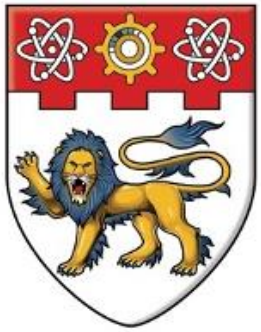


NANYANG TECHNOLOGICAL UNIVERSITY

CZ4031

Project 2 Report

Group Members:
Chua Peng Shaun (U1821442G)
Peng Pang Boon, Alex (U1820569L)
Tan Chuan Xin (U1821755D)
Teo Chen Ning (U1820456K)

Date of Submission:
30th November 2020

Choice of Programming Language:
Javascript (Frontend) on React
Python (Backend) on Flask

Source Code:

https://github.com/chenningg/sql-query-optimizer/

Contents

1. Introduction

This report consists of two parts, A and B. In part A, we generate a dummy set of data based on the TPC-H benchmark and run various SQL queries on the dataset. These queries are then fed into Picasso, which visualizes the plan choices made by the database management system (DBMS). In our case, this is PostgreSQL.

We will then explain the various diagrams and how the DBMS chooses its optimal plan as the data varies in selectivity (the number of rows accessed by the query of a certain relation).

In part B, we propose an algorithm written in Python that takes in an SQL query, returns alternative query plans, and then generates an appropriate natural language explanation on why a certain query plan was chosen by the DBMS.

It is our aim to make this natural language explanation as precise and accurate as possible, as well as work generically with various SQL queries.

1.1 The TPC-H dataset

The dataset we are working with is generated based on the TPC-H benchmark. A short summary of the relations found in the generated database is as follows:

TPC-H relation	Description	Cardinality (No. of rows approx)
REGION	Contains the continents that nations belong in.	5
NATION	Contains a list of all supported nations.	25
SUPPLIER	Contains details about the suppliers of parts.	10000
CUSTOMER	Contains a list of all customers, including their name, address and other information.	150000
PART	Contains parts sold and their respective suppliers and prices.	200000
PARTSUPP	Contains information from the supplier about the part, including inventory and price.	800000
ORDERS	Contains all customers' orders, the order status, as well as the price of the order.	1500000
LINEITEM	Contains all order items within each order, including the shipping dates and prices.	6001215

TPC-H mimics real world usage of a database, where joins can take extremely long if working on relations with more than 6 million rows. As such, a DBMS' query optimizer plays a large part in optimizing the query to minimize the inputs to joins.

1.2 Understanding Picasso

Picasso is a DBMS query optimization visualizer. By connecting to a database within PostgreSQL, it is able to leverage the DBMS' query optimizer to produce cost estimates on query plans and visualize them accordingly.

Picasso takes in an input known as a query template. The query template is essentially an SQL query with an important differentiating factor - a range of selectivities for some relation attributes.

The example below shows an SQL query and its corresponding query template:

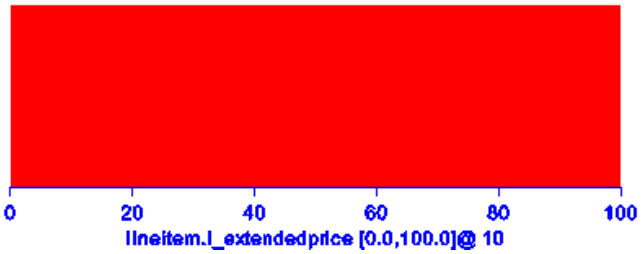
```
select
  sum(l_extendedprice * l_discount) as
revenue
from
  lineitem
```

```
  select
    sum(l_extendedprice * l_discount) as
revenue
  from
    lineitem
+ where
+   l_extendedprice :varies
```

The only difference is the line with :varies, which specifies a predicate that Picasso can detect. In essence, this line states that the query will be optimized in accordance with the variance in selectivity of *l_extendedprice*.

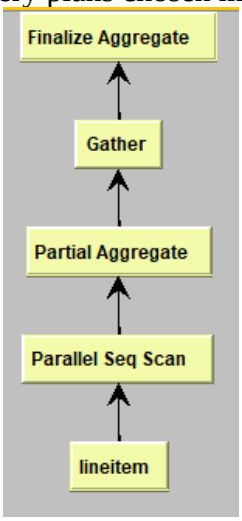
In simpler terms, Picasso leverages PostgreSQL's query optimizer to determine a set of optimal query plans. As more rows (higher selectivity) of *l_extendedprice* gets used in the query, Picasso visualizes which plans are chosen (determined to be most optimal) by PostgreSQL.

The following diagram shows the output of Picasso's Plan Diagram. The Plan Diagram is a pictorial representation of the execution plan choices made by the optimizer over a relational selectivity space. It does this by assigning a unique colour to each distinct optimal plan in the diagram. It is always portrayed as a 2d representation of the plans chosen by the DBMS as the selectivity of the chosen attributes vary, even when the query template has more than 2 dimensions.



Plan Diagram

In the above query, we see that over the range of 0% - 100% (i.e. we involve 0 rows of *l_extendedprice* to all rows of *l_extendedprice*), PostgreSQL determines that it should always use the same query plan because it is the most optimal. In other cases, the query plans chosen may change along the range of selectivity.



QEP Tree

Picasso is able to visualize the specific query plans chosen to enable better explanations on why the DBMS made such a choice. Here, we see that since this query happens only on a single relation, the query plan is relatively simple and thus does not change even as the selectivity varies.

Note that Picasso supports multiple dimensions (aka multiple predicates). Therefore, it can visualize how the DBMS chooses various plans as multiple attributes vary along their selectivities.

It is worth noting that when the plan diagram generates a large number of plans resulting in a complex plan diagram, it is possible to enforce plan cardinality reduction by swallowing some plans to form a smaller set of optimal plans without increasing the cost of any individual query by more than a user-specified threshold value λ%. This results in a reduced plan diagram which will have substantial reductions in plan space cardinalities, allowing for efficient analysis without significant tradeoff in cost.

Other than the plan diagram, which shows the optimal plan chosen by the DBMS as the selectivity varies along the range specified, Picasso also produces various other diagrams. In this report, we will only cover two diagrams other than the plan diagram - the cost diagram and cardinality diagram.

The cost diagram is a visualization of the associated estimated plan execution cost as the selectivity varies. It is similar to a Plan Diagram, except for the addition of a third dimension (compiled cost), forming a 3d representation which shows the estimated cost of executing the query templates over the selectivity space. The estimated costs are normalized to the range [0,1] with respect to maximum cost.

The cardinality diagram is largely similar to the cost diagram but instead is a visualization of the estimated query result cardinality (number of rows returned) over the selectivity space.

2. Part A - Analysing queries

In part A, we will introduce a few SQL queries (transformed into Picasso's query templates). These queries are specifically chosen to show various key factors in how the DBMS' query optimizer chooses which optimal plans to execute. We will also explain each output provided by Picasso and why the DBMS chose that plan across the range of selectivities.

In all the below examples, we make several assumptions for the sake of consistency:

1. All the query templates have their predicates vary from the scale of 0% to 100%. This means that for each query, we examine the DBMS' query optimizer along the full range of possible rows accessed by the query for each attribute (predicate).
2. The selectivity range is sampled based on linear sampling. This means that across the range of 0% - 100%, we sample points with a fixed distance between any two points, instead of a skewed sampling (which can be better representative of the dataset in some cases).
3. All queries and optimizations carried out in this experiment runs on PostgreSQL 13.1.

The query templates used are the default queries for PostgreSQL bundled together with the Picasso software, since they have already been tuned for the TPC-H dataset. We will pick out a few key queries to demonstrate our observations.

