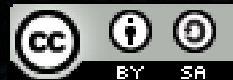
Evergreen SQL Bootcamp

Dan Scott
Coffee|Code Consulting
January 31/February 1, 2013



Day 1 agenda

- 9:00 – 9:30: Connect to the test server
- 9:30 – 10:30: Database concepts
- 10:30 – 11:00: Explore the Evergreen schema
- 11:00 – 12:00: SELECT statements, part 1
- 12:00 – 12:30ish: Lunch break
- 12:30ish – 1:30: SELECT statements, part 2
- 1:30 – 2:00: SELECT exercises
- 2:00 – 3:00 : Joining tables

Introducing SQL databases

- SQL: Structured Query Language
- Tables
- Rows (aka *tuples*) and columns (aka *fields*)
- Schemas
- Data types
- Constraints
- What more could you possibly want?

Tables, rows, and columns

- A database contains one or more *tables*, each of which has a specific name
  - actor.usr, action.circulation, asset.copy
- Tables hold rows of data that conform to a specific definition for that table; each row has one or more columns with specific *data types*
  - id INTEGER, first_given_name TEXT

Tables - rows

Row 1				
Row 2				
Row 3				
Row 4				

Note that rows in tables have no implicit order; the 1, 2, 3, 4 is just for demonstration purposes.

Tables - columns

id INTEGER	*code* TEXT	*name* TEXT	*created* DATE

Schemas

- The overall design of a database – the way that data is split between different tables – is called the *database schema*
- Tables are logically grouped together in namespaces that are also, confusingly, called *schemas*
  - *actor* schema: *org_unit* and *usr* tables
  - *asset* schema: *call_number* and *copy* tables
- A fully-qualified table name includes the schema name: *actor.org_unit*

Schemas – table groupings

asset

call_number
copy
copy_location
stat_cat
uri
...

actor

card
org_unit
stat_cat
usr
usr_address
...

action

circulation
hold_request
hold_transit_copy
hold_request_note
survey
...

config

bib_source
billing_type
circ_modifier
copy_status
identification_type
...

Data types used in Evergreen

Type	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
BOOL	Boolean	TRUE or FALSE
TIMESTAMP WITH TIME ZONE	Timestamp	4713 BC to 294276 AD
TIME	Time	Expressed in HH:MM:SS
NUMERIC	Decimal	Mostly used for money values in Evergreen

