How to SQL (Sierra)

Part 2

- **JEREMY GOLDSTEIN**: Minuteman Library Network
- **PHIL SHIRLEY**: Cuyahoga Falls Library
- **RAY VOELKER**: The Public Library of Cincinnati and Hamilton County

  https://iug2019-sql.github.io/

Recap

- Getting started
- PGAdmin III
- Basic Query Statement:
  - Clauses: **`SELECT`**, **`FROM`**, **`WHERE`**, **`GROUP BY`**, etc.
    - Order is important!
  - Comments: `--`
    - used to add comments to statement, or to prevent execution of statement

Recap: Relational Database

- Sierra SQL database is a **relational database**
  - Data is structured in tables
  - Relationships between tables are often defined by **keys**
    - **primary** key
    - **foreign** key

Recap: Keys

sierra_view
record_metadata

primary key

foreign key

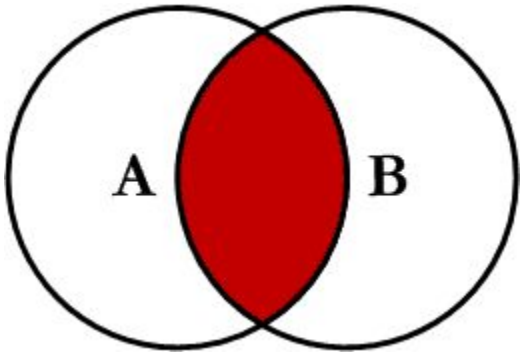
	id bigint	record_type_code character(1)	record_num integer	creation_date_gmt timestamp with time zone
1	420907795009	b	1000001	2012-06-19 18:48:06-04
2	420907795010	b	1000002	2012-06-19 18:48:07-04
3	420907795011	b	1000003	2012-06-19 18:48:07-04
4	420907795012	b	1000004	2012-06-19 18:48:07-04
5	420907795013	b	1000005	2012-06-19 18:48:08-04

sierra_view
bib_record_property

	id integer	bib_record_id bigint	best_title character varying(1000)	publish_year integer
1	357762	420907795009	Water monsters : opposing viewpoints	1991
2	357763	420907795010	Seeking the old paths, and other sermons;	1899
3	357764	420907795011	The Foundation grants index.	1971
4	357765	420907795012	The religion of tomorrow	1899
5	357766	420907795013	Upward steps	1899

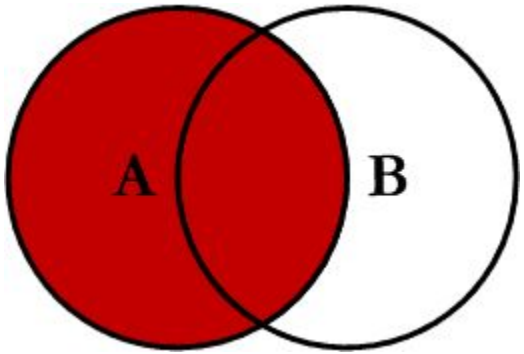
Recap: Join

- `JOIN` (or `INNER JOIN`)
  - Given two sets `A` (left) and `B` (right), performing a `JOIN` will return a set containing all elements of set `A` that also belong to set `B`



Recap: Left Join

- `LEFT JOIN` (or `LEFT OUTER JOIN`)
  - Given two sets `A` (left) and `B` (right) performing this join will return a set containing ALL elements of set `A` AND elements of set `A` that also belong to set `B`



Recap: Left Join (cont.)

- **`LEFT JOIN`** operation will still return data for sets to the *left* when no data exists in the sets to the (right)
  - As you see below, `NULL` values are returned in columns from **`sierra_view.bib_record_property`**

	id bigint	record_type_code character(1)	record_num integer	creation_date_gmt timestamp with time zone	deletion_date_gmt date	num_revisions integer	bib_record_id bigint	best_title character varying(1000)
1	420907795049	b	1000041	2012-06-19 18:48:16-04		2	420907795049	Richard's cork leg.
2	420907795050	b	1000042	2012-06-19 18:48:16-04	2016-01-21	2		
3	420907795051	b	1000043	2012-06-19 18:48:16-04		2	420907795051	Initiative and refe
4	420907795052	b	1000044	2012-06-19 18:48:17-04		2	420907795052	A country without s
5	420907795053	b	1000045	2012-06-19 18:48:17-04		2	420907795053	A new parliamentary

Recap: Left Join (cont.)

SQL statement that produced the previous output:

```
SELECT
r.id, r.record_type_code,
r.record_num, r.creation_date_gmt,
r.deletion_date_gmt, r.num_revisions,
p.bib_record_id, p.best_title

FROM
sierra_view.record_metadata AS r

LEFT OUTER JOIN
sierra_view.bib_record_property AS p
ON
p.bib_record_id = r.id
```

Recap: Subqueries

- Useful for breaking up query into logical, more understandable parts, as well as constraining one-to-many relationships
- Examples:
  - Get names of bib record titles that have a creation date within the last 12 hours
    https://iug2019-sql.github.io/figs/figure_2.1.html
  - Get all patron notes by patron record number (subquery in `SELECT` clause)
    https://iug2019-sql.github.io/figs/figure_2.1.1.html

Agenda

- Why Use SQL
- Let's build a query from a scenario:
  - We want to start producing reports concerning holds that patrons create on different record types
  - Explore a number of concepts along the way
    - Aggregates, case, temp tables, indexes, data types and casting
- Tips and tricks
  - Working with strings
- Some further examples and resources

Why Use SQL?