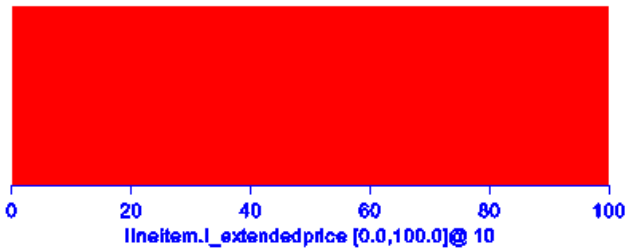
2.1 Query 1

The first case we shall explore has been examined above - that of a single predicate. This means that we only observe a single attribute of a relation as it varies in selectivity. Let us assume a more complicated query as shown below, where we are trying to get the line items in each order and their total prices after discounts and taxes.

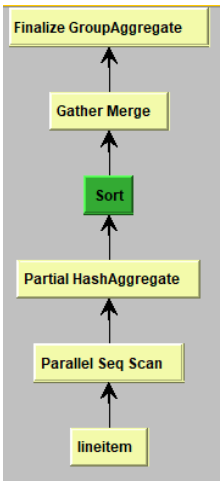
```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
where
    l_extendedprice :varies
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus
```

We vary on *l_extendedprice*.

2.1.1 1-dimensional plan diagram



Plan Diagram

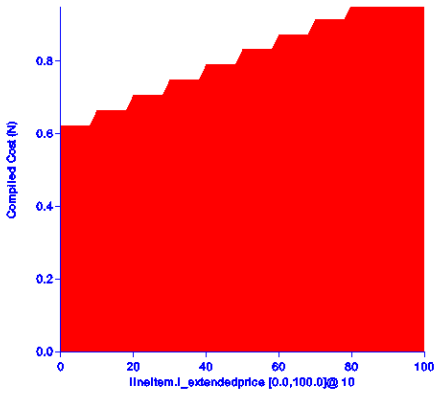


QEP tree

As we can see, with only 1-dimensional (only one attribute varying over the selectivity space) predicates, the query plan usually does not change. This is because as the query acts only on a single relation, the number of optimizations the DBMS can do is limited in nature, and hence it usually selects the same plan.

For example, if the query requires scanning all records in a relation, there are only so many ways to do so - in this case, the DBMS does parallel sequential scanning. Similarly, if the SQL query calls for an order by, which it does in this case, then we must perform a sort no matter what, making the final optimized query similar across the selectivity space.

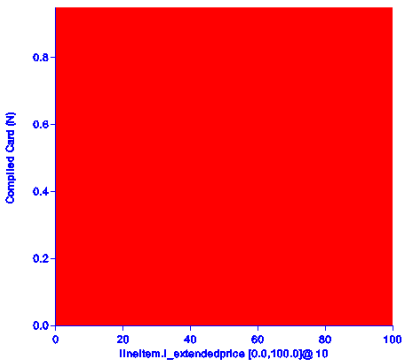
2.1.2 2-dimensional cost diagram



Cost Diagram

For a 1-dimensional predicate, the cost of executing the query can also be estimated to increase somewhat linearly as the selectivity constant goes up. This makes sense - the more rows queried in the database, the higher the cost, especially if a sequential scan is used.

2.1.3 1-dimensional cardinality diagram



The cardinality result remains constant across the entire selectivity range, showing that all available plans (in this case, only one) return a similar number of results. Running this query through a DBMS reveals that a maximum of 4 rows is returned at almost all selectivities, hence maximum cardinalities portrayed in this cardinality diagram. While it is intuitive that a selectivity of 0 should not return any rows, we take into consideration this observation that graphical representations of low selectivity might omit selectivities of 0.

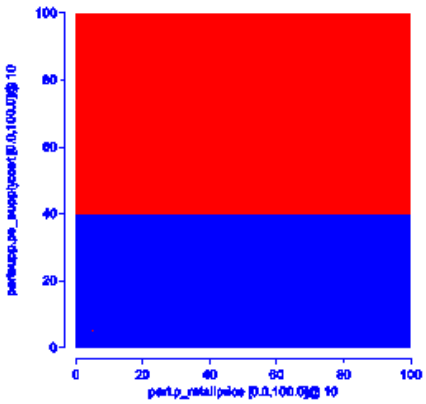
Cardinality Diagram

2.2 Query 2

```
select
    s_acctbal,
    s_name,
    n_name,
    p_partkey,
    p_mfgr,
    s_address,
    s_phone,
    s_comment
from
    part,
    supplier,
    partsupp,
    nation,
    region
where
    p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and p_retailprice :varies
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'EUROPE'
    and ps_supplycost <= (
        select
            min(ps_supplycost)
        from
            partsupp,
            supplier,
            nation,
            region
        where
            p_partkey = ps_partkey
            and s_suppkey = ps_suppkey
            and s_nationkey = n_nationkey
            and n_regionkey = r_regionkey
            and r_name = 'EUROPE'
            and ps_supplycost :varies
    )
order by
    s_acctbal desc,
    n_name,
    s_name,
    p_partkey
```

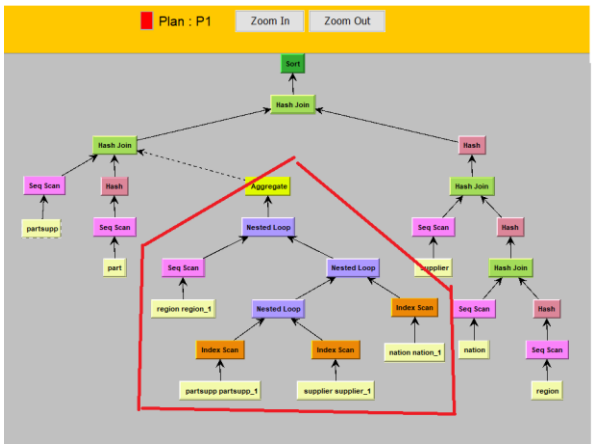
We vary on *ps_supplycost* and *p_retailprice*.

2.2.1 2-dimensional plan diagram

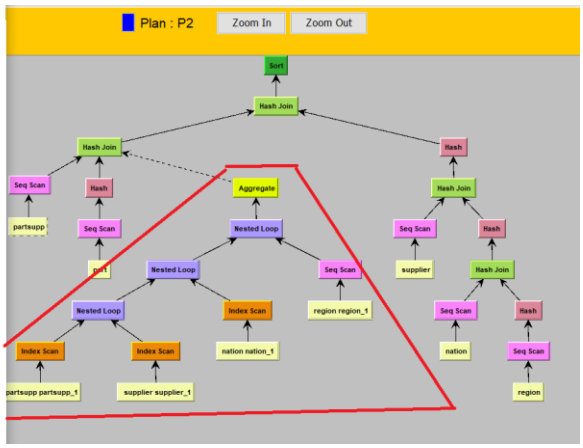


Plan Diagram

As seen in the diagram, a plan switch point exists at 0.4 selectivity of *ps_supplycost*, whereby a selectivity of more than 0.4 results in a switch to Plan 1 from Plan 2.