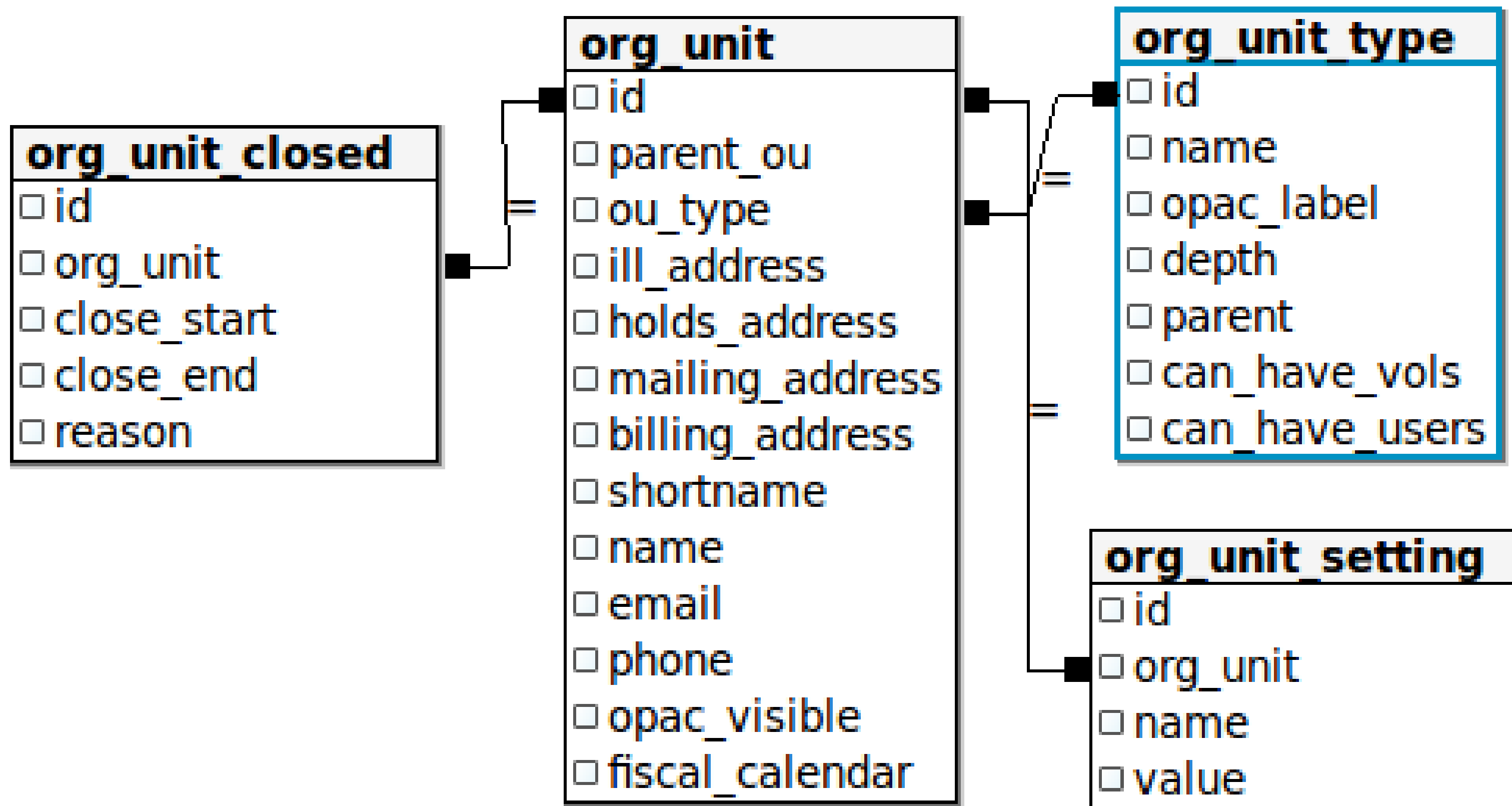
Constraints

- Column constraints ensure that the values in a given table make sense for the object being modelled
  - Data types are a kind of constraint
  - *NOT NULL* constraints require any value
  - *Primary key* uniquely identifies a row
  - *Foreign key* must have a corresponding value in another table
  - *Check constraints* place arbitrary requirements on the value (e.g. ZIP code)

Sequences

- Sequences are often used to provide a synthetic key for a table
- Normally defined as a SERIAL or BIGSERIAL column
  - Type will be INTEGER or BIGINT
  - Default value of that column will be associated with a sequence object
- If you insert a non-default value for a sequence, subsequent INSERT statements may fail until the sequence is clear

Simple relational example



psql client

- \h for SQL statement help
- \? for psql command help
- Tab completion and readline support
- \d commands describe database objects
  - \d - tables and views
  - \dn - schemas
  - \df - functions

Let's explore the Evergreen schema

- Goal: tell me about the *actor.card* table
  - How many columns, of what type?
  - What kind of constraints are on each column?
  - Are there any relationships to other tables?
- Goal: tell me about the *actor.usr* table
- Goal: tell me about the *actor.org_unit* table

The SELECT statement

- The SELECT statement selects one or more column values from a set of data
- SQL 101:

```
SELECT * FROM actor.usr;
```

- * means “select all columns from the set”
- actor.usr is the schema-qualified table name that forms the set of data

Selecting specific columns

- Name the columns you want, separated by commas

```
SELECT actor.usr.first_given_name, actor.usr.family_name
  FROM actor.usr;
```

- If the column name is unambiguous, you can drop the schema & table qualifiers:

```
SELECT first_given_name, family_name
  FROM actor.usr;
```

Sorting rows: ORDER BY

- If you want the rows returned in a particular order, use the ORDER BY clause to identify the columns to sort the results by in ascending or descending order

```
SELECT first_given_name, family_name
  FROM actor.usr
  ORDER BY family_name, first_given_name DESC;
```

- You can also use the column number instead of the column name; useful when the column has no name!

Filtering rows: WHERE clause

- Specify one or more conditions in the WHERE clause to exclude rows from the results

```
SELECT first_given_name, family_name
  FROM actor.usr
  WHERE family_name = 'System Account';
```

- Conditions can be connected with AND, OR, and NOT; parentheses group conditions

```
SELECT first_given_name, family_name
  FROM actor.usr
  WHERE family_name = 'System Account'
    AND first_given_name = 'Administrator';
```

WHERE clause operators

- The WHERE clause supports a number of comparison operators:
  - x = y (x is equal to y)
  - x != y (x is not equal to y)
  - x < y (x is less than y)
  - x > y (x is greater than y)
  - x IN (a, b, c) (x matches one of a, b, or c)
  - x ~ '<regex>' (x matches a regular expression)

