Data 101: Data Engineering
Final Exam
Exam Reference Packet

UC Berkeley, Fall 2024

December 17, 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 FEMA Database Description

The United States Federal Emergency Management Agency (FEMA) maintains OpenFEMA, an open-source database "in support of FEMA's mission to help people before, during, and after disasters." We show sample records of two tables below.

`incidents` contains incident records for all federally declared disasters.

- `id TEXT`: Record ID.
- `disaster_num INTEGER`: Sequentially assigned number used to designate an event or incident declared as a disaster.
- `incident_type TEXT`: The primary or official type of incident, e.g., fire, flood, etc.
- `begin_date DATE`: Date the incident began.
- `end_date DATE`: Date the incident ended.
- `fips_state CHAR(2)`: FIPS two-digit numeric code used to identify the United States, the District of Columbia, US territories, outlying areas of the US and freely associated states. FIPS codes range from `00` to `99`.
- `declaration_title TEXT` Title for the disaster.

id	disaster_num	incident_type	begin_date	end_date	fips_state	declaration_title
fbb566	4829	Hurricane	2024-09-24		45	Hurricane Helene
2b7c14	4837	Tropical Storm	2024-09-15	2024-09-19	37	Potential Tropical Cyclone Eight
7cae4e	5534	Fire	2024-09-01		41	Rail Ridge Fire
30962a	5534	Fire	2024-09-01		41	Rail Ridge Fire
0a7d78	4037	Hurricane	2011-08-24	2011-08-30	10	Hurricane Irene

`grants` contains financial assistance information about disasters.

- `id TEXT`: Unique ID assigned to the record.
- `disaster_num INTEGER`: Sequentially assigned number used to designate an event or incident declared as a disaster.
- `ia_approved_num INTEGER`: The number of disaster assistance applications that were approved for Individual Assistance (IA).
- `ia_updated DATE`: The date the Individual Assistance (IA) data was updated.

[Exam clarification]: The `ia_approved_num` value for record ID `20fc79` should be 1, not 0.

id	disaster_num	ia_approved_num	ia_updated
bd0ea1	4037	272	2024-12-12
44230e	4829	224337	2024-12-11
20fc79	5534	1	2024-12-12

2 Yelp Database Description

The Yelp dataset contains `users`, and `businesses` which are each collections of JSON documents. Here we've shown example documents which also include a default `_id` key from MongoDB.

Note: Compared to the dataset used in project 4, we have adapted the `friends`, `elite`, and `categories` fields to be native lists (as opposed to comas-separated string values).

2.1 A user

```
{
  "_id": ObjectId("..."),
  "user_id": "kBBwlCcbL1s4NVK3g",
  "name": "Jane",
  "review_count": 1220,
  "yelping_since": "2005-03-14",
  "useful": 15038,
  "funny": 10030,
  "cool": 11291,
  "elite": [2006, ..., 2013, 2014],
  "friends": ["xBDpTUbai0DXrvxCe3X16Q",
    "gsiJiB8MGewErmvQIvnGxw"],
  "fans": 1357,
  "average_stars": 3.85,
  "compliment_hot": 1710,
  "compliment_more": 163,
  "compliment_profile": 190,
  "compliment_cute": 361,
  "compliment_list": 147,
  "compliment_note": 1212,
  "compliment_plain": 5691,
  "compliment_cool": 2541,
  "compliment_funny": 2541,
  "compliment_writer": 815,
  "compliment_photos": 323
}
```