- **Advantages over other Sierra tools:**
  - Powerful text searching, parsing, formatting
  - Aggregation of data
  - Incorporate mathematical calculations into output
  - Fully customizable
- **Extract otherwise inaccessible data**
  - Sierra user permissions
  - Order and checkin record data across accounting units
  - Reading History
  - Network access table

Why Use SQL (cont.)

- **“Simplicity” / Standardization of SQL Language:**
  - Resources for creating meaningful queries are plentiful
  - SQL skills are transferable to other applications.
- **Can incorporate queries into many useful external applications**
  - Automate reports
  - Add live Sierra data to websites
  - Combine with Sierra APIs to streamline workflows

Let's build a query

- Good place to start is with the Sierra DNA documentation:
  - https://techdocs.iii.com/sierradna/
    - Table concerning holds is in the section `Transactions` -> `Circulation` as table `sierra_view.hold`

hold

Each row of hold describes a bibliographic, item, or volume hold.

Column	Data Type	Not NULL?	Comment
id	bigint	false	System-generated sequential ID.
patron_record_id	bigint	false	Foreign key to patron_record.
record_id	bigint	false	Foreign key to record.
placed_gmt	timestamp	false	Date the hold was placed.
is_frozen	boolean	false	Specifies whether the hold is frozen (suspended).
delay_days	int	false	Stores the "not wanted before" date as a number of days after the date the hold was placed. The maximum value is "180". If a "not wanted before" date was not specified, the value is '0'.
location_code	varchar	false	For bib or volume-level holds, the branch location from which to fill the hold, if the hold is set for 'Limit to Location'. Does not apply to item-level holds (blank).
expires_gmt	timestamp	false	"Not needed after" date.

Let's build a query (cont.)

```
SELECT
*
FROM
sierra_view.hold
LIMIT 10
```

Figure 2.9

	id bigint	patron_record_id bigint	record_id bigint	placed_gmt timestamp with time zone	is_frozen boolean	delay_days integer	location_code character varying(5)	expires_gmt timestamp wi
1	37719995	481037629759	420908702561	2019-04-10 21:05:13-04	f	0		2020-04-09 2
2	37797119	481037639972	420909765303	2019-04-16 13:08:18-04	f	0		2020-04-15 1
3	37408801	481037584945	420910215100	2019-03-23 09:55:02-04	f	0		2020-03-22 0
4	38000841	481038538067	450975189926	2019-04-25 20:00:07-04	f	0		2020-04-24 2
5	35366619	481037418872	420910189903	2018-11-18 15:20:41-05	t	255		2019-11-18 1
6	37408894	481037445657	420910208482	2019-03-23 09:58:32-04	f	0		2020-03-22 0
7	37976614	481037364670	450978381833	2019-04-28 20:56:03-04	f	0		2020-04-27 2
8	37941007	481038546782	450981601433	2019-04-25 20:18:21-04	f	0		2020-04-24 2
9	37976599	481037828512	450981522751	2019-04-12 15:15:01-04	f	0		2020-04-11 1
10	37941015	481037430701	450977235008	2019-04-25 21:33:47-04	f	0		2020-04-24 2

Let's build a query: Aggregate

- Getting a sense of the scope of the holds:
  - Running a query to gather a **COUNT()**, by type (bib, item, volume level holds): We'll use the **GROUP BY** clause

```
SELECT
r.record_type_code,
COUNT(r.record_type_code) as count_holds

FROM
sierra_view.hold AS h

JOIN
sierra_view.record_metadata as r
ON
  r.id = h.record_id

GROUP BY
r.record_type_code
```

Let's build a query: Aggregate (cont.)

- Output of that query breaks down the numbers by type:

	record_type_code character(1)	count_holds bigint
1	b	181033
2	i	51836
3	j	6780

`b` = bib level holds
`i` = item level holds
`j` = volume level holds

- How about next breaking that up by patron type?

Let's build a query: Aggregate (cont.)

```
SELECT
r.record_type_code,
p.ptype_code,
COUNT(r.record_type_code) as count_holds
FROM
sierra_view.hold AS h
JOIN
sierra_view.record_metadata AS r
ON
  r.id = h.record_id
JOIN
sierra_view.patron_record AS p
ON
  p.record_id = h.patron_record_id
GROUP BY
r.record_type_code,
p.ptype_code
ORDER BY
r.record_type_code,
p.ptype_code
```

Figure 12

- Notice that we now **`JOIN sierra_view.patron_record`** to bring in the **`ptype_code`**
- **`sierra_view.patron_record`** was added to the **`GROUP BY`** clause to be aggregated as well
  - Note that all columns selected need to be in the **`GROUP BY`** clause as well
- The aggregate function **`COUNT()`** returns a count of those groupings

Let's build a query: Aggregate (cont.)

Previous query output (partial)...

	record_type_code character(1)	ptype_code smallint	count_holds bigint
1	b	0	166991
2	b	1	93
3	b	2	58
4	b	3	122
5	b	5	23
6	b	6	60
7	b	10	1319
8	b	12	2298
9	b	15	204
10	b	22	1065
11	b	32	1092
12	b	51	38
13	b	196	7180
14	i	0	43661
15	i	1	218
16	i	2	68
17	i	3	76
18	i	5	41
19	i	6	54
20	i	10	2219
21	i	12	830

- Output still consists of **record_type_code**, but now also aggregates on another column, **ptype_code**
- These two columns are aggregated together in the **COUNT()** function and are represented by the column **count_holds**

Figure 13

Let's build a query: Aggregate (cont.)

