UPDATE ON                      ❷
        biblio.record_entry FOR EACH ROW
        EXECUTE PROCEDURE asset.cache_copy_visibility()
    aaa_indexing_ingest_or_delete AFTER INSERT OR UPDATE ON                ❸
        biblio.record_entry FOR EACH ROW
        EXECUTE PROCEDURE biblio.indexing_ingest_or_delete()
    audit_biblio_record_entry_update_trigger                               ❹
        AFTER DELETE OR UPDATE ON biblio.record_entry FOR EACH ROW
        EXECUTE PROCEDURE auditor.audit_biblio_record_entry_func()
    b_maintain_901 BEFORE INSERT OR UPDATE ON                              ❺
        biblio.record_entry FOR EACH ROW
        EXECUTE PROCEDURE maintain_901()
    bbb_simple_rec_trigger AFTER INSERT OR DELETE OR UPDATE ON             ❻
        biblio.record_entry FOR EACH ROW EXECUTE PROCEDURE
        reporter.simple_rec_trigger()
    c_maintain_control_numbers BEFORE INSERT OR UPDATE ON                  ❼
        biblio.record_entry FOR EACH ROW
        EXECUTE PROCEDURE maintain_control_numbers()
    fingerprint_tgr BEFORE INSERT OR UPDATE ON                             ❽
        biblio.record_entry FOR EACH ROW
        EXECUTE PROCEDURE biblio.fingerprint_trigger('eng', 'BKS')
```

❶ Checks to ensure that the MARCXML is clean XML.

❷ Updates the visible copies for the record, in the case that the record was undeleted.

❸ Updates the search indexes for the record after all modifications to the MARCXML are complete.

❹ Adds an entry to the `audit.biblio_record_entry_history` table to track the change to this record.

❺ Updates the 901 field for the record before it is inserted or updated.

❻ Updates the `reporter.materialized_simple_record` table.

❼ Updates the 001 and 035 fields of the record.

❽ Creates a fingerprint of the record, generally consisting of a concatenation of the author and title, and derives a quality value for the record.

7.3 Rules

Rules are conceptually similar to triggers, in that they modify the results of SELECT, INSERT, UPDATE, or DELETE statements for a given table, but their implementation is not contained in a function. Instead, the implementation is one or more SQL statements that are run in addition to, or instead of, the matching SQL statement.

For example, the following rule on `biblio.record_entry` prevents a DELETE statement from actually deleting the targeted row from the table, but instead sets the value of the `deleted` column to `TRUE` and removes the corresponding row from the `metabib.metarecord_source_map` table to effectively make the record invisible to regular users. This enables Evergreen to maintain referential integrity for any bookbags or call numbers that might reference the now-deleted record, for example.

Rule defined on biblio.record_entry

```
Rules:
    protect_bib_rec_delete AS
    ON DELETE TO biblio.record_entry DO INSTEAD (
 UPDATE biblio.record_entry SET deleted = true
  WHERE old.id = record_entry.id;
 DELETE FROM metabib.metarecord_source_map
  WHERE metarecord_source_map.source = old.id;
```

8 Part 6: Issuing batch updates for bib records

Evergreen sites often have to make changes in bulk to the MARC records in the database. Given that the MARC is stored as a `TEXT` field, we can use standard search and replace string functions for very simplistic changes.

For example, let's assume that the hostname for your electronic books provider has changed (seemingly capriciously) from `http://books.example.com` to `https://ebooks.example.com`. You could use the `replace()` function to update all of the records in your database in a single statement, as follows:

Example: updating a hostname across all records (take 1)

```
UPDATE biblio.record_entry
  SET marc = replace(marc,
    'http://books.example.com',
    'https://ebooks.example.com'
  )
;
```

This would be inefficient, however, as the statement would attempt to update every record in the `biblio.record_entry` table, even those which do not contain a matching string. So, a more efficient approach would include a WHERE clause:

Example: updating a hostname across all records (take 2)

```
UPDATE biblio.record_entry
  SET marc = replace(marc,
    'http://books.example.com',
    'https://ebooks.example.com'
  )
  WHERE marc LIKE '%http://books.example.com%'
;
```

However, this may still be problematic. The `replace()` function does not understand the structure of MARCXML at all, and it is possible (although unlikely with this particular example) that the targeted string could also occur in the title of the record, or the author field, or places other than the 856 field, $u subfield that we actually care about. Thus, we can try using a regular expression to limit the update:

Example: updating a hostname across all records (take 3)

```
UPDATE biblio.record_entry
  SET marc = regexp_replace(
    marc,
    '(<datafield tag="856"[^>]*?>.*?<subfield code="u">)' ||
      'http://books.example.com',
    '\1https://ebooks.example.com',
    'g'
  ) WHERE marc ~
```

```
    '<datafield tag="856"[^>]*?>.*?<subfield code="u">' ||
      'http://books.example.com'
;
```