WHERE clause operators (2)

- x BETWEEN a AND b (syntactic sugar for *x >= a AND x <= b*)
- x LIKE 'a%x_z' (text pattern match)
- x ILIKE 'a%x_z' (case-insensitive text pattern match)

* % wildcard matches zero or more characters
* _ - wildcard matches *exactly* one character

```
SELECT * FROM actor.usr WHERE first_given_name = 'Admin%istrator';
– 1 row
SELECT * FROM actor.usr WHERE first_given_name = 'Admin_istrator';
– 0 rows
```

NULL values

- A NULL value is not an empty string, or a 0 – it is a non-value; use the IS NULL or IS NOT NULL comparison operators

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

- NULL values will throw curves at you!
  - NULL never matches any other value, even other NULL values
  - Concatenating a NULL always returns NULL

Text delimiter: '

- TEXT values are delimited by single quotes (')
- To use a single quote inside a TEXT value, *escape* the single quote by prepending another single quote to it:

```
SELECT first_given_name, family_name
  FROM actor.usr
  WHERE family_name IS 'L''estat';
```

Grouping results: GROUP BY

- The GROUP BY clause returns a unique set of results for the grouped columns:

```
SELECT ou_type
  FROM actor.org_unit
  ORDER BY ou_type;
 ou_type
---------
       1
       2
       2
       3
       3
       3
       3
       4
       5
(9 rows)
```

```
SELECT ou_type,
COUNT(ou_type)
  FROM actor.org_unit
  GROUP BY ou_type
  ORDER BY ou_type;

 ou_type | count
---------+-------
       1 |     1
       2 |     2
       3 |     4
       4 |     1
       5 |     1
(5 rows)
```

Filtering grouped rows: HAVING

- While the WHERE clause filters individual rows, the HAVING clause filters rows based on an aggregate function:

```
SELECT ou_type, COUNT(ou_type)
  FROM actor.org_unit
  GROUP BY ou_type
  HAVING COUNT(ou_type) > 1;

 ou_type | count
---------+-------
       3 |     4
       2 |     2
(2 rows)
```

Eliminating duplicates: DISTINCT

- Use the DISTINCT operator to eliminate duplicate rows from your results:

```
SELECT DISTINCT ou_type
FROM actor.org_unit
ORDER BY ou_type;

 ou_type
---------
       1
       2
       3
       4
       5
(5 rows)
```

Eliminating: DISTINCT ON ()

- The DISTINCT ON () operator eliminates duplicate sets of one or more column values; must match ORDER BY column order

```
SELECT DISTINCT ON (ou_type) name
  FROM actor.org_unit
  ORDER BY ou_type;

         name
-----------------------
 Example Consortium
 Example System 1
 Example Branch 1
 Example Sub-library 1
 Example Bookmobile 1
(5 rows)
```

Paging: LIMIT / OFFSET

- The LIMIT clause specifies the maximum number of rows to return from the complete result set
- The OFFSET clause specifies how far to advance in the result set before returning the first row

```
SELECT * FROM actor.usr LIMIT 5 OFFSET 10;
```

- This example would return 5 or fewer rows, starting at the 10th row of the result set

Exercises

1) List all of the values for the first ten users, by create date, in the system.

2) List the first name and last name of the 10th through 20th users, ordered by last name, whose home library is set to `Example Branch 1`.

3) List each library with a count of the number of users per library who have not been deleted.

4) List the email address and user name of all active users with a last name of “Scott” or “Smith”.

JOINed at the hip

- You need to master joins to be able to work effectively with data from multiple tables. A join always brings two sets of data together
- If you're joining 10 tables, you're still working with two sets of data at a time; the sets on the left-hand side are just getting bigger and bigger each time.
- The INNER JOIN is the easiest join to master; it returns rows only if both the left-hand table and right-hand table match the join condition.

INNER JOIN

id	usrname
1	Frank
2	Carol
3	Bob

usr	title	value
2	Hey!	This is a note
4	Ho ho ho	Loves XMAS
10	Curses	Foul mouth
20	Buffy	BTVS

```
SELECT au.usrname, aun.title
  FROM actor.usr au INNER JOIN actor.usr_note aun
  ON au.id = aun.usr;

 usrname   title
---------+-------
Carol     Hey!
(1 rows)
```

INNER JOIN practice

1) List the user name, email address, and home library name for the first 10 users in order of last name (Z to A)
2) List the barcodes for all users who have a home library of `Example Branch 1`.
3) Count the number of circ transactions for users who have a home library with the short name `BR1`.