- Suppose now we wanted to filter or constrain the results to groups of `**ptype_code**` that had a **COUNT()** of holds above a certain threshold?
  - **WHERE** clause **won't** work on aggregates
  - **HAVING** clause **will** work on aggregates

Let's build a query: Aggregate (cont.)

```
SELECT
r.record_type_code,
p.ptype_code,
COUNT(*) as count_holds
FROM
sierra_view.hold AS h
JOIN
sierra_view.record_metadata AS r
ON
  r.id = h.record_id
JOIN
sierra_view.patron_record AS p
ON
  p.record_id = h.patron_record_id
GROUP BY
r.record_type_code,
p.ptype_code
HAVING
COUNT(*) > 1000
ORDER BY
r.record_type_code,
p.ptype_code
```

- Using the **HAVING** clause below, we're able to limit to the patron types having more than 1000 holds of each of the hold level types (`b`, `i`, `j`)

Figure 14:
https://iug2019-sql.github.io/figs/figure_2.14.html

Let's build a query: Aggregate (cont.)

Previous query results ...

	record_type_code character(1)	ptype_code smallint	count_holds bigint
1	b	0	166940
2	b	10	1394
3	b	12	2275
4	b	22	1065
5	b	32	1080
6	b	196	7308
7	i	0	42152
8	i	10	2106
9	i	196	4130
10	j	0	6455

Figure 15

Let's build a query: Aggregate (cont.)

- Other useful aggregates:
  - MIN()
  - MAX()
  - AVG()
  - SUM()

```
SELECT
MIN(h.placed_gmt) AS min_hold_placed,
MAX(h.placed_gmt) AS max_hold_placed,
AVG(
        AGE(h.placed_gmt)
) AS avg_age_hold,
-- this isn't very useful to us, but demonstrates `SUM()`
EXTRACT(
        YEARS FROM SUM(
                AGE(h.placed_gmt)
        )
) AS sum_years_holds

FROM
sierra_view.hold as h
```

Figure 16

Let's build a query: Aggregate (cont.)

- Previous query output...

	min_hold_placed timestamp with time zone	max_hold_placed timestamp with time zone	avg_age_hold interval	sum_years_holds double precision
1	2012-07-04 01:01:01-04	2019-03-15 11:34:23-04	1 mon 21 days 18:48:33.415516	27058

Figure 17

Let's build a query: Temp Tables

- We're interesting in examining holds now from a "**supply and demand**" perspective:
  - We'd like to **resolve** each hold to a `**bib_record_id**` so we could get a sense of the counts of holds on each title.
  - A hold in the hold table is on a `**record_id**`, which could be for bib (`**b**`), item (`**i**`), or volume (`**j**`) level

Let's build a query: Temp Tables (cont.)

- Lets create a **`TEMPORARY TABLE`** (or, `TEMP TABLE`) with data from multiple tables to help simplify things…
  - These tables are removed after a session is ended
    https://www.postgresql.org/docs/current/sql-createtable.html#AEN67422
  - Useful to:
    - Simplify / make a statement more logical
    - Speed up other parts of the query (create indexes, etc)

Let's build a query: Temp Tables (cont.)

```
DROP TABLE IF EXISTS temp_hold_data;

CREATE TEMP TABLE temp_hold_data AS
SELECT
r.record_type_code, r.record_num,
p.ptype_code, h.*
FROM
sierra_view.hold AS h
JOIN
sierra_view.record_metadata AS r
ON
  r.id = h.record_id
JOIN
sierra_view.patron_record AS p
ON
  p.record_id = h.patron_record_id
;
```

Figure 18

- **DROP TABLE** clause helps if you're going to modify the query, and re-run it (to avoid an error on multiple runs)
- We bring in data about the record type (`**r.record_type_code**`),the patron type (`**p.ptype_code**`), and all the rest of the data concerning the hold (`**h.***`)
- We can work with our temp table in subsequent statements, as long as it's the **same session**

Let's build a query: Temp Tables (cont.)

- The previous **`TEMP TABLE`** query only tells us what ***type*** of **record** the hold was for.
- How do we resolve record types that are not bib (`b`) to the bib record they're linked to?
- **`CASE`** statement, can be used to produce different results depending on a **conditional expression**

Let's build a query: Temp Tables (cont.)

- **CASE** statement:
  - Allows for the execution of a block of code conditionally
  - Similar to IF / THEN / ELSE
  - Tip: make sure *something* is returned, if the main conditions are not met!

