Data 101/Info 258: Data Engineering
Final Exam
Exam Reference Packet

UC Berkeley, Spring 2025

May 14, 2025

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 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 )
```

1.1 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
```

1.2 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
);
```

2 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.

3 SQL Pattern Matching

3.1 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.

3.2 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'
```

4 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;
```

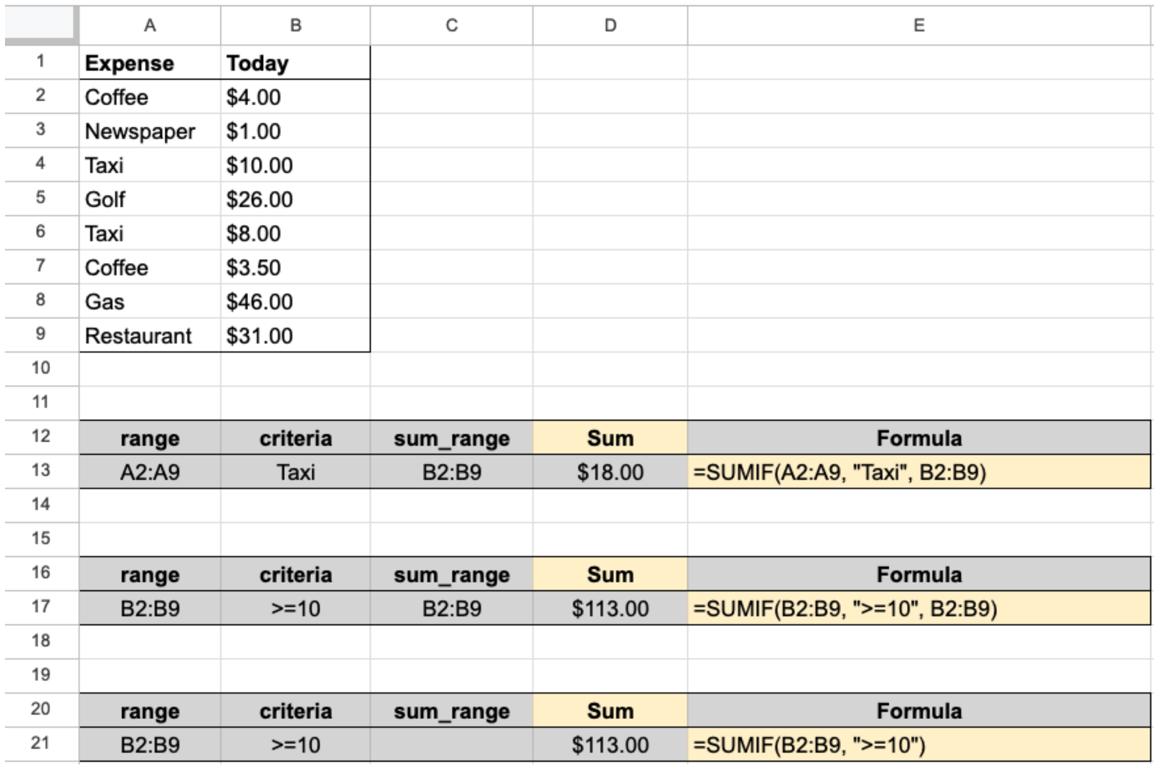
5 SUMIF: Google Sheets

`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 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.

6.1 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: {...} }
] );
```

6.2 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"
    }
}
```

7 Odds and Ends

7.1 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 $X \to Y$ 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 $AB \to C$ 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 $t1.C = t2.C$.

7.2 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.

7.3 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'

7.4 Entity Relationship (ER) Diagrams

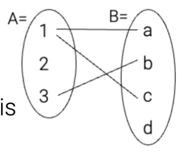
Entity set (rectangles)

- 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.

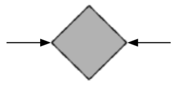
Relationships (diamonds)

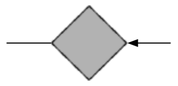
- Connects entity sets.
- A relationship between the entity sets A and B is a subset of A x B.

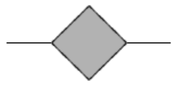


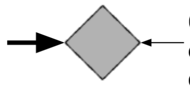
Edges in ER Diagrams can be directed/undirected and represent constraints on subset A x B.

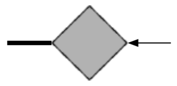
- Undirected edge (with no arrows): no constraints
- Directed edge (arrow): constrains, or determines, the relation to be at most one.
- Bolded edge determines the relation to be at least one.

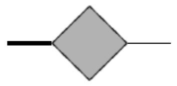

One-one: One on LHS connected to at most one of RHS, and vice-versa


Many-one: One on LHS connected to many on RHS


Many-many: One on LHS connected to many/few on the RHS, and vice versa


One-one: One on LHS connected to exactly one of RHS (≤ 1 & ≥ 1); one on RHS connected to at most one on LHS


Many-one: One on LHS connected to at least one on RHS; one on RHS connected to at most one on LHS


Many-many: One on LHS connected to at least one on RHS; RHS unconstrained