OUTER JOIN

- An outer join returns NULL values for all columns in rows that do not match the join condition
- There are three kinds of outer join:
  - The left outer join returns all rows from the left-hand table
  - The right outer join returns all rows from the right-hand table
  - The full outer join returns all rows from the left-hand table and the right-hand table

OUTER JOIN practice

- List the user name, family name, and any user notes for all users in the system whether or not they have user notes attached to their account (first 100 results only).

Some handy functions

- string1 || string2 - || concatenates two strings together; if one string is NULL, then a NULL is returned instead
- coalesce(value 1, value2) – returns the first non-NULL value
- trim() - removes spaces by default from the start and end of a string
- upper() - changes a string to upper case
- lower() - changes a string to lower case
-

Set operators

- UNION – adds the set of rows from the right-hand table to the left-hand table
- INTERSECT – returns the rows that exist in both the left-hand and right-hand tables
- EXCEPT – returns the rows from the left-hand table that do not exist in the right-hand table

Day 2 agenda

- 9:00 – 10:00: SELECT refresher exercises
- 10:00 – 10:30: Subqueries
- 10:30 – 11:00: Common table expressions
- 11:00 – 12:00: Views
- 12:30 – 1:30: INSERTing data
- 1:30 – 2:00: DELETEing data
- 2:00 – 2:30: UPDATEing data

Refreshers

1) List the schema names starting with `meta`
2) List the table names starting with `meta`
3) List the record #, indicators, and value for rows in `metabib.full_rec` with a tag of `'245'`, a subfield of `'a'`, and a value containing `'mozart'`
4) List the date of creation for the records returned in query # 3

Subqueries

- A subquery is a complete SELECT statement that you can use to replace hard-coded values
  - Such as an `IN (a, b, c)` clause
  - Or a complete table in a JOIN clause
- Wrap the SELECT statement in `()`
- To act as a table, append `AS` *`query-name`* (which functions as a table name)
- Subqueries can be nested for maximum confusion!

Subquery in an IN() clause

```
SELECT create_date
FROM biblio.record_entry
WHERE id IN (
  SELECT record
  FROM metabib.full_rec
  WHERE tag = '245'
    AND subfield = 'a'
    AND value ~ 'mozart'
);
```

Subquery in a JOIN clause

```
SELECT bre.create_date
FROM biblio.record_entry bre
  INNER JOIN (
    SELECT record, ind1, ind2, value
    FROM metabib.full_rec
    WHERE tag = '245'
      AND subfield = 'a'
      AND value ~ 'mozart'
  ) AS mozart
ON mozart.record = bre.id;
```

Nested subqueries

```
SELECT id, create_date
FROM biblio.record_entry
WHERE id IN (
  SELECT record
  FROM asset.call_number acn
  WHERE acn.id IN (
    SELECT call_number
    FROM asset.copy ac
    WHERE ac.id IN (
      SELECT target_copy
      FROM action.circulation
    )
  )
);
```

Subquery exercises

- Create a unique list of the shortnames of libraries that have users with a family name starting with 'S'
- List the barcodes for books that have circulated at the library with shortname 'BR3' with a duration of 1 hour

JOINs rule!

```
SELECT bre.id, bre.create_date
FROM biblio.record_entry bre
  INNER JOIN asset.call_number acn
    ON acn.record = bre.id
  INNER JOIN asset.copy ac
    ON ac.call_number = acn.id
  INNER JOIN action.circulation acirc
    ON ac.id = acirc.target_copy
-- GROUP BY bre.id, acirc.xact_start
-- ORDER BY acirc.xact_start DESC
-- LIMIT 10
-- Concise SQL means bonus statements!
;
```

Common table expressions (CTEs)

- Define CTEs via a leading WITH clause
- One or more leading subqueries that you can then reference in the rest of the statement
- Useful for clarity, if you need a complicated subquery for an IN clause or JOIN condition
- Beware: results go into temporary tables, so large CTEs can impact performance

WITH a basic CTE

```
WITH records AS (
  SELECT record
  FROM asset.call_number acn
  WHERE acn.owning_lib IN (
    SELECT id
    FROM actor.org_unit
    WHERE shortname = 'BR1'
  )
)
SELECT id, create_date
FROM biblio.record_entry bre
  INNER JOIN records
ON records.record = bre.id
;
```