Let's build a query: CASE

```
CASE
WHEN r.record_type_code = 'i' THEN (
        SELECT
        l.bib_record_id
        FROM
        sierra_view.bib_record_item_record_link as l
        WHERE
        l.item_record_id = h.record_id
        LIMIT 1
)
WHEN r.record_type_code = 'j' THEN (
        SELECT
        l.bib_record_id
        FROM
        sierra_view.bib_record_volume_record_link as l
        WHERE
        l.volume_record_id = h.record_id
        LIMIT 1
)
WHEN r.record_type_code = 'b' THEN (
        h.record_id
)
ELSE NULL
END AS bib_record_id,
```

- This section of the **partial** SQL statement demonstrates resolving item (`**i**`) and volume (`**j**`) to the `**bib_record_id**` that they are linked to.
- Full **`TEMP TABLE`** creation: Figure 19.1: https://iug2019-sql.github.io/figs/figure_2.19.1.html

Let's build a query: `WITH` clause

- Now that we have our **`TEMP TABLE`**, **`temp_hold_data`** we can do some more with it
- We can also simplify things by using **`WITH`** clause to create a **Common Table Expression** (**CTE**)
  - **CTE** can be thought of as defining temporary tables that exist just for one query
  - This is just one ***optional*** method that can be used to simplify logic of a complex SQL statement

Let's build a query: `WITH` clause (cont.)

```
WITH distinct_titles AS (
        SELECT
        t.bib_record_id,
        string_agg(t.pickup_location_code::TEXT, ',') AS pickup_locations,
        COUNT(*) as count_holds_title
        FROM
        temp_hold_data as t
        GROUP BY
        t.bib_record_id
)

SELECT
d.*
FROM
distinct_titles AS d
ORDER BY
d.count_holds_title DESC
;
```

Figure 20:
https://iug2019-sql.github.io/figs/figure_2.20.html

Let's build a query: `WITH` clause (cont.)

	bib_record_id bigint	pickup_locations text	count_holds_title bigint
1	420910219176	ba,ha,re,dt,sh,an,cr,ha,an,md,ha,wt,pl,1,ch,sm,1,wh,ge,lv,	3062
2	420910212190	ch,ba,cv,an,gr,grw,fo,an,av,ha,gr,1,dp,mo,mw,nr,ch,dt,ha,g	3037
3	420910217875	sb,cv,1,re,sm,ha,wy,sm,av,gr,hp,ge,dt,sm,sb,ge,dt,ma,oa,sh	2914
4	420910219177	co,lv,1,dp,ba,dt,an,ha,wh,ma,mw,fo,gh,ha,mo,sb,sh,gr,rew,c	2817
5	420910214745	gr,dp,oa,lv,ba,dt,ns,hp,ma,ge,sm,lv,ww,1,sb,sm,sm,hp,mn,dp	2816
6	420910221212	av,oa,nr,sh,fo,sm,mn,mn,ge,ha,sb,md,1,co,sb,mn,sm,ep,grw,m	2793
7	420910219178	sm,cv,ge,gh,mn,ha,ba,ba,ma,sm,ba,lv,an,ma,oa,ma,1,mo,cv,1,	2763
8	420910207644	dt,gh,ns,lv,pl,ge,co,sh,gr,1,mm,hp,dt,dt,sm,an,wy,mn,re,ma	2740
9	420910216470	lv,nr,re,sb,an,mt,wh,sm,an,1,ba,mm,an,wh,wy,nw,ge,md,dp,ha	2692
10	420910221213	ba,1,wt,cv,oa,hp,1,1,ba,rew,re,fo,ba,mm,mo,wy,mn,md,sm,ww,	2651
11	420910219175	mo,av,ha,wy,mn,1,sb,ch,cr,ch,ns,sm,sm,ch,pl,sh,ha,ma,sb,wh	2646
12	420910221214	gr,mw,cl,an,cv,ge,dt,sm,wh,md,sm,ge,ha,bh,ns,sm,fo,mt,pl,w	2630
13	420910216469	ww,lv,ge,sm,1,nw,sh,1,md,os,wy,lv,an,nr,ns,wt,ha,1,mo,sm,l	2622
14	420910216471	mw,gr,sm,ma,ba,oa,ch,hp,ha,hp,ma,nr,wt,sm,fo,oa,nw,ch,oa,l	2597
15	420910222250	an,1,ha,ch,gr,gr,wh,ma,mo,grw,dp,1,ma,lv,dp,sh,an,sb,sh,wy	2550
16	420910212189	ge,1,ma,1,wh,nw,sm,mn,sh,sm,lv,mm,wy,ge,rew,gr,wy,oa,hp,mc	2548
17	420910214744	ch,wh,mn,oa,dt,nw,ww,mm,lv,nr,sm,pl,1,cv,ch,mo,sh,mn,an,ha	2493

Let's build a query: STRING_AGG()

```
string_agg(t.pickup_location_code::TEXT, ',') AS pickup_locations,
```

- From previous query, the PostgreSQL **`STRING_AGG()`** function allows us to create a list (delimited by the `,` character) of the `pickup_location_code` values for each title
- **`STRING_AGG()`** takes a **`TEXT`** data type as the first argument, and a **`TEXT`** data type as the delimiter
- https://www.postgresql.org/docs/current/functions-aggregate.html

Data Types & Casting

https://www.postgresql.org/docs/current/datatype.html

- Some important and common PostgreSQL data types to understand
  - **`INTEGER`**: signed, four-byte integer (`1`, `-1`, `42`, etc)
  - **`NUMERIC`**: real number or **`NUMERIC(p,s)`** with p digits with s number after the decimal point
  - **`TEXT`**: character string with unlimited length
  - **`CHAR`**: single character, or `CHAR(n)` fixed-length of `n` characters with *space padded*
  - **`VARCHAR(n)`**: variable-length character string of `n` characters with *no space padded*
  - **`BOOLEAN`**: true or false values (can use special `IS TRUE` or `IS FALSE` clause to test)