QEP Tree Plan 1

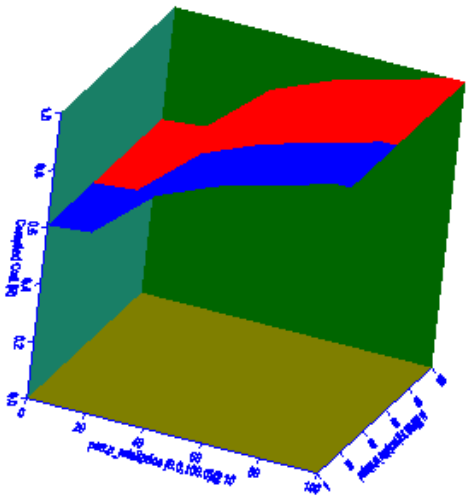


QEP Tree Plan 2

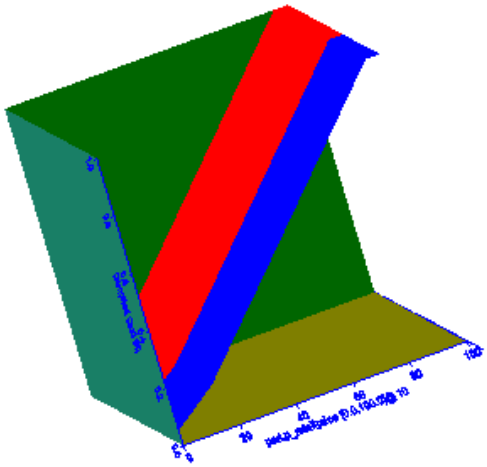
A closer look at the schematic plan tree diagrams of Plans 1 and 2 show that Plan 1 has a bushy tree whereas Plan 2 has a left deep tree. The key difference lies in the nested loop join sequence where *region* becomes the left child in Plan 1 as compared to Plan 2.

Plan 2 is selected when *ps_supplycost* is low in selectivity (<0.4), otherwise sequential scan on region is done first. This makes sense: if the region returns on average 40% of the tuples, and if the selection condition on *ps_supplycost* is strict enough such that it returns fewer tuples than *region* does, it should be used as the leftmost selection on a left-deep tree to get a smaller intermediate result.

2.2.2 3-dimensional cost diagram and cardinality diagram



Cost Diagram



Cardinality Diagram

As seen from the cost diagram, we observe that both plans have identical costs. Selectivity of *p_retailprice* is the main factor that influences cost. However, selectivity of *ps_supplycost* on the other hand only determines the possibility of a plan switch when its selectivity increases. Similarly, the cardinality diagram depicts an identical observation as cardinality increases linearly with the increasing selectivity of *p_retailprice* whereas selectivity of *ps_supplycost* determines the plan selected.

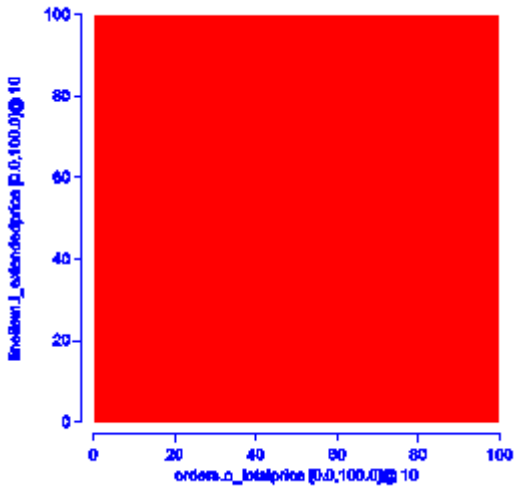
Given a relatively simple selection criteria, and varying the selectivities on simple attributes, we can see that these two suggested plans do not actually diverge in optimal costs.

2.3 Query 3

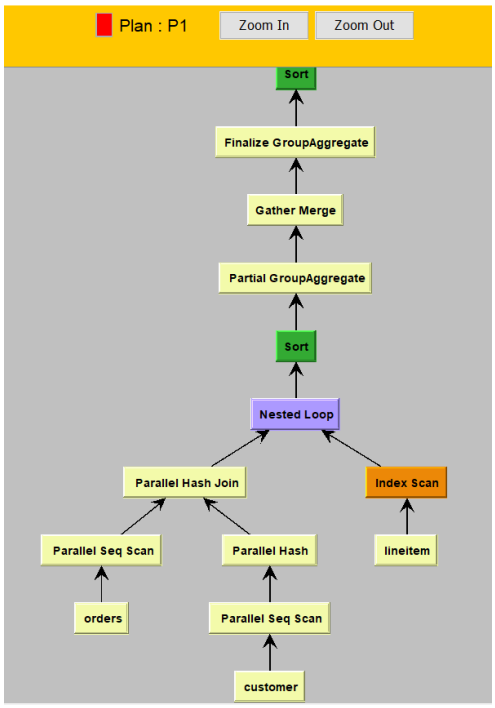
```
select
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    customer,
    orders,
    lineitem
where
    c_mktsegment = 'BUILDING'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_totalprice :varies
    and l_extendedprice :varies
group by
    l_orderkey,
    o_orderdate,
    o_shippriority
order by
    revenue desc,
    o_orderdate
```

We vary on *o_totalprice* and *l_extendedprice.*

2.3.1 2-dimensional plan diagram



Plan Diagram

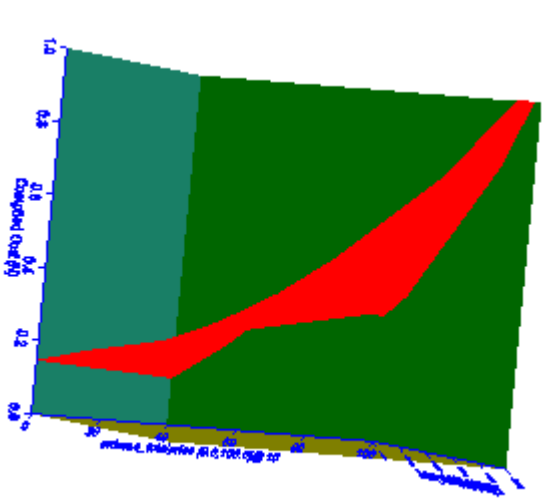


QEP Tree

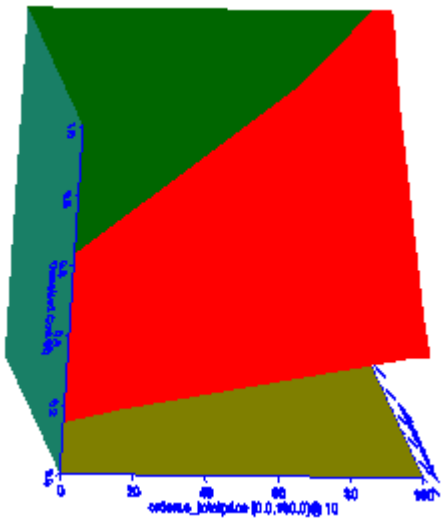
From the plan diagram, we observe that both predicate selectivities only result in one suggested plan being returned with orders on the left, lineitem on the right. This suggests that querying on *o_totalprice* first will

always be more selective than querying on *l_extendedprice*, therefore there is only one suggested plan presented.

2.3.2 3-dimensional cost diagram



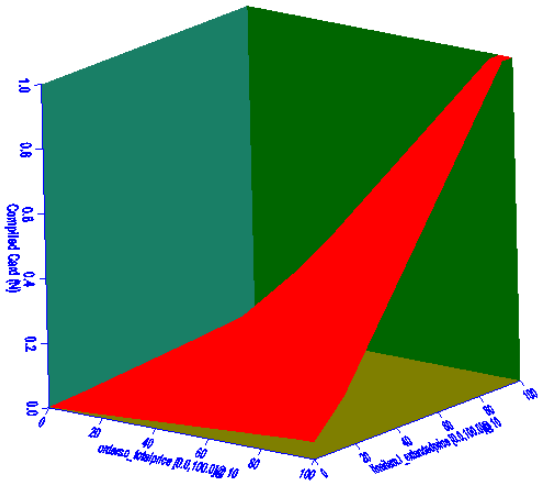
Cost Diagram



Cost Diagram

As seen from the cost diagram, the curved surface indicates that having high selectivities on both *o_totalprice* and *l_extendedprice* results in the highest cost possible. This is logical since it results in the greatest number of tuples selected. Also, we observe a non-linear relationship from the curved surface, which indicates that *l_extendedprice* and *o_totalprice* do not have proportional relationships as well.

