From Sudbury to Southbury: Evergreen SQL Bootcamp

Dan Scott
Coffee|Code Consulting
February 18/19, 2010

Agenda

- Day 1:
  - Introduction to SQL databases
  - Basic SELECT statements
- Day 2:
  - Advanced SELECT statements
  - Inserting, updating, and deleting data
  - Specific Bibliomation reports
  - Using the Evergreen reporter interface

Daily agenda

- 9:30 – 11:30: Dan talks and demonstrates
- 11:30 – 12:00: You practice
- 12:00 – 12:30ish: We all eat and make merry
- 12:30ish – 3:00: Dan talks and demonstrates
- 3:00 – 4:00: You practice
- 4:00 - ??: We all go home, eat, and make merry

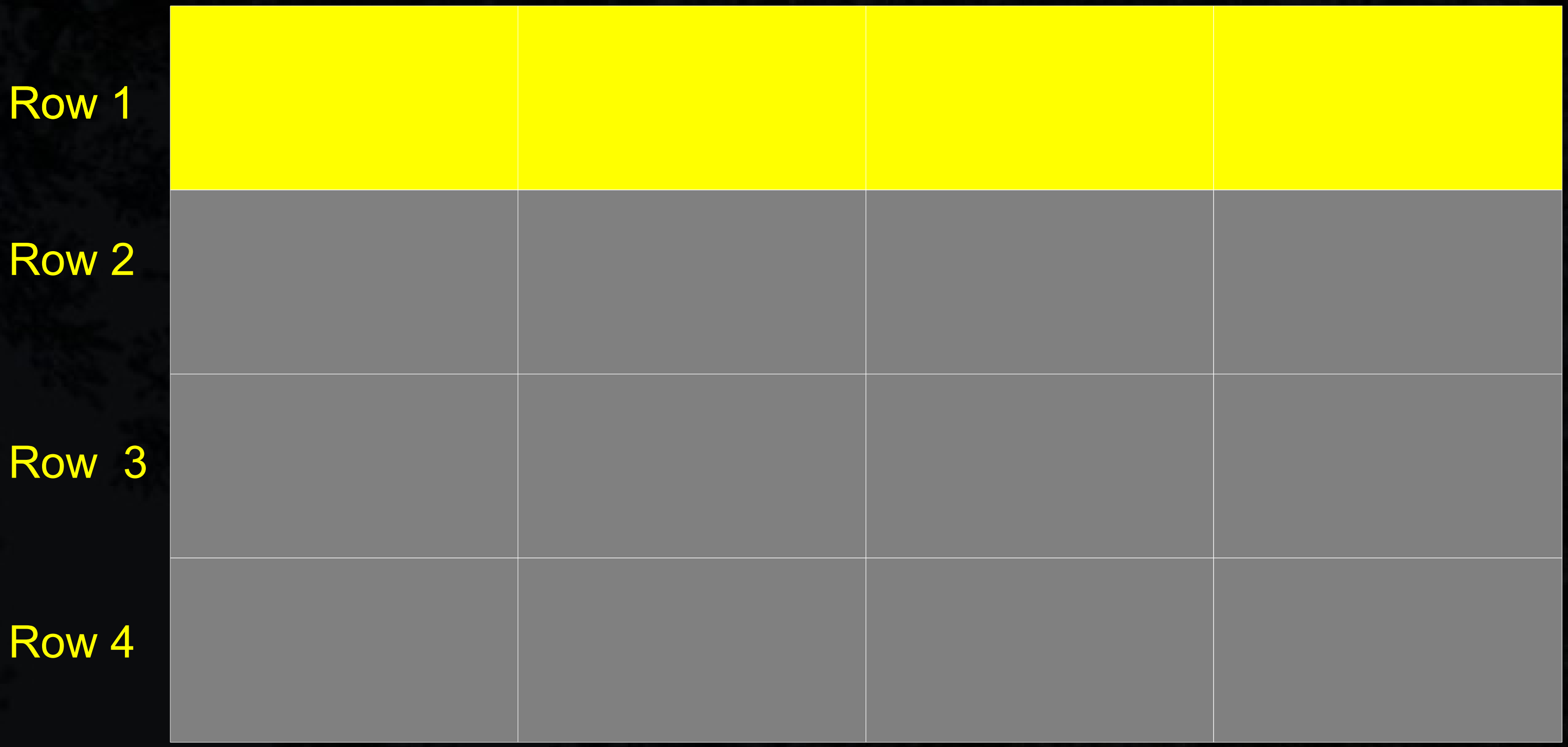
Introducing SQL databases

- 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



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
...