Data Types & Casting (cont.)

https://www.postgresql.org/docs/current/datatype-datetime.html

- Date / Time Types:
  - **`DATE`**: ISO 8601 (`YYYY-MM-DD`):
    `2019-03-17`
  - **`TIMESTAMP`**: ISO 8601 date with time (24-hour clock):
    `2019-03-17 11:41:13.979849`
    Time zone is optional
    `TIMESTAMP WITH TIME ZONE`:
    `2019-03-17 11:41:13.979849-04`

Data Types & Casting (cont.)

https://www.postgresql.org/docs/current/datatype-datetime.html

- Date / Time Types (cont.):
  - **INTERVAL**: defines periods of time
    - Traditional Postgres format:
      `1 year 2 months 3 days 4 hours 5 minutes 6 seconds`
    - Useful in defining ranges of time limit in **WHERE** clause

```
SELECT
*
FROM
sierra_view.circ_trans AS c
WHERE
c.transaction_gmt <= NOW() - '1 hour'::INTERVAL
AND c.transaction_gmt > NOW() - '2 hours'::INTERVAL
ORDER BY
c.transaction_gmt
```

Figure 23

Data Types & Casting (cont.)

- Casting one data type to another is necessary to perform some operations: `::` or **`CAST(`*`expression AS type`*`)`**
  (`CAST` example here: https://iug2019-sql.github.io/figs/figure_2.23.1.html)
  - From the previous query example:

```
c.transaction_gmt <= NOW() - '1 hour'::INTERVAL
```

- The string value `1 hour` is being converted to the **`INTERVAL`** data type, so that it may be included in an operation (subtraction) involving another date format
  - **`TIMESTAMP`** data type is returned from the function, **`NOW()`**

Working With Date Types

- **`NOW()`** will return current timestamp
- Use **`::`** to convert data types
- **`TO_CHAR()`** can be used for date and timestamp formatting
- Remember that ISO 8601 (**`YYYY-MM-DD`**) can be useful for sorting!

```
SELECT
now(),
now()::DATE,
DATE(now()),
to_char(now(), 'MM-DD-YYYY'),
to_char(now(), 'MON-DD-YYYY'),
to_char(now(), 'Day Month DD, YYYY')
```

now timestamp with time zone	now date	date date	to_char text	to_char text	to_char text
2019-03-15 15:29:38.7211-04	2019-03-15	2019-03-15	03-15-2019	MAR-15-2019	Friday March 15, 2019

- Template Patterns for Date/Time Formatting can be found here: https://www.postgresql.org/docs/current/functions-formatting.html

Let's build a query: INDEX

- Returning to our example, we were working with a **TEMP TABLE**: https://iug2019-sql.github.io/figs/figure_2.20.html
- What if our query is slow?
- Queries can be made to run significantly more quickly when an **INDEX** is used!
- Adding the **CREATE INDEX** statement to the query:

```
CREATE INDEX temp_hold_data_bib_record_id ON temp_hold_data(bib_record_id);
ANALYZE temp_hold_data;
```

https://iug2019-sql.github.io/figs/figure_2.26.html

Let's build a query: `INDEX`

- Creating good indexes can be useful when building a **`TEMP TABLE`** that will be used in **multiple** or **complex queries** involving a **`JOIN`** or **`GROUP BY`** operation.
  - Keep in mind that a index scan is better than a sequential scan when doing an operation on columns.
  - Further reading about using indexes:
    - http://www.postgresqltutorial.com/postgresql-indexes/postgresql-index-types/
    - https://use-the-index-luke.com

Let's build a query (cont.)

- Here's the query script up to this point: https://iug2019-sql.github.io/figs/figure_2.28.html
- We want to bring in some counts of available items.
  - To keep things simple, we're going to limit to:
    - Holds that are bib level
    - Holds placed by patrons with ptype_code = 0

Let's build a query (cont.)

```
SELECT
COUNT(*)
FROM
sierra_view.bib_record_item_record_link AS l
JOIN
sierra_view.item_record AS i
ON
  i.record_id = l.item_record_id
LEFT OUTER JOIN
sierra_view.checkout as c
ON
  c.item_record_id = l.item_record_id
WHERE
l.bib_record_id = d.bib_record_id
-- item has a status code of something that we'd want to see
AND i.item_status_code IN (
        '-', '!', 'b', 'p', '(', '@', ')', '_', '=', '+'
)
AND COALESCE(
        --if this age is >= 60 days, it'll return FALSE,
        -- and not count as an "available item"
        age(c.due_gmt) < '60 days'::INTERVAL,
        -- if there is no due_gmt value (NULL) return TRUE
        TRUE
)
```

- Statement will count available items meeting certain criteria:
  - `item_status_code`
  - `due_gmt`

Figure 29:
https://iug2019-sql.github.io/figs/figure_2.29.html

Let's build a query: COALESCE()

```
AND COALESCE(
        --if this age is >= 60 days, it'll return FALSE,
        -- and not count as an "available item"
        age(c.due_gmt) < '60 days'::INTERVAL,
        -- if there is no due_gmt value (NULL) return TRUE
        TRUE
)
```

Figure 30

- **`COALESCE()`**: Returns the ***first argument*** that is not **`NULL`**
- In the example above, '**`c.due_gmt`**` could have a value of **`NULL`**` (remember `LEFT OUTER JOIN`?)
- If age of due date is greater or equal to 60 days, we get a value of **`FALSE`**
- Otherwise, we get a value of **`TRUE`** and can consider the item to be "active"