2.3.3 3-dimensional cardinality diagram



Cardinality Diagram

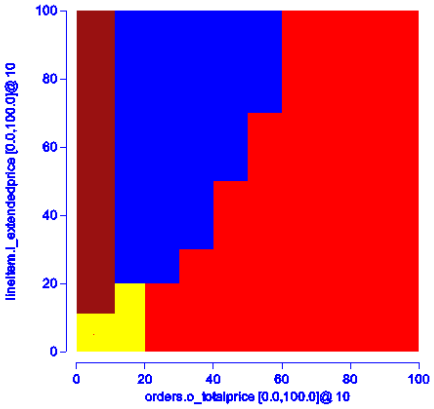
The cardinality diagram shows that selectivities of both *l_extendedprice* and *o_totalprice* is proportional to the number of tuples returned for the query. The cardinality is dependent on both predicate selectivities and returns a small number of tuples if only one predicate has high selectivity while the other is low.

2.4 Query 4

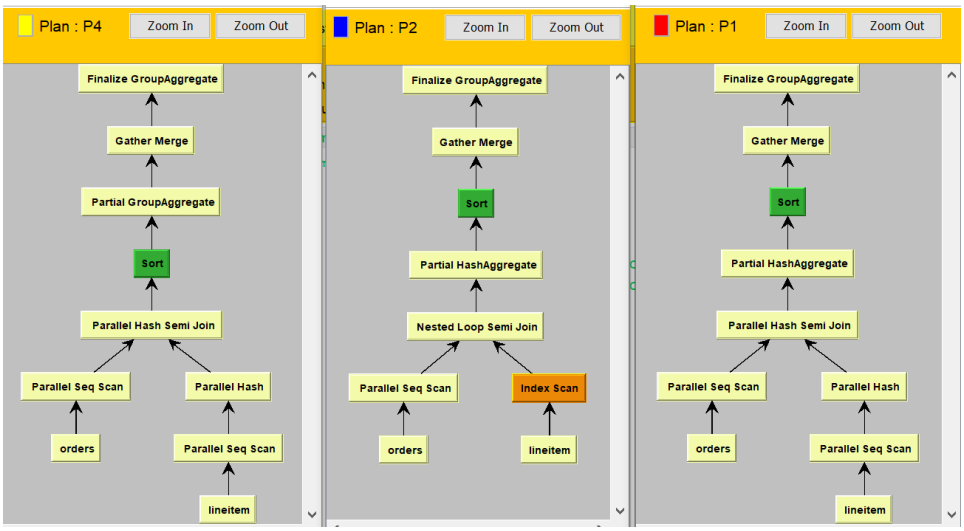
```
select
    o_orderpriority,
    count(*) as order_count
from
    orders
where
    o_totalprice :varies
    and exists (
        select
            *
        from
            lineitem
        where
            l_orderkey = o_orderkey
            and l_extendedprice :varies
    )
group by
    o_orderpriority
order by
    o_orderpriority
```

We vary on *o_totalprice* and *l_extendedprice.*

2.4.1 2-dimensional plan diagram



Plan Diagram



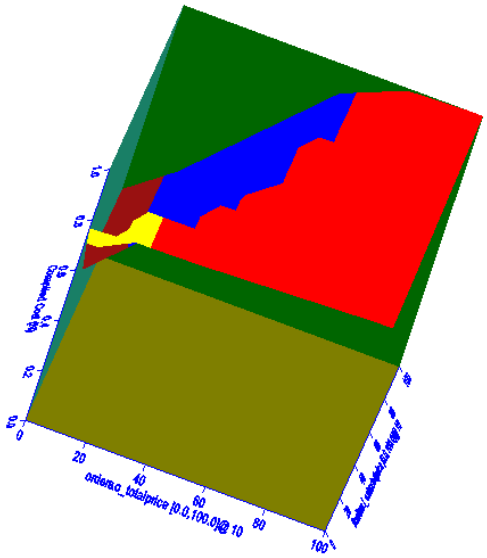
QEP Trees for P1, P2, P4

As seen from the plan diagram and schematic plan-tree diagrams, Plans 1, 2 and 4 meet at a point and hence at that point with particular selectivities of *o_totalprice* and *l_extendedprice*, we have an equivalent choice of 3 different plans. Plan 4 occurs at low selectivities of *o_totalprice* and *l_extendedprice.* Upon increasing

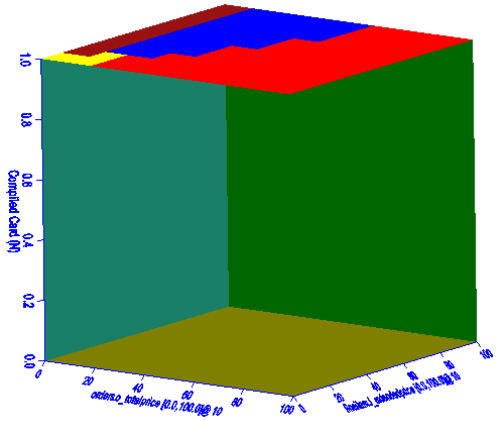
selectivity of *o_totalprice*, a partial HashAggregate is introduced after the partial hash semi join. This makes sense as with small groups, a partial aggregate cannot reduce the number of rows significantly and merely adds overhead to the query. However, with increased cardinality, it is easy to parallelize the aggregation by hashing on the group by key *o_orderpriority* by distributing different groups to different threads.

Upon increasing selectivity of *l_extendedprice*, there are 2 changes, *lineitem* is read with an index scan rather than a sequential scan due to the significant increase in cardinality as *lineitem* has the largest number of rows among all relations. Instead of a parallel hash semi join, a nested loop semi join is used instead due to the introduction of an index in *lineitem*, which improves performance of nested loop joins significantly.

2.4.2 3-dimensional cost and cardinality diagram



Cost Diagram



Cardinality Diagram

As seen from the cost diagram, cost is lowest when selectivity for *o_totalprice* is low while selectivity of *l_extendedprice* is high, possibly due to cost savings from the index scan. As selectivity for *o_totalprice* increases, the cost is seen to increase linearly as well before plateauing at approximately 0.7 selectivity. However, an interesting observation depicts high levels of cost when both predicate sensitivities are low, likely due to the sequential scanning of *lineitem* which is the relation with the highest cardinality.

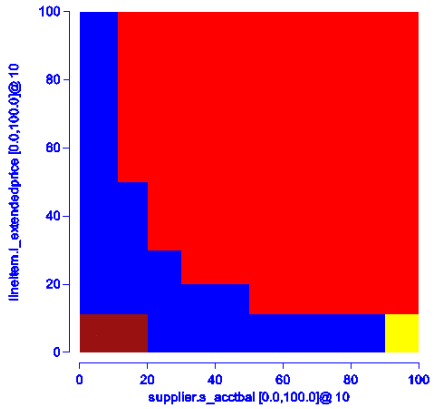
The cardinality diagram on the other hand always depicts that all plans selected return all rows of the relations. That is to be expected as *o_orderpriority* has only 5 unique values and at all possible selectivities would return all 5 unique values.

