Data 101: Data Engineering
Midterm Exam
Exam Reference Packet

UC Berkeley, Fall 2024

October 16, 2024

Name: ____________________

Email: ____________________@berkeley.edu

Student ID: ____________________

Instructions

Do not open this exam reference packet until you are instructed to do so.

Make sure to write your name, email, and SID on this cover page.

1 Library Database Description

This schema is a simplified version of a local library system. Broadly, it tracks books which are stored at various locations. Users can check out books from a specific location.

- A library is made of several locations, each of which has its own set of books.
- Users can check out and return books from a specific location. A book which is actively checked out will have a `checkout_date` and a `due_date`, but a `NULL` `return_date`. All three of these attributes are the SQL `DATE` type.
- Each book has an ISBN associated with it. ISBNs are internationally *unique* numbers that are assigned to books by a publisher. We will assume that all ISBNs are 13 digits in the following format: `978-6-543-21012-3`.
- The type `SERIAL` is an auto-incrementing (unique) integer (starting from 1) that PostgreSQL manages for each record which is inserted into the table.
- We have intentionally removed all primary keys, foreign keys, and indexes from this schema.

```
CREATE TABLE locations (
  id SERIAL,
  name TEXT NOT NULL,
  address TEXT NOT NULL,
  phone_number VARCHAR(20)
);

CREATE TABLE users (
  id SERIAL,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  email TEXT UNIQUE NOT NULL,
  phone_number VARCHAR(20),
  joined_date DATE
    DEFAULT CURRENT_DATE
);

CREATE TABLE books (
  id SERIAL,
  title TEXT NOT NULL,
  author TEXT NOT NULL,
  isbn TEXT UNIQUE NOT NULL,
  publication_year INTEGER
);

CREATE TABLE book_locations (
  id SERIAL,
  book_id INTEGER,
  location_id INTEGER,
  total_copies INTEGER NOT NULL,
  available_copies INTEGER NOT NULL
);

CREATE TABLE checkouts (
  id SERIAL,
  user_id INTEGER,
  book_id INTEGER,
  location_id INTEGER,
  checkout_date DATE
    DEFAULT CURRENT_DATE,
  return_date DATE,
  due_date DATE NOT NULL
);
```

2 Ed Database Description

We consider a simplified, text-only version of Ed discussion forum for a single Data 101 course. Users can both start new threads and write comments on existing threads. Users can also make hearts (i.e., likes or upvotes) on threads or comments within threads.

Table	Description	Details
`users`	All students and course staff in the current Data 101 course.	N/A
`threads`	Each thread has an original post `text` and corresponding `title`.	Thread hearts are the number of hearts (i.e, likes) on the original post text.
`comments`	Comments are all user text posts on a thread that are not the original post text.	Comment hearts are the number of hearts (i.e., likes) on that single comment.
`child_comments`	Parent-child edges of all comment trees in all threads.	A comment tree for Comment C is formed when other comments are written in reply to C or (recursively) in reply to a comment in C's comment tree. C's comment tree can be described by its set of parent-child edges.

Below, Figures 1 and 2a are example threads. Figure 2b shows the comment tree for Comment ID 22 (from Figure 2a); note 21 is a parent and not part of Comment ID 22's comment tree.

On the next page are the database schema and the corresponding sample tables for these figures.