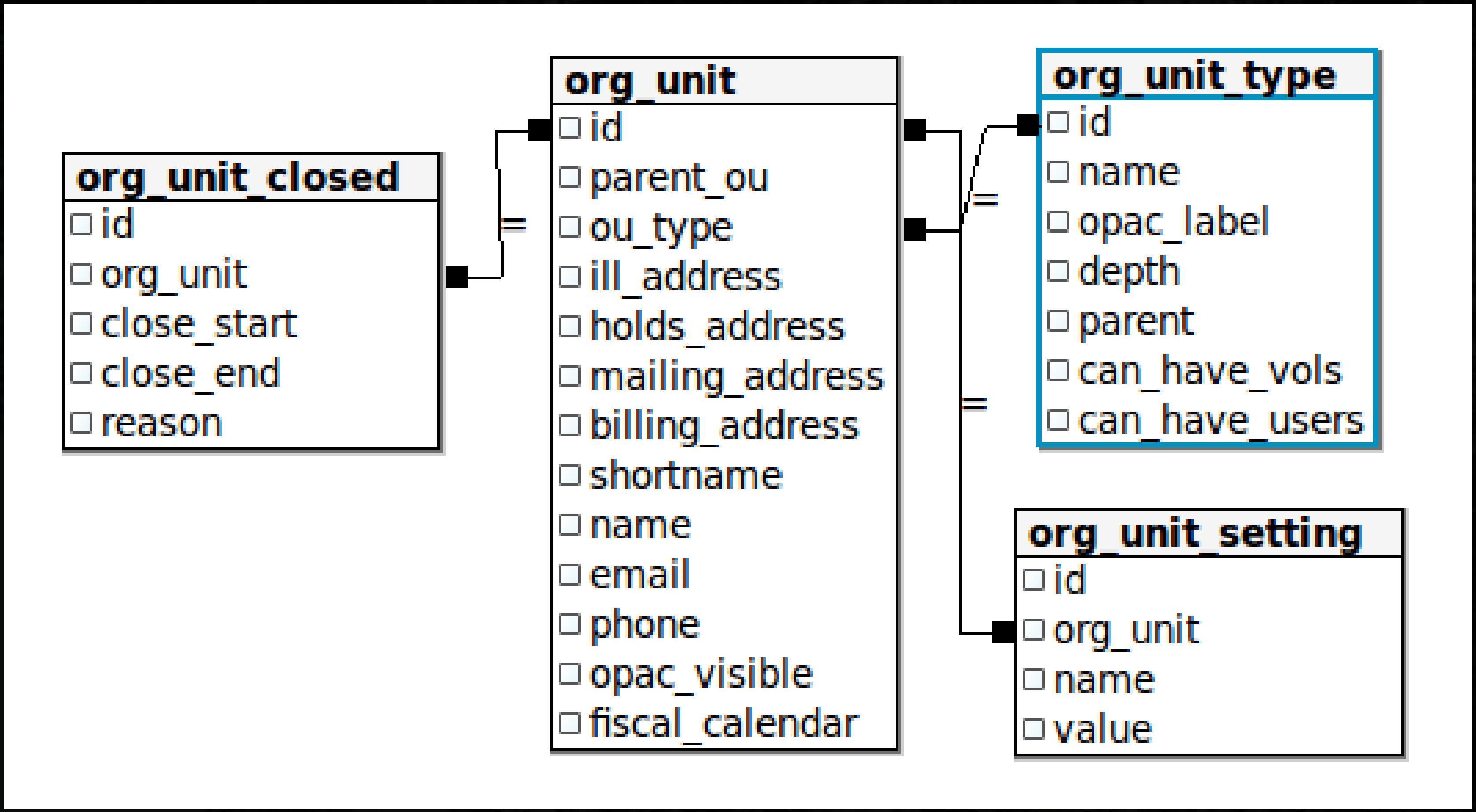
Data types used in Evergreen

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)

Simple relational example



Let's explore the Evergreen schema

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 column values 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, and 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)

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!

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

Agenda, Day 2

- Exercises (simple SELECT queries)
- Advanced SELECT queries
- Inserting, updating, deleting data
- Specific Bibliomation reports

Exercises

1) List all of the values for the first ten users 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 Beacon Falls.
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 record ID, call number, owning library by name, barcode, circulation library by name, title, and author for the first 10 records in the system. Ensure none of the records, call numbers, or barcodes have been deleted.

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

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*
    …
- The INSERT-via-SELECT approach is really useful for data loading

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, a delete operation often sets the *deleted* column to true

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 sub-selects instead of joins
- Practice: Begin a transaction; then set all middle names in the user table to NULL where they are currently an empty string. Then rollback the transaction.

And now... to walk through the Bibliomation reports