2.5 Query 8

```
select
    o_year,
    sum(case
        when nation = 'BRAZIL' then volume
        else 0
    end) / sum(volume) as mkt_share
from
    (
        select
            DATE_PART('YEAR',o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) as volume,
            n2.n_name as nation
        from
            part,
            supplier,
            lineitem,
            orders,
            customer,
            nation n1,
            nation n2,
            region
        where
            p_partkey = l_partkey
            and s_suppkey = l_suppkey
            and l_orderkey = o_orderkey
            and o_custkey = c_custkey
            and c_nationkey = n1.n_nationkey
            and n1.n_regionkey = r_regionkey
            and r_name = 'AMERICA'
            and s_nationkey = n2.n_nationkey
            and o_orderdate between '1995-01-01' and '1996-12-31'
            and p_type = 'ECONOMY ANODIZED STEEL'
            and s_acctbal :varies
            and l_extendedprice :varies
    ) as all_nations
group by
    o_year
order by
    o_year
```

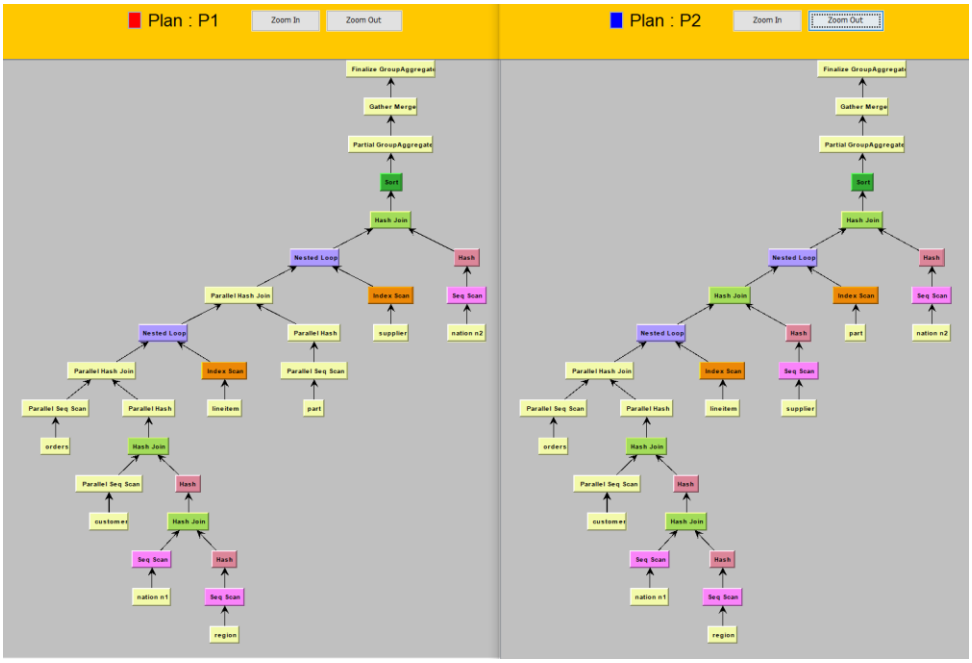
We vary on *s_acctbal* and *l_extendedprice.*

2.5.1 2-dimensional plan diagram



Plan Diagram

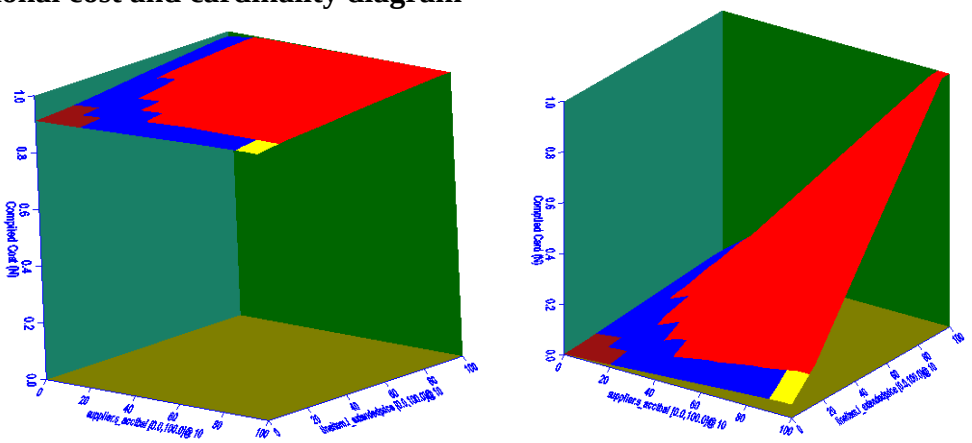
As observed from the plan diagram, different query plans are chosen by the DBMS for optimization as the selectivity varies. We pick the two most frequently chosen plans, Plan 1 (red) and Plan 2 (blue) for observation.



QEP Trees (P1, P2)

In this case, both query trees are left deep, but this may not be true for all queries. The difference between these two plans is that plan 1 utilizes a parallel hash join in the middle of the query plan, while plan 2 utilizes a standard hash join. We observe that plan 2 is only chosen when the selectivities for either attribute is low. This could be because with a low selectivity, a standard hash join running on a single process is sufficient for the query to be optimal enough. This is compared to plan 1, where due to a high selectivity, the hash join needs to occur in parallel on memory to speed up the process. For a query with low selectivity, the overhead of managing multiple threads may be less efficient than a non-parallel hash join.

2.5.2 3-dimensional cost and cardinality diagram



Cost Diagram **Cardinality Diagram**

As seen from the cost diagram, all plans already incur a high cost whereby increasing selectivities of both attributes lead to a slight increase in cost. The consistently high cost is probably a result of having to join almost all the tables to check for conditions.

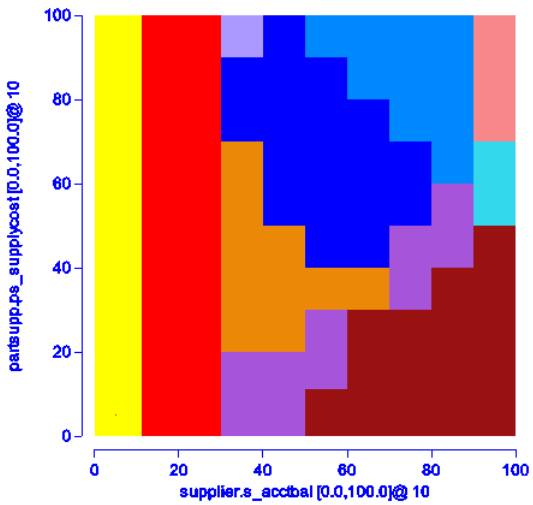
The cardinality diagram shows that the selectivity for both attributes result in a proportional increase in cardinality of the output as selectivities increase. Since the query does not restrict the output rows, the results returned increase with higher selectivity.