2.2 A business

```
{
  "_id": ObjectId("..."),
  "business_id": "2HFDym3zjuRg0shjw",
  "name": "Oskar Blues Taproom",
  "address": "921 Pearl St",
  "city": "Boulder",
  "state": "CO",
  "postal_code": "80302",
  "latitude": 40.0175444,
  "longitude": -105.2833481,
  "stars": 4.0,
  "review_count": 86,
  "is_open": 1,
  "attributes": {
    "RestaurantsTableService": "True",
    "OutdoorSeating": "True",
    ...
    "BusinessAcceptsBitcoin": "False",
    "RestaurantsPriceRange2": "2",
    "RestaurantsAttire": ""casual"",
    "RestaurantsDelivery": "None"
  },
  "categories": ["Gastropubs", ...,
    "Restaurants", "Breweries"],
  "hours": {
    "Monday": "11:00-23:00",
    "Tuesday": "11:00-23:00",
    "Wednesday": "11:00-23:00",
    "Thursday": "11:00-23:00",
    "Friday": "11:00-23:00",
    "Saturday": "11:00-23:00",
    "Sunday": "11:00-23:00"
  }
}
```

3 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.

This database is an expanded version of the midterm library schema. You **do not need to know** the midterm schema to understand this expanded dataset, but in case it is useful, the differences are also listed below.

- A library is made of several locations, each of which has its own set of books.
- A `book` tracks information about a book, but the library stores (and loans out) individual `book_copies` of each book. Each book may have 1 or more copies.
- Each `book_copy` has a status (e.g., "available" or "on loan") and is stored at a specific `location`.
- Users can check out and return a `book_copy` from a specific location. Each checkout is one entry in the `checkouts` table, which gets its own `id`. A book is actively checked out if there is no entry for the corresponding `checkout_id` in the `book_returns` table.
- The type `SERIAL` is an auto-incrementing (unique) INTEGER (starting from 1) that PostgreSQL manages for each record which is inserted into the table.

Differences from the midterm schema:

- The `checkouts` table used to have a column for `return_date`. This has been changed to be its own table.
- The midterm schema did not track individual `book_copies`.
- The midterm schema had a `book_locations` table, which has essentially been migrated to the `book_copies`, which tracks an individual book's current location.

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

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

CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT NOT NULL,
    phone_number TEXT,
    is_ya BOOLEAN DEFAULT false,
    joined_date DATE NOT NULL
);
```

```
CREATE TABLE book_copies (
    id SERIAL PRIMARY KEY,
    book_id INTEGER NOT NULL
        REFERENCES books(id),
    location_id INTEGER NOT NULL
        REFERENCES locations(id),
    acquisition_date DATE,
    status TEXT DEFAULT 'available'
);

CREATE TABLE checkouts (
    id SERIAL PRIMARY KEY,
    user_id INTEGER
        REFERENCES users(id),
    book_copy_id INTEGER
        REFERENCES book_copies(id),
    location_id INTEGER
        REFERENCES locations(id),
    checkout_date DATE,
    due_date DATE NOT NULL
);

CREATE TABLE book_returns (
    id SERIAL PRIMARY KEY,
    checkout_id INTEGER
        REFERENCES checkouts(id),
    location_id INTEGER
        REFERENCES locations(id),
    return_date DATE NOT NULL
);
```

4 OLAP in SQL

This page contains excerpts from the official PostgreSQL documentation.

7.2.4. GROUPING SETS, CUBE, and ROLLUP

More complex grouping operations than those described above are possible using the concept of grouping sets. The data selected by the FROM and WHERE clauses is grouped separately by each specified grouping set, aggregates computed for each group just as for simple GROUP BY clauses, and then the results returned. For example:

```
=> SELECT * FROM items_sold;

 brand | size | sales
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
```

```
=> SELECT brand, size, sum(sales)
FROM items_sold
GROUP BY GROUPING SETS ((brand), (size), ());
 brand | size | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
       |      |  50
```

Each sublist of GROUPING SETS may specify zero or more columns or expressions and is interpreted the same way as though it were directly in the GROUP BY clause. An empty grouping set means that all rows are aggregated down to a single group (which is output even if no input rows were present), as described above for the case of aggregate functions with no GROUP BY clause.

A shorthand notation is provided for specifying two common types of grouping set. A clause of the form
`ROLLUP ( e1, e2, e3, ... )` represents the given list of expressions and all prefixes of the list including the empty list; thus it is equivalent to