WITH multiple CTEs

```
WITH libs AS (
  SELECT id
  FROM actor.org_unit
  WHERE shortname = 'BR1'
), records AS (
  SELECT record
  FROM asset.call_number acn
    INNER JOIN libs ON libs.id = acn.owning_lib
)
SELECT id, create_date
FROM biblio.record_entry bre
  INNER JOIN records
ON records.record = bre.id
;
```

CTE exercises

- Create a unique list of the shortnames of libraries that have users with a family name starting with 'S'
- List the barcodes for books that have circulated at the library with shortname 'BR3' with a duration of 1 hour
- *Yes, these should look familiar :)*

Views

- A view is a stored SELECT statement with a name that acts like a read-only table
  - Views can be arbitrarily complex and participate in JOINs just like a real table
  - Excellent means of capturing knowledge and simplifying queries

```
CREATE VIEW view-name AS SELECT ...;
```

Viewing views (continued)

- To view a view definition in psql, use the \d+ command:

```
\d+ reporter.circ_type
              View "reporter.circ_type"
 Column |  Type  | Modifiers | Storage  | Description
--------+--------+-----------+----------+-------------
 id     | bigint |           | plain    |
 type   | text   |           | extended |
View definition:
 SELECT circulation.id,
        CASE WHEN circulation.opac_renewal OR
circulation.phone_renewal OR circulation.desk_renewal
THEN 'RENEWAL'::text
            ELSE 'CHECKOUT'::text
        END AS type
   FROM action.circulation;
```

Creating views

- Create a variation of the circ_type view:

```
CREATE VIEW newview AS
  SELECT circulation.id,
    CASE
      WHEN circulation.opac_renewal OR
circulation.phone_renewal OR circulation.desk_renewal
        THEN 'RENEWAL'::text
      ELSE 'CHECKOUT'::text
    END AS type
  FROM action.circulation;
```

View exercises

1) Create a view that lists the barcode, library short name, and circulation duration for all books
2) Then use that view in a query to count the number of books that circulated from the library with short name 'BR4' with a duration of 1 hour

Modifying data

INSERT, UPDATE, DELETE
And functions

First: a word about transactions

- A transaction enables you to undo any changes you have made to the database
- Begin with a BEGIN statement to signal the start of the transaction
- Issue as many INSERT/ UPDATE/ DELETE/ ALTER statements as you like
- Then decide whether you want to COMMIT your work, or ROLLBACK your work

COPY data

- Fastest means of getting data into PostgreSQL:

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

- Or from a file:

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

INSERT statements

- Basic INSERT statement

```
INSERT INTO table (column, column, …)
VALUES (value, value, ...)[, (...)];
```

- You can also insert one or more rows via a SELECT statement:

```
INSERT INTO table (column, column, …)
SELECT column, column, … FROM table2 …
```

INSERT (continued)

- You can specify `DEFAULT` for any column value to use the table's defined default value
  - If you do not include the column in your INSERT clause, it supplies the default value
  - If there is no default defined - ***ERROR***
- A common data loading approach:
  - COPY (bulk-load) data into a staging table
  - Munge the data via UPDATEs
  - INSERT...SELECT to insert the data into the production table

INSERT exercises

- Insert two new rows into the table `asset.copy_location`, with an owning library of **4** and names **'Reserves'** and **'Storage'**
- For each row in actor.usr with family_name beginning with 'S', insert a row into actor.usr_note with:
  - title = 'S name', value = the family_name value, creator = 1, and public = TRUE

DELETE statements

- DELETE FROM *table* WHERE *condition*;
- Warning: if you fail to give a condition, the DELETE statement will happily delete all rows from the table
- Delete operations are restricted by relational constraints
- In Evergreen, rules often change a DELETE operation to just set the *deleted* column to true

DELETE exercises

- For each of the following, use a transaction to make and then rollback your changes:
  - DELETE all rows from the actor.usr table. What happens?
  - DELETE all rows from the action.circulation table, SELECT * from action.circulation. What happens?
    - ROLLBACK the transaction and SELECT again. What happens?
  - DELETE FROM actor.usr WHERE id = 15;
    - What happens?

UPDATE statement

- UPDATE *table*
  SET *column* = *value*, *column* = *value*, ...
  WHERE *condition*;
- The UPDATE statement is an odd duck because it almost forces you to rely on subquries instead of joins

UPDATE exercises

- For each of the following, use a transaction to make and then rollback your changes:
  - Set all middle names in the user table to NULL where they are currently an empty string.
  - Set the email address for every user to their user name @example.org.
  - Set the user name for every user to their barcode from actor.card.