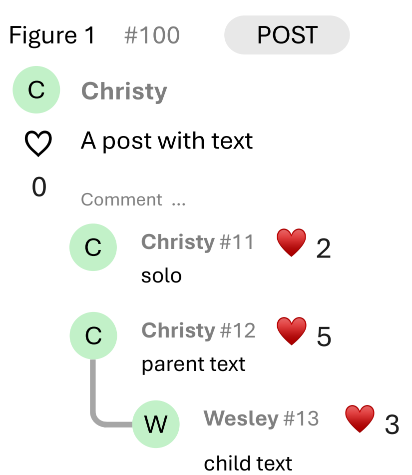

Figure 1

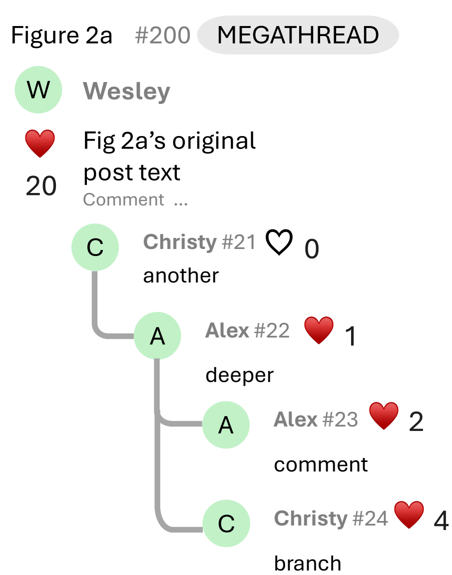

Figure 2a

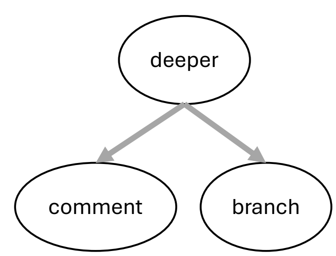

Figure 2b

Ed discussion forum database schema:

```
CREATE TABLE users (
  id INTEGER PRIMARY KEY,
  email TEXT,
  display_name TEXT,
  hearts_made INTEGER
);

CREATE TABLE threads (
  id INTEGER PRIMARY KEY,
  thread_type TEXT NOT NULL,
  hearts INTEGER,
  user_id INTEGER
    REFERENCES users(id),
  title TEXT,
  text TEXT
);
```

```
CREATE TABLE comments (
  id INTEGER PRIMARY KEY,
  thread_id INTEGER REFERENCES threads(id),
  hearts INTEGER,
  user_id INTEGER REFERENCES users(id),
  text TEXT
);

CREATE TABLE child_comments (
  comment_id INTEGER REFERENCES comments(id),
  child_id INTEGER REFERENCES comments(id)
);
```

Sample tables for the two Ed threads shown on the previous page:

id	email	display_name	hearts_made
9001	wz@...	Wesley	9
9002	cq@...	Christy	15
9003	ag@...	Alex	13

`users`

id	thread_type	hearts	user_id	title	text
100	Post	0	9002	Figure 1	A post with text
200	Megathread	20	9001	Figure 2a	Fig 2a’s original post text

`threads`

id	thread_id	hearts	user_id	text
11	100	2	9002	solo
12	100	5	9002	parent text
13	100	3	9001	child text
21	200	0	9002	another
22	200	1	9003	deeper
23	200	2	9003	comment
24	200	4	9002	branch

`comments`

comment_id	child_id
12	13
21	22
22	23
22	24

`child_comments`

3 SQL Timestamps

We discuss a few ways (among many) of storing date and time values ("date/time values") in a single, combined attribute:

- **SQL timestamp**: The SQL standard `timestamp with time zone` datatype, which stores date/time values relative to the UTC timezone (Coordinated Universal Time, or Greenwich Mean Time), which is 7 hours ahead of Pacific Daylight Time (PDT). The SQL timestamp datatype occupies **8 bytes**.
- **Epoch time**: Otherwise known as UNIX Time, epoch time is measured in seconds since the Unix epoch, or January 1st 1970 UTC (Coordinated Universal Time). Unix times (at least until the year 2038) can be stored into a **4 byte `INTEGER`**.
- **String**: A text string that represents the SQL timestamp (UTC) in a standard format. For simplicity, we use the format used to display SQL timestamps, occupying **20 bytes**.
- July 7, 2018 3:09:11pm PDT is July 7, 2018, 10:09:11pm UTC is 1531001351 in epoch time is `'2018-07-07 22:09:11'`.