Let's build a query: Final Output

- "Final" bib level holds to available item query: https://iug2019-sql.github.io/figs/figure_2.31.html

	bib_record_id bigint	bib_record_num text	count_holds_title bigint	count_items_available bigint	hold_to_item_ratio text	best_title character varying(1000)	pickup_locations text
1	420907797479	b1002471a	1	0		Luciano Pavarotti premieres Verdi [sound recording] : [first	pr
2	420907799032	b1004024a	1	1	1.00	Otto of the Silver Hand.	dt
3	420907799561	b1004553a	1	1	1.00	Milestones [sound recording]	co
4	420907799835	b1004827a	1	2	.50	Giving you the best that I got [sound recording]	pr
5	420907800116	b1005108a	1	2	.50	The miracle of mindfulness : a manual on meditation	an
6	420907801727	b1006719a	1	1	1.00	B.B. King live at the Regal [sound recording]	ww
7	420907801789	b1006781a	1	1	1.00	In the age of the smart machine : the future of work and pow	oa
8	420907802767	b1007759a	1	2	.50	Orthodoxy.	mo
9	420907803146	b1008138a	1	13	.08	Lolita	l
10	420907803182	b1008174a	1	3	.33	Dead man's folly	cl
11	420907803201	b1008193a	2	1	2.00	Notes of a native son.	ww,md
12	420907803535	b1008527a	1	5	.20	Player piano	ch

Figure 32:
https://iug2019-sql.github.io/figs/figure_2.32.png

Let's build a query: Final Output (cont.)

- https://iug2019-sql.github.io/figs/figure_2.31.html
- Please note the following things about this final SQL statement:
  - We created a **second** **`TEMP TABLE`** called "**`temp_title_item_counts`**", to more easily make the final calculation for **`` `hold_to_item_ratio` ``** (which is the ratio between holds: **`` `count_holds_title` ``** and available items: **`` `count_items_available` ``**
  - NOTE that this is also a simplified output of the bib level holds *only*
  - *Does anyone know why we have a* ***`CASE`*** *clause checking to see if* ***`` `count_items_available` ``*** *is equal to zero?*

Tips and Tricks

- Orders of operations and parentheses are important!

```sql
-- find holds placed up to 2 days ago, ready for pickup
SELECT
h.id,
AGE(h.placed_gmt) as hold_age,
h.status
FROM
sierra_view.hold AS h
WHERE
h.placed_gmt >= NOW() - '2 days'::INTERVAL
AND h.status = 'b' OR h.status = 'j' OR h.status = 'i'
ORDER BY
hold_age DESC
LIMIT 10
```

	id bigint	hold_age interval	status character(1)
1	30931202	1 year 1 mon 4 days 10:16:33	i
2	37179891	11 mons 22 days 11:52:21	i
3	37229026	11 mons 18 days 06:05:51	i
4	37161396	10 mons 7 days 03:10:43	i
5	37206717	9 mons 27 days 06:39:22	i
6	36944773	9 mons 12 days 05:51:45	i
7	37262863	8 mons 28 days 09:41:10	i
8	37228182	8 mons 24 days 09:34:25	i
9	37184688	8 mons 6 days 12:57:22	i
10	37109094	7 mons 16 days 12:19:58	i

```sql
-- find holds placed up to 2 days ago, ready for pickup
SELECT
h.id,
AGE(h.placed_gmt) as hold_age,
h.status
FROM
sierra_view.hold AS h
WHERE
h.placed_gmt >= NOW() - '2 days'::INTERVAL
-- note the added '(', ')'
AND (h.status = 'b' OR h.status = 'j' OR h.status = 'i')
ORDER BY
hold_age DESC
LIMIT 10
```

	id bigint	hold_age interval	status character(1)
1	37291801	1 day 16:12:34	b
2	37292521	1 day 15:59:29	b
3	37292557	1 day 15:45:01	b
4	37292362	1 day 15:44:00	b
5	37292181	1 day 15:31:42	b
6	37292623	1 day 15:31:01	b
7	37291032	1 day 15:21:56	b
8	37295434	1 day 15:09:48	b
9	37291922	1 day 14:58:39	b
10	37297738	1 day 14:53:18	b

Figure 35

String Functions

- PostgreSQL has many **String Functions** / **Operators** available
  - Functions allow you to modify, parse, and search within strings
  - Includes POSIX regex and simplified pattern matching
  - https://www.postgresql.org/docs/9.1/functions-string.html

CONCAT

- Use concatenation to chain strings together
- Three methods available: **CONCAT()**, **CONCAT_WS()**, **||**

```
SELECT
CONCAT(code, name),
CONCAT_WS(', ',code, name),
code || ' (' || name || ')'
FROM
sierra_view.location_myuser
WHERE
code = 'acta'
```

concat text	concat_ws text	?column? text
actaACTON/Adult	acta, ACTON/Adult	acta (ACTON/Adult)

CONCAT and COALESCE

- Be careful with `NULL` values!
  - This results in a `NULL` value:

```
SELECT
NULL || 'concatinate this!' AS output
```