2.6 Query 9

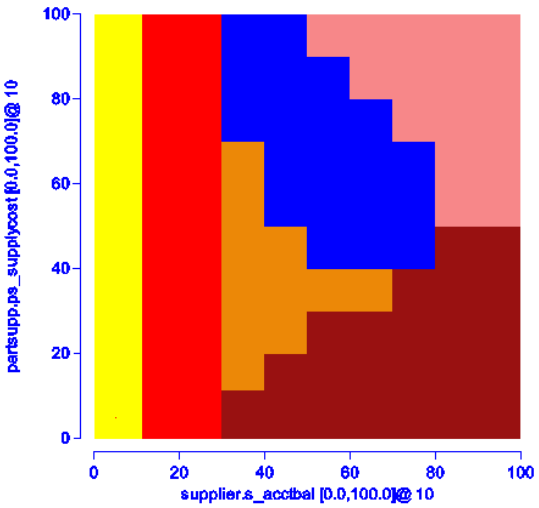
```sql
select
    n_name,
    o_year,
    sum(amount) as sum_profit
from
    (
        select
            n_name,
            DATE_PART('YEAR',o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
        from
            part,
            supplier,
            lineitem,
            partsupp,
            orders,
            nation
        where
            s_suppkey = l_suppkey
            and ps_suppkey = l_suppkey
            and ps_partkey = l_partkey
            and p_partkey = l_partkey
            and o_orderkey = l_orderkey
            and s_nationkey = n_nationkey
            and p_name like '%green%'
            and s_acctbal :varies
            and ps_supplycost :varies
    ) as profit
group by
    n_name,
    o_year
order by
    n_name,
    o_year desc
```

We vary on *s_acctbal* and *ps_supplycost.*

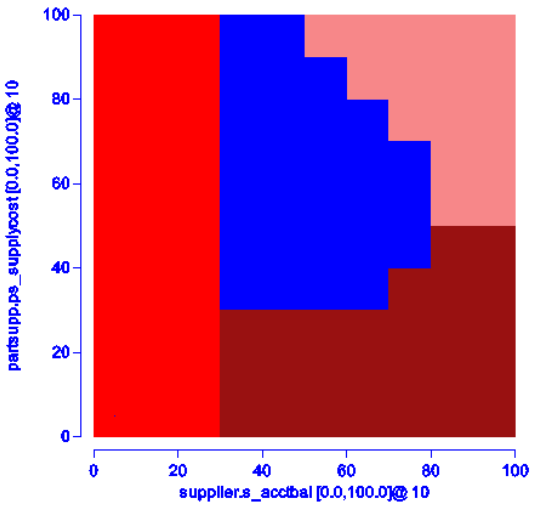
2.6.1 2-dimensional plan diagrams



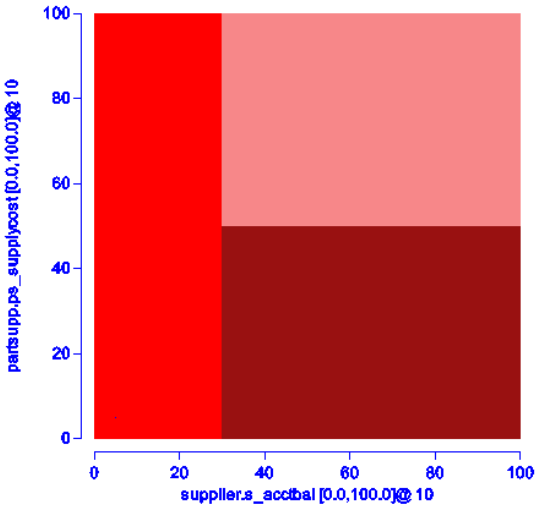
Plan Diagram



Plan Diagram 10% threshold

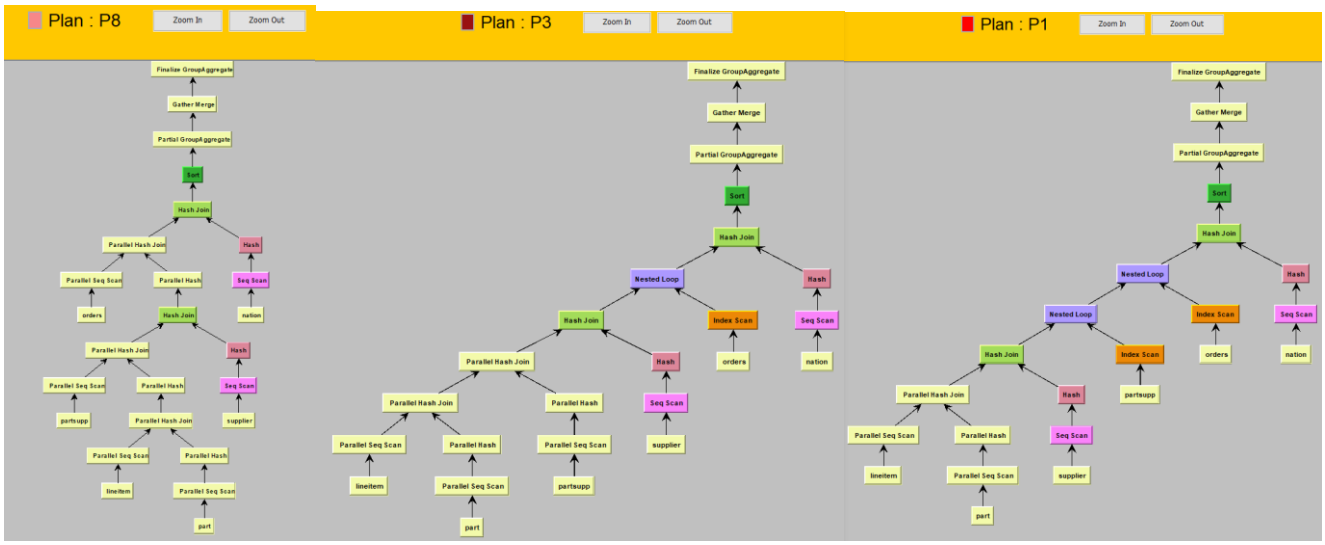


Plan Diagram 20% threshold



Plan Diagram 30% threshold

As seen from query 9, the initial plan diagram results in 10 plans being generated. Among the 10 plans, the plan in purple violates the basic tenet of Parametric Query Optimization - plan uniqueness: An optimal plan appears at only a single contiguous region in the space. Subsequently, as we explore increasing the threshold for a reduced plan diagram, we see that the number of plans reduces to 6 at 10%, 4 at 20% and 3 at 30%. This highlights the existence of fine-grained choices that the DBMS makes which can be alleviated by swallowing some plans to form a smaller optimal set of query execution plans.



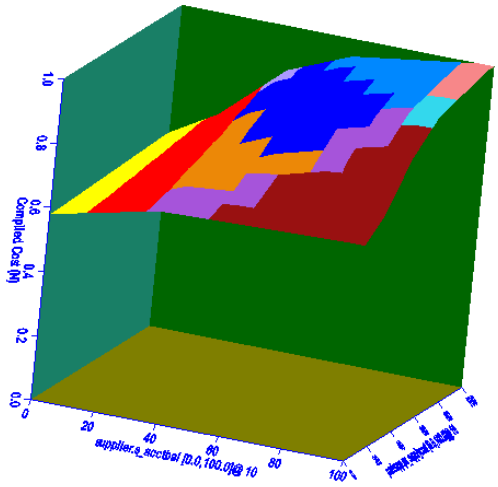
QEP Trees (P1, P3, P8)

As seen from the 3 most frequently chosen plans, we observe that Plan 1 and Plan 3 have left deep trees. However, as selectivities for both attributes increase greatly, this is replaced by a deep tree seen in Plan 8.

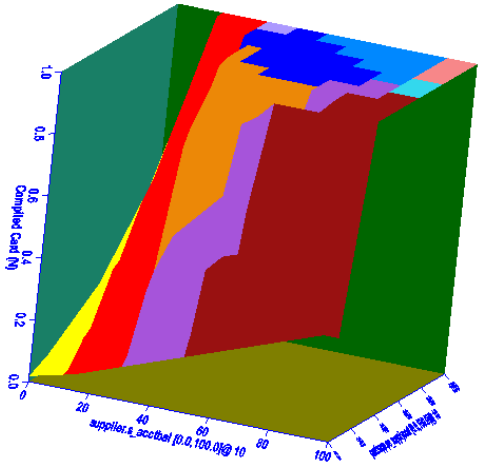
Comparing Plan 1 and Plan 3, upon increasing selectivity of *s_acctbal, partsupp* is accessed earlier on the left of the tree and is sequentially searched instead of using index scanning. This makes sense as Plan 1 includes high selectivity of *ps_supplycost* which hence utilises index scanning. Furthermore, as selectivity of *s_acctbal* becomes greater than that of *ps_supplycost, partsupp* returns a lower number of tuples and should be used on the left of the sequence.