The below psql extended output demonstrates how to convert between the timestamp, epoch time, and string (`TEXT`) representations of the date/time value July 7th, 2018, 10:09:11pm UTC.

```
WITH example(ts_orig, epoch_orig, text_orig) AS (
    VALUES ('2018-07-07 22:09:11'::TIMESTAMP, 1531001351,
            '2018-07-07 22:09:11')
)
SELECT
    ts_orig,
    EXTRACT('EPOCH' FROM ts_orig)              AS ts_to_epoch,
    epoch_orig,
    TO_TIMESTAMP(epoch_orig) AT TIME ZONE 'UTC' AS epoch_to_ts_utc,
    TO_TIMESTAMP(epoch_orig) AT TIME ZONE 'PDT' AS epoch_to_ts_pdt,
    text_orig,
    text_orig::TIMESTAMP                        AS text_to_ts_utc,
    EXTRACT('EPOCH' FROM text_orig::TIMESTAMP)  AS text_to_epoch
FROM example;

-[ RECORD 1 ]---+--------------------
ts_orig         | 2018-07-07 22:09:11
ts_to_epoch     | 1531001351.000000
epoch_orig      | 1531001351
epoch_to_ts_utc | 2018-07-07 22:09:11
epoch_to_ts_pdt | 2018-07-07 15:09:11
text_orig       | 2018-07-07 22:09:11
text_to_ts      | 2018-07-07 22:09:11
text_to_epoch   | 1531001351.000000
```

4 (Excerpt) PostgreSQL Recursive Queries

The optional RECURSIVE modifier changes WITH from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. Using RECURSIVE, a WITH query can refer to its own output.
A very simple example is this query to sum the integers from 1 through 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t
    WHERE n < 100
)
SELECT sum(n) FROM t;
```

The general form of a recursive `WITH` query is always a *non-recursive term*, then `UNION` (or `UNION ALL`), then a *recursive term*, where only the recursive term can contain a reference to the query's own output. Such a query is executed as follows:

Recursive Query Evaluation

1. Evaluate the non-recursive term. For `UNION` (but not `UNION ALL`), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary *working table*.
2. So long as the working table is not empty, repeat these steps:
   (a) Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For `UNION` (but not `UNION ALL`), discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary *intermediate table*.
   (b) Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.

In the example above, the working table has just a single row in each step, and it takes on the values from 1 through 100 in successive steps. In the 100th step, there is no output because of the WHERE clause, and so the query terminates.

Recursive queries are typically used to deal with hierarchical or tree-structured data. A useful example is this query to find all the direct and indirect sub-parts of a product, given only a table that shows immediate inclusions:

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity * pr.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part;
```

5 PostgreSQL Reference

```
[ WITH with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ]
                          [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

where `from_item` can be one of:

```
    table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
                [ TABLESAMPLE sampling_method ( argument [, ...] ) ]
    [ LATERAL ] ( select ) [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item join_type from_item { ON join_condition |
                            USING ( join_column [, ...] )
              [ AS join_using_alias ] }
    from_item NATURAL join_type from_item
    from_item CROSS JOIN from_item
```

and `grouping_element` can be one of: `expression` or `( expression [, ...] )`

and `with_query` is:

```
    with_query_name [ ( column_name [, ...] ) ] AS ( SELECT | VALUES )
```

5.1 Window Functions

```
<window or agg_func> OVER (
  [PARTITION BY <...>] [ORDER BY <...>] [RANGE BETWEEN range_start AND range_end] )
```

where `<window or agg_func>` can be one of:

```
    aggregate functions: AVG, SUM, etc., or:
    RANK() -- ordering within the window
    LEAD/LAG(exp, n) -- value of exp that is n ahead/behind in the window
    PERCENT_RANK() -- relative rank of current row as a %
    NTH_VALUE(exp, n) -- value of exp @ position n in window
```

and `range_start` and `range_end` can be one of:

```
    UNBOUNDED PRECEDING, UNBOUNDED FOLLOWING, CURRENT ROW,
    offset PRECEDING, offset FOLLOWING
```

5.2 Example Queries

```
SELECT id, location, age,
  AVG(age) OVER () AS avg_age
FROM residents;

SELECT id, location, age,
  SUM(age) OVER (
    PARTITION BY location
    ORDER BY age
    RANGE BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING ) AS a_sum
FROM residents
ORDER BY location, age;