```
GROUPING SETS (
    ( e1, e2, e3, ... ), ... ( e1, e2 ), ( e1 ), ( )
)
```

This is commonly used for analysis over hierarchical data; e.g., total salary by department, division, and company-wide total. A clause of the form `CUBE ( e1, e2, ... )` represents the given list and all of its possible subsets (i.e., the power set). Thus `CUBE ( a, b, c )` is equivalent to

```
GROUPING SETS (
    ( a, b, c ), ( a, b    ), ( a,    c ), ( a       ),
    (    b, c ), (    b    ), (       c ), (         )
)
```

The CUBE and ROLLUP constructs can be used either directly in the GROUP BY clause, or nested inside a GROUPING SETS clause. If one GROUPING SETS clause is nested inside another, the effect is the same as if all the elements of the inner clause had been written directly in the outer clause.

5 SUMIF: Google Sheets

`SUMIF(range, criterion, [sum_range])` Returns a conditional sum across a range.

- `range` - The range which is tested against `criterion`.
- `criterion` - The pattern or test to apply to `range`.
  - If `range` contains numbers (or booleans) to check against, `criterion` may be either a string or a number (or boolean). If a number is provided, each cell in `range` is checked for equality with `criterion`. Otherwise, `criterion` may be a string containing a number (which also checks for equality), or a number prefixed with any of the following operators: = (checks for equality), > (checks that the range cell value is greater than the criterion value), or < (checks that the range cell value is less than the criterion value)
  - If `range` contains text to check against, `criterion` must be a string. `criterion` can contain wildcards [...omitted...]. A string criterion must be enclosed in quotation marks. Each cell in `range` is then checked against `criterion` for equality (or match, if wildcards are used).
- `sum_range` - The range to be summed, if different from `range`.

Sample Usage:

- `SUMIF(A1:A10,">20")`
- `SUMIF(A1:A10,"Paid",B1:B10)`

	A	B	C	D	E
1	**Expense**	**Today**			
2	Coffee	$4.00			
3	Newspaper	$1.00			
4	Taxi	$10.00			
5	Golf	$26.00			
6	Taxi	$8.00			
7	Coffee	$3.50			
8	Gas	$46.00			
9	Restaurant	$31.00			
10					
11					
12	**range**	**criteria**	**sum_range**	**Sum**	**Formula**
13	A2:A9	Taxi	B2:B9	$18.00	=SUMIF(A2:A9, "Taxi", B2:B9)
14					
15					
16	**range**	**criteria**	**sum_range**	**Sum**	**Formula**
17	B2:B9	>=10	B2:B9	$113.00	=SUMIF(B2:B9, ">=10", B2:B9)
18					
19					
20	**range**	**criteria**	**sum_range**	**Sum**	**Formula**
21	B2:B9	>=10		$113.00	=SUMIF(B2:B9, ">=10")

6 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 )
```

6.1 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
```

6.2 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;

SELECT id, location, age,
  AVG(age) OVER () AS avg_age,
  CASE WHEN location = 94720 THEN 'UC Berkeley'
       WHEN location = 94709 THEN 'city of Berkeley'
       WHEN location = 94105 THEN 'San Francisco'
       ELSE 'Unknown Zip Code'
       END AS city_name
FROM residents;
```

```
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
);
```

7 PostgreSQL String Utilities

*This section is **not** needed for the final exam, but we include it for completeness.*

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.

8 SQL Pattern Matching

*This section is **not** needed for the final exam, but we include it for completeness.*

8.1 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.

8.2 Regular Expressions: An Abbreviated Reference

The functions ~, REGEXP_REPLACE, and SUBSTRING accept `re_pattern` regex arguments.