As selectivities for both attributes increase, *orders* which was further on the right of the tree has instead been used on the left as it now returns less tuples.

2.6.2 3-dimensional cost and cardinality diagram



Cost Diagram



Cardinality Diagram

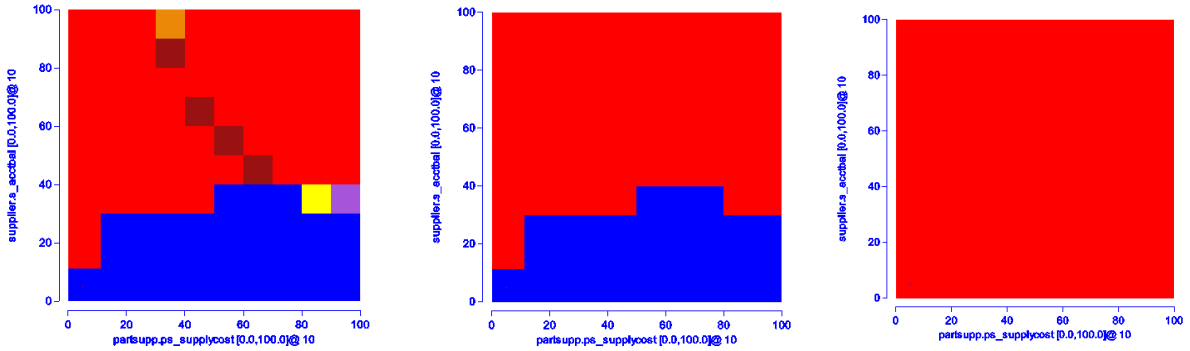
As seen from the cost diagram, increasing selectivities of both attributes result in a non-linear increase in the cost. Cost increases the most at low selectivities for both predicates before increasing only slightly at a higher selectivity. This trend is also observed in the cardinality diagram where maximum cardinality is achieved at low levels of selectivities. This can be attributed to the selected attributes *n_name* and *o_year* which return a very low number of cardinalities hence maximum cardinality is easily achieved.

2.7 Query 11

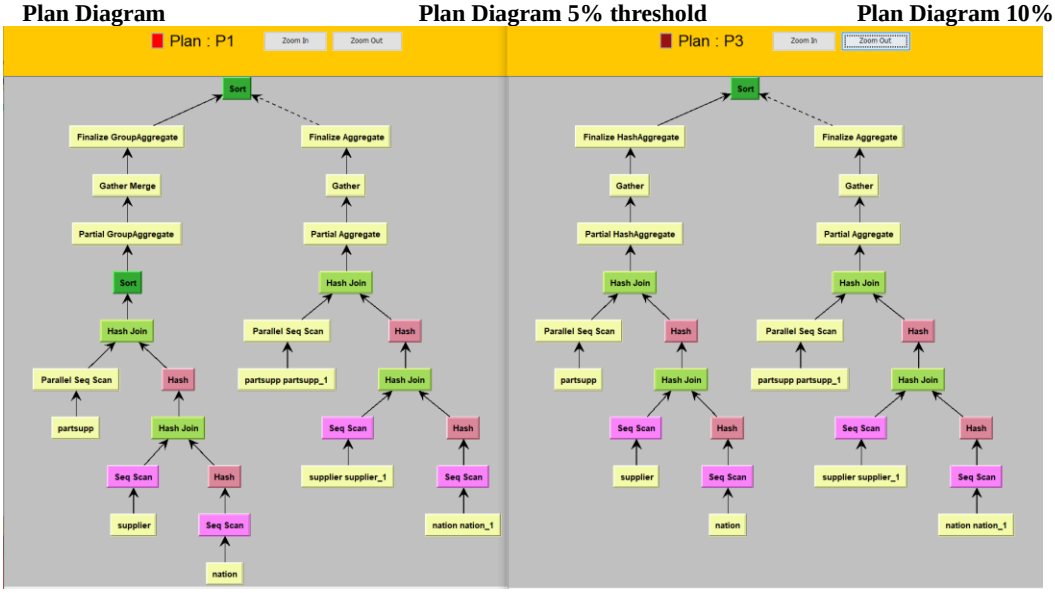
```
select
    ps_partkey,
    sum(ps_supplycost * ps_availqty) as value
from
    partsupp,
    supplier,
    nation
where
    ps_suppkey = s_suppkey
    and s_nationkey = n_nationkey
    and n_name = 'GERMANY'
    and ps_supplycost :varies
    and s_acctbal :varies
group by
    ps_partkey having
        sum(ps_supplycost * ps_availqty) > (
            select
                sum(ps_supplycost * ps_availqty) * 0.0001000000
            from
                partsupp,
                supplier,
                nation
            where
                ps_suppkey = s_suppkey
                and s_nationkey = n_nationkey
                and n_name = 'GERMANY'
        )
order by
    value desc
```

We vary on *s_acctbal* and *ps_supplycost.*

2.7.1 2-dimensional plan diagram



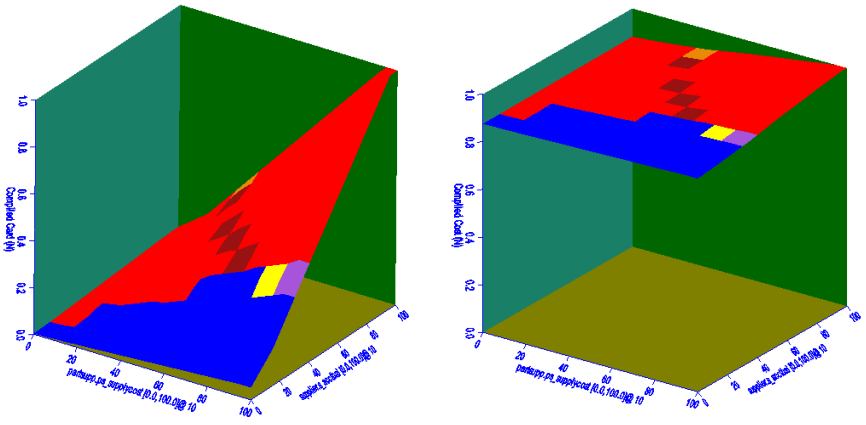
Plan Diagram **Plan Diagram 5% threshold** **Plan Diagram 10% threshold**



QEP Trees (P1, P3)

As seen in the plan diagram, an interesting pattern for Plan 3 is observed which appears discontiguously within Plan 1, violating the basic tenet of Parametric Query Optimization of plan uniqueness. Looking at the schematic plan-tree diagrams, we observe that this is due to sorting in Plan 1, which at particular selectivities of both attributes results in a lower cost than Plan 1 and hence the discontiguous pattern. It is interesting to note that the reduced plan diagrams appear radically different at very low thresholds of 5% and 10% where number of query plans decrease from 6 to 2 and then to 1, indicating that the plans seem to barely differ in cost. This thus further supports the need for a reduced plan diagram to combat the possibly suboptimal fine-grained decisions made by the DBMS.