Figure 33

- To avoid this type of behaviour, consider using the **CONCAT()** or **COALESCE()** functions: https://iug2019-sql.github.io/figs/figure_2.34.html

```
SELECT
NULL || 'concatinate this!' AS output,
-- result: NULL

-- CONCAT will take "unlimited" variables
CONCAT(NULL, 'concatinate this!', NULL, '!') AS output2,
-- result: 'concatinate this!!'

-- COALESCE() will return ONLY the first non-null value
COALESCE(NULL, '', ' hello?') || 'concatinate this!' as output3
-- result: 'concatinate this!'
```

Figure 34

Nesting String Functions

Using string functions to display an author in first name, last name order

```
SELECT
b.best_author AS original,
SPLIT_PART(b.best_author,' (',1) AS author_1,
SPLIT_PART(SPLIT_PART(b.best_author,' (',1),', ',2) AS author_2,
REPLACE(SPLIT_PART(SPLIT_PART(b.best_author,' (',1),', ',2),'.','') AS author_3,
REPLACE(SPLIT_PART(SPLIT_PART(b.best_author,' (',1),', ',2),'.','')
    ||' '||SPLIT_PART(b.best_author,', ',1) AS author_4
FROM
sierra_view.bib_record_property b
WHERE
best_author LIKE 'Sharma, Robin S. (Robin Shilip), 1964- author%'
```

original character varying(1000)	author_1 text	author_2 text	author_3 text	author_4 text
Sharma, Robin S. (Robin Shilip), 1964- author.	Sharma, Robin S.	Robin S.	Robin S	Robin S Sharma

Pattern Matching: LIKE

- LIKE provides a simple pattern matching option
- Two Wildcards
  - '**_**' single instance of any character
  - '**%**' any number of characters (including 0)
- Here we are finding all locations starting with 'act'

```
SELECT
code
FROM
sierra_view.location_myuser
WHERE
code LIKE 'act%'
```

code character varying(5)
actas
actap
actae
actal
actan
actnn
actjl
actjn
actjh
actjt
actjp
actjr
actyn

Pattern Matching: POSIX Regex

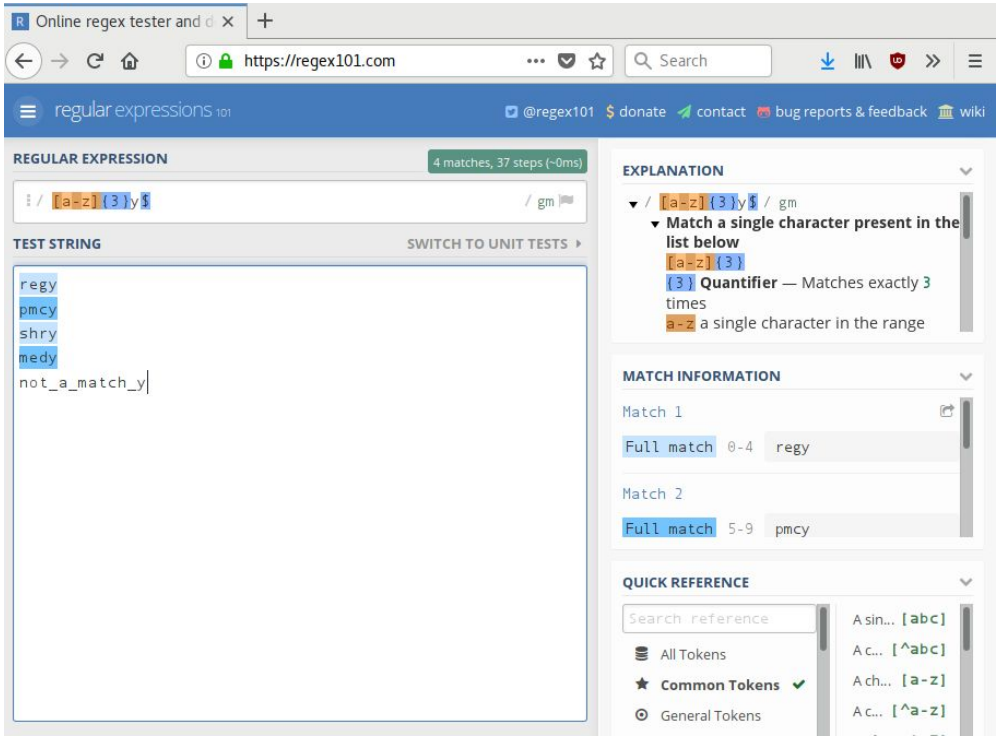
- POSIX regex operators: `~`, `~*`, `!~`, ` !~*`
  - Matches and Not matches
  - With and without case sensitivity
- Here we are finding all locations containing 4 lowercase letters and ending in y

```
SELECT
code
FROM
sierra_view.location_myuser
WHERE
code ~ '[a-z]{3}y$'
```

code character varying(5)
regy
pmcy
shry
medy
ashy
nory
wwdy
mayy
somy
blmy
camy
mily
wsny
neey
arly

