New Rust Code Challenge

You are tasked with creating a queryparser which will be used by downstream services to construct a global view of data flow between tables. The downstream services need a precise column lineage and how the upstream columns are defined in order to correctly construct lineage, and you need to define and implement the interface between your queryparser and the downstream service.

You may use any OSS libraries to help you out, but you must implement the analysis algorithms themselves. For example, please use sqlparser to handle parsing SQL queries.

With the implementation please leave a brief description of the strategy you used to generate lineage and calculate column usage (as defined below), and how to interpret the output. More docs going into the details is nice, but not required.

For this challenge, the scope of SQL will be limited to the following SQL query forms:

Queries

- SELECT queries
- UNION queries

Expressions

- (Qualified) Identifier expressions
- Add expressions
- Comparison expressions
- Basic Functions (`SUM`, `COUNT`, etc.)
- Boolean expressions (`AND`, `OR`)

Simplifications from standard SQL

- For all non-trivial expressions, if shows up in the output list of a select, it will always be aliased.
- For any other query/expression, it is ok to `panic!()` instead of properly handling those as errors.

- You may assume that everything is case-sensitive, even though normal SQL is case-insensitive
- You do not need to handle wildcard expressions, e.g. `select * from ...` or `select ident.* from ...`

Along with the query, you will be provided a list of columns which may appear in the query, as a csv.

For all examples given here, use the columns csv. But for your challenge you need to work with any csv in this format.

```
database_name,schema_name,table_name,column_name
prod,integrations,orders,id
prod,integrations,orders,item_name
prod,integrations,orders,customer_name
prod,integrations,orders,price
prod,integrations,order_items,id
prod,integrations,order_items,order_id
prod,integrations,order_items,date
prod,integrations,order_items,count
prod,platform,order_items,order_id
prod,platform,order_items,date
prod,platform,order_items,item_name
prod,platform,order_items,count
```

Part 1: Column Lineage

You should do a dataflow analysis to figure out the upstream columns - where each column's data comes from. For this analysis, we care about the data the query output columns, not columns are used as to filter the outputs (for example, we don't care about what columns in the where clause).

You can assume that all columns in the query will show up in the csv.

```
select orders.id, COUNT(order_items.id) as item_count
from orders, integrations.order_items
where order_items.order_id = orders.id
group by order_items.order_id
```

We should expect to see `item_count` depends on `prod.integrations.order_items.id` and `id` depends on `prod.integrations.orders.id`

```
select order_id from integrations.order_items
union all
select order_id from platform.order_items
```

We should expect to see `order_id` depends on `prod.integrations.order_items.order_id` and `prod.platform.order_items.order_id`. Of course, you should handle any number of unions.

```
select sum(count * price) as total_price
from orders, platform.order_items
where order_items.order_id = orders.id
```

We should expect to see `total_price` depends on `prod.platform.order_items.count` and `prod.integrations.orders.price`

```
select sum(order_items.count * price) as total_price
from orders, (
  select order_id as my_order_id, count from integrations.order_items
  union all
  select order_id as my_order_id, count from platform.order_items
) order_items
where order_items.my_order_id = orders.id
```

And of all this should be tracked through subqueries.

We should expect to see `total_price` depends on `prod.integrations.order_items.count`, `prod.platform.order_items.count` and `prod.integrations.orders.price`.

Part 2: Column Usage

For each pair, of select output column and each of it's upstream column, figure out if that upstream column's data must pass through an *opaque function* before it reaches the select output column.

An *opaque function* is one of `SUM` or `COUNT`

For example,

```
select order_id, sum(select id from orders) as sum_ids
from integrations.order_items
```

Here `order_id` depends on `prod.integrations.order_items.order_id` which doesn't pass through any *opaque functions.*

`sum_ids` depends on `prod.integrations.orders.id` which does pass through `sum`, and *opaque function.* There are no other ways for `prod.integrations.orders.id` to reach `sum_ids`, so we can tell that yes, must pass through an *opaque function*.

Feature Priorities

Not all features are weighted the same, here's the order of importance for each of the features for this challenge.

- simple select with identifiers and simple functions (like `COUNT`) only
- nested queries
    - handling column aliases
- handling expressions with multiple source columns (like `+` or functions)
- column usage
- unions