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'
```

9 MQL: Mongo Query Language Reference

In MongoDB we commonly query by a `collection`. The argument to a query function is usually one or more objects (dicts) that represent a series of steps.

9.1 Query Functions

```
db.collection.find({});
db.collection.findOne({});

db.collection.find(
    {  category: "peace" },
    { _id: 0, category: 1, year: 1,
        laureates.firstname: 1,
        laureates.surname: 1
    }
)
.sort({ year: 1, category: -1 })
.limit(2);

db.collection.aggregate( [
    { stage: {...} },
    { stage: {...} }
] );
```

9.2 Aggregation Stage References

The `stage` is often one these functions, though there are many more:

```
$match
$project
$sort/$limit
$group, e.g., {
"$group" :
    {  "_id" : "$item",
        "totalqty" : {"$sum" : "$instock.qty" } }
}

$unwind, e.g., { $unwind: "$instock" }
$lookup, e.g., {
    $lookup : {
        from : "inventory",
        localField : "instock.loc",
        foreignField : "instock.loc",
        as :"otheritems"
    }
}
```

10 Odds and Ends

10.1 Base-2 Logarithm Lookup Table

From Wikipedia: For a real number N, the binary logarithm ($\log_2 n$) is the power to which the number 2 must be raised to obtain the value n.

$$N = \log_2 n \Leftrightarrow 2^N = n$$

N	2^n	$\log_2(n)$
1	2^0	0
2	2^1	1
4	2^2	2
8	2^3	3
16	2^4	4
32	2^5	5
64	2^6	6
128	2^7	7
256	2^8	8
512	2^9	9
1024	2^{10}	10
2048	2^{11}	11
4096	2^{12}	12

10.2 Functional Dependencies

A functional dependency (FD) is a form of constraint between 2 sets of attributes in a relation. For a relational instance with attributes X, Y, and Z:

- The FD $X \rightarrow Y$ is satisfied if for every pair of tuples $t1$ and $t2$ in the instance, if $t1.X = t2.X$, then $t1.Y = t2.Y$.
- The FD $AB \rightarrow C$ is satisfied if for every pair of tuples $t1$ and $t2$ in the instance, if $t1.A = t2.A$ and $t1.B = t2.B$, then $t1.C = t2.C$.

10.3 Median Absolute Deviation

- For a dataset X with median $\tilde{X} = \text{median}(X)$, the **Median Absolute Deviation (MAD)** is $\text{MAD}(X) = \text{median}(|X_i - \tilde{X}|)$.
- The Minimum Description Length (MDL) for encoding a set of values c in a set of types H is $\text{MDL} = \min_{T \in H} \sum_{v \in c} (I_T(v) \log(|T|) + (1 - I_T(v))\text{len}(v))$
- where $I_T(v)$ is an indicator for if v “fits” in type T (with $|T|$ distinct values), log is base 2, and $\text{len}(v)$ is the cost for encoding a value v in some default type.

10.4 Map Reduce

```
Map(k, v) → <k', v'>*
```

- Takes a key-value pair and outputs a set of key-value pairs
- There is one Map function call for each (k,v) pair

```
Reduce(k', <v'>*) → <k', v''>*
```

- All values v' with same key k' are reduced together and processed in v' order
- There is one Reduce function call for each unique key k'

10.5 Entity Relationship (ER) Diagrams

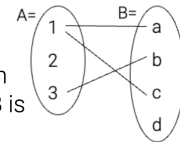
Entity set (rectangles)

- Entities: things, objects, etc.;
- Entity sets: sets entities w/commonalities.
- Every entity set is required to have a primary key (underlined attribute).

Attributes (ovals)
Atomic features connected to entity sets or relationships.

Relationships (diamonds)

- Connects entity sets.
- A relationship between the entity sets A and B is a subset of A x B.



Edges in ER Diagrams can be directed/undirected and represent constraints on subset A x B.

- Undirected edge (with no arrows): no constraints
- Directed edge (arrow): constrains, or determines, the relation to be at most one.
- Bolded edge determines the relation to be at least one.