Pattern Matching: POSIX Regex

Regular Expression tip: Use the site regex101.com to test



Pattern Matching: Regex Functions

- **SUBSTRING()** extracts a specified set of characters from a string
- Can accomplish this two ways
  - Regex '^[a-z]{3}': extract 3 lowercase letters from start
  - Positionally 'FROM 1 FOR 3': extract 3 letters starting at 1st character

```
SELECT
code,
SUBSTRING(code, '^.{3}') as substring_regex,
SUBSTRING(code FROM 1 FOR 3) as substring_extract
FROM
sierra_view.location_myuser
ORDER BY
code
```

	code character varying(5)	substring_regex text	substring_extract text
1	1		1
2	1cj	1cj	1cj
3	1cjar	1cj	1cj
4	1cjau	1cj	1cj
5	1cjbd	1cj	1cj
6	1cjbg	1cj	1cj
7	1cjbi	1cj	1cj
8	1cjeb	1cj	1cj
9	1cjer	1cj	1cj
10	1cjfl	1cj	1cj
11	1cjgn	1cj	1cj

Pattern Matching: Regex Functions

- **regexp_matches()**
  Returns matches on POSIX regular expression against a string

```
-- return ISBN number (or null, if number doesn't match expected format of ISBN
-- and record number associated with the ISBN
SELECT
v.record_id,
(regexp_matches(
        v.field_content,
        -- the regex to match on (10 or 13 digits, with the possibility of the
        -- 'X' character in the check-digit spot)
        '[0-9]{9,10}[x]{0,1}|[0-9]{12,13}[x]{0,1}',
        -- regex flags; ignore case
        'i'
))[1]::varchar(30) as search_isbn

FROM
sierra_view.varfield AS v

WHERE
v.marc_tag || v.varfield_type_code = '020i'

LIMIT 100
```

	record_id bigint	search_isbn character varying(30)
1	420908754250	093503739X
2	420908085848	0262193019
3	420908086376	0936889187
4	420908085851	0812017803
5	420909037215	0395299152
6	420908085855	0404114210
7	420908085858	0801425433
8	420908085858	0801497787
9	420908272594	0819185108
10	420908085861	0800628268

Figure 36

String Functions Cont

Some other useful functions to know

LOWER()
UPPER()
INITCAP()
REVERSE()
LENGTH()

LEFT()
TRIM()
REGEXP_MATCHES()
REGEXP_REPLACE()

Tables of Note: Linking Records

- `sierra_view.*_record_link` type tables contain primary keys for *both* record types, therefore *linking* them
  - Examples:
    - **`bib_record_item_record_link`**
    - **`bib_record_volume_record_link`**
    - **`bib_record_order_record_link`**
  - Useful for:
    - Gather record counts
    - Chain data types together without having to touch record tables

Tables of Note: Linking Records (cont.)

```
-- get all the items linked to a specific bib
SELECT
r.id,
l.bib_record_id, l.item_record_id,
l.items_display_order
FROM
sierra_view.record_metadata as r
JOIN
sierra_view.bib_record_item_record_link as l
ON
  l.bib_record_id = r.id
WHERE
r.record_type_code = 'b'
AND r.id = 420909710085
ORDER BY
l.items_display_order
```

	id bigint	bib_record_id bigint	item_record_id bigint	items_display_order integer
1	420909710085	420909710085	450979030326	0
2	420909710085	420909710085	450979030325	1
3	420909710085	420909710085	450979030333	2
4	420909710085	420909710085	450979030330	3
5	420909710085	420909710085	450979030321	4
6	420909710085	420909710085	450979030320	5
7	420909710085	420909710085	450979030332	6
8	420909710085	420909710085	450979030322	7
9	420909710085	420909710085	450979030316	8
10	420909710085	420909710085	450979197059	9
11	420909710085	420909710085	450979030324	10
12	420909710085	420909710085	450979030329	11
13	420909710085	420909710085	450979030319	12
14	420909710085	420909710085	450979030317	13
15	420909710085	420909710085	450979030318	14
16	420909710085	420909710085	450979030331	15
17	420909710085	420909710085	450979030327	16

Unique to SierraDNA queries

```
--Provides usage count of the reading history feature

SELECT
p.home_library_code,
COUNT(p.is_reading_history_opt_in)
FROM
sierra_view.patron_record p
GROUP BY 1
ORDER BY 1
```

Other unique fields for fun queries:

- record_metadata.deletion_date_gmt
  - Count deleted records
- varfield.occ_num
  - Pick out first occurrences of varfields such as ISBN
- bool_info.sql_query
  - See sql queries underlying create list searches

Further examples

- Useful resources on GitHub:
  - Links to these presentations: https://github.com/iug2019-sql/iug2019-sql.github.io
  - Tips and Tricks: https://github.com/iug2019-sql/iug2019-sql.github.io/blob/master/tips_and_tricks.md

Further examples

- Useful resources on GitHub (cont.):
  - **The Public Library of Cincinnati and Hamilton County**: https://github.com/plch/sierra-sql/wiki
  - **Minuteman Library Network**: https://github.com/jmgold/SQL-Queries/wiki
  - **The University of North Carolina at Chapel Hill**: https://github.com/UNC-Libraries/III-Sierra-SQL/wiki

Consider Attending

- Automating Booklist Curation with SQL
  - Tuesday 1:30-2:30 Deer Valley
- Cache and Release: Capturing and Using Sierra's Temporary SQL Data
  - Wednesday 3:00-4:00 Deer Valley
- SQL Users Birds of A Feather

Find Us On Slack

All three of us can be found on the Sierra_ILS slack workspace, managed by Craig Bowman, Jeremy and Ray

Questions?

Jeremy Goldstein jgoldstein@minlib.net

Phil Shirley pshirley@fallslibrary.org

Ray Voelker ray.voelker@cincinnatilibrary.org