CREATE TABLE <relation name> AS ( <subquery> );
CREATE TABLE zips (
    location VARCHAR(20) NOT NULL,
    zipcode INTEGER,
    in_district BOOLEAN DEFAULT False,
    PRIMARY KEY (location),
    UNIQUE (location, zipcode)
);

DROP TABLE [IF EXISTS] <relation name>;
ALTER TABLE zips
        ADD avg_pop REAL,
        DROP in_district;


CREATE TABLE cast_info (
  person_id INTEGER,
  movie_id INTEGER,
  FOREIGN KEY (person_id) REFERENCES actors (id)
    ON DELETE SET NULL ON UPDATE CASCADE,
  FOREIGN KEY (movie_id) REFERENCES movies(id) ON DELETE SET NULL
);
```

6 PostgreSQL String Utilities

String utility functions:

- `string || string` → text (concatenation)
- `SUBSTRING( string FROM start)` → text
- `SUBSTRING( string FROM re_pattern )` → text
- `SUBSTR( string, count )` → text
- `REPLACE(source, pattern, replacement)` → text
  In `REPLACE` pattern operates similar to `LIKE`, not a regular expression.
- `REGEXP_REPLACE(source, re_pattern, replacement, flags)` → text
  Note: `flags` must be `'g'` to execute a global match replacing all instances.
- SQL supports matching strings using two different types of pattern matching: SQL-style `LIKE` patterns, and POSIX Regular Expressions.
  - `string LIKE pattern` → boolean
  - `string ~ re_pattern` → boolean

Examples:

```
'Hello' || 'World' → 'HelloWorld'
STRPOS('Hello', 'el') → 2
SUBSTRING('Thomas' FROM 3) → 'omas'
SUBSTRING('Hello', 2, 3) → 'ell'
SUBSTR('Hello World', 7) → ‘World’
```

See the next page for Section 7: SQL Pattern Matching, which includes regular expressions.

7 SQL Pattern Matching

7.1 LIKE Patterns

SQL's `LIKE`, and `REPLACE` functions operate using a simplified pattern syntax.

```
'abc' LIKE 'abc' →   true                'abc' LIKE '_b_' →   true
'abc' LIKE 'a%'  →   true                'abc' LIKE 'c'   →   false
REPLACE('Hello World', 'l', 'L') → 'HeLLo WorLd'
```

If `pattern` does not contain percent signs (%) or underscores (_), then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore in pattern stands for (matches) any single character; a percent sign matches any sequence of zero or more characters.

7.2 Regular Expressions

This is an abbreviated reference which may prove helpful. The functions ~, `REGEXP_REPLACE`, and `SUBSTRING` accept `re_pattern` arguments which are regular expressions.

Escapes	Shorthand used in a match
`\d`	matches any digit
`\s`	matches any white space character
`\w`	matches any word character
Constraints	Used at the beginning or end of a match
`^`	matches at the beginning of the string
`$`	matches at the end of the string
Quantifier	Used after a match section
`*`	a sequence of 0 or more matches of the atom
`+`	a sequence of 1 or more matches of the atom
`?`	a sequence of 0 or 1 matches of the atom
`{m}`	a sequence of exactly m matches of the atom
`{m,}`	a sequence of m or more matches of the atom
`{m,n}`	a sequence of m through n (inclusive) matches of the atom; m cannot exceed n

```
'abcd' ~ 'a.c'      →  true        dot matches any character
'abcd' ~ 'a.*d'     →  true        * repeats the preceding pattern item
'abcd' ~ '(b|x)'    →  true        | means OR, parentheses group
'abcd' ~ '^a'       →  true        ^ anchors to start of string
'abcd' ~ '^(b|c)'   →  false
substring('foobar' from 'o.b')    → 'oob'
substring('foobar' from 'o(.)b')  →  'o'
substring('Thomas' from '...\$')  → 'mas'
regexp_replace('foobarbaz', 'b..', 'X')          → 'fooXbaz'
regexp_replace('foobarbaz', 'b..', 'X', 'g')     → 'fooXX'
regexp_replace('Hello World', '[aeiou]', '-', 'g') → 'H-ll- W-rld'
```