2.7.2 3-dimensional cost and cardinality diagram



Cost Diagram **Cardinality Diagram**

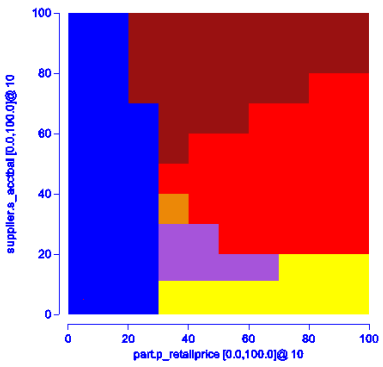
As seen from the cost diagram, selectivities of both attributes are almost directly proportional to the cost. Cardinality also observes the same trend but only increases slightly due to the already high cardinality.

2.8 Query 16

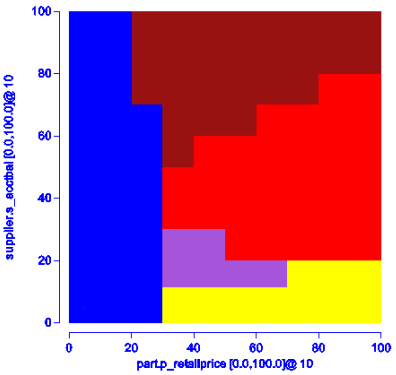
```
select
    p_brand,
    p_type,
    p_retailprice,
    count(distinct ps_suppkey) as supplier_cnt
from
    partsupp,
    part
where
    p_partkey = ps_partkey
    and p_retailprice :varies
    and ps_suppkey in (
        select
            s_suppkey
        from
            supplier
        where
            s_acctbal :varies
    )
group by
    p_brand,
    p_type,
    p_retailprice
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_retailprice
```

We vary on *s_acctbal* and *p_retailprice.*

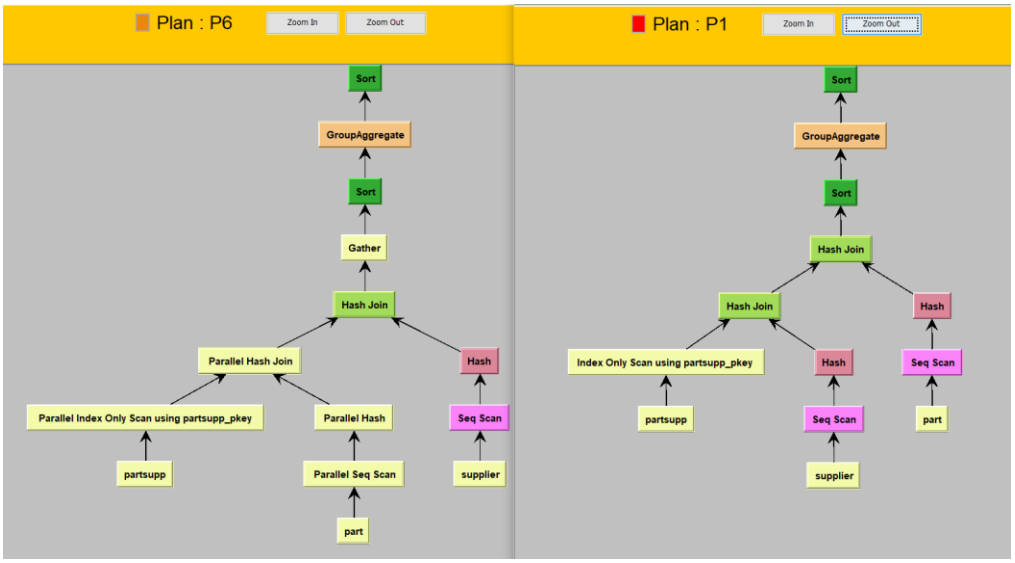
2.8.1 2-dimensional plan diagrams



Plan Diagram



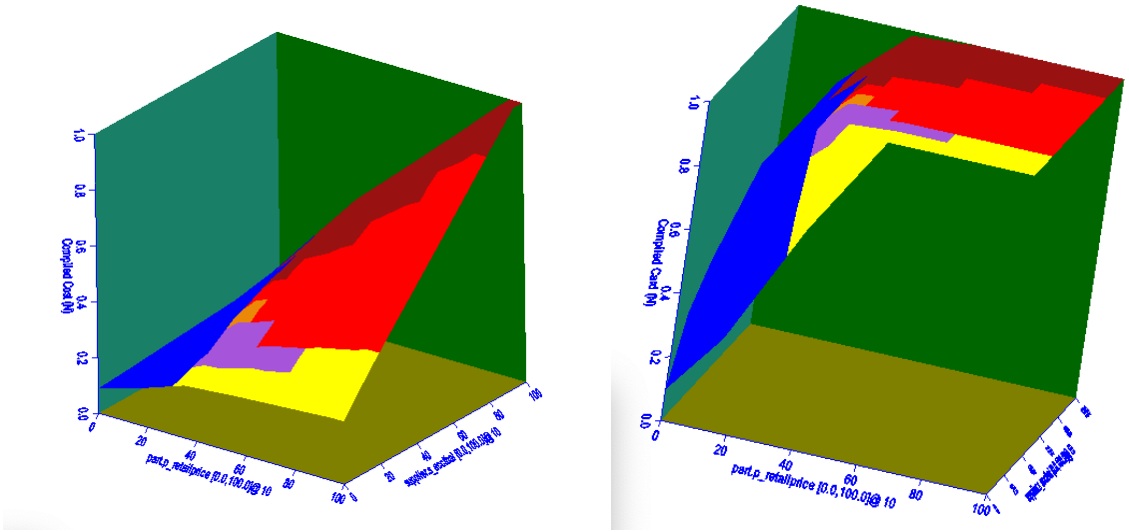
Plan Diagram 20/30% threshold



QEP Trees (P1, P3)

From the plan diagram, we observe that the reduced plan diagram at thresholds of 20-30% remain identical and in comparison, to the original plan diagram have swallowed Plan 6. The key difference between the plans is that Plan 6 accesses *part* on the left of the tree compared to Plan 1. This is logical as Plan 1 includes a greater range of selectivity of *p_retailprice*, naturally having a higher cost when accessing higher selectivity. Plan 6 also has a Gather function which is due to hash join being run in parallel. These two differences at approximately 0.3 selectivity for *p_retailprice* help save 20% of the cost compared to a higher selectivity for *p_retailprice.*

2.8.2 3-dimensional cost and cardinality diagram



Cost Diagram **Cardinality Diagram**

As seen from the cost diagram, selectivities of both attributes increase proportionally to the cost, with *p_retailprice* increasing cost to a greater extent at lower selectivity of approximately 0.2 and below. As for the cardinality diagram, cardinality is largely determined by selectivity of *p_retailprice.* The cardinality increases linearly with *p_retailprice* till approximately 0.5 selectivity where cardinality becomes fully maximised.

3. Part B - Query Execution Plan Explanation Algorithm

Referring to the project guidelines, we are expected to *"Design and implement an algorithm that takes as input a query, retrieves its query execution plan, and returns as output an explanation (i.e., reason) that describes the reason why the underlying RDBMS selected this plan among many others"*. This leads us to consider two considerations when selecting a QEP.

3.1 Alternative Query Execution Plans for Given Selectivity Space

Alternative QEPs for a given selectivity space are the set of different operator trees that can be generated for a given query. These alternative QEPS operate on the same selectivity space of the original query, but achieve the same output through a different method. Alternative QEPs are all equivalence preserving, but have different costs associated with them depending on the data access methods or data joining methods chosen, or even the sequence they are executed in.

A DBMS will generate numerous alternative QEPs for a given query and its selectivity range. Various heuristics will be applied to estimate the cost of these alternative QEPs, and the QEP with the cheapest estimated cost will be selected as the optimal QEP. Note that these are estimations, and actual execution costs might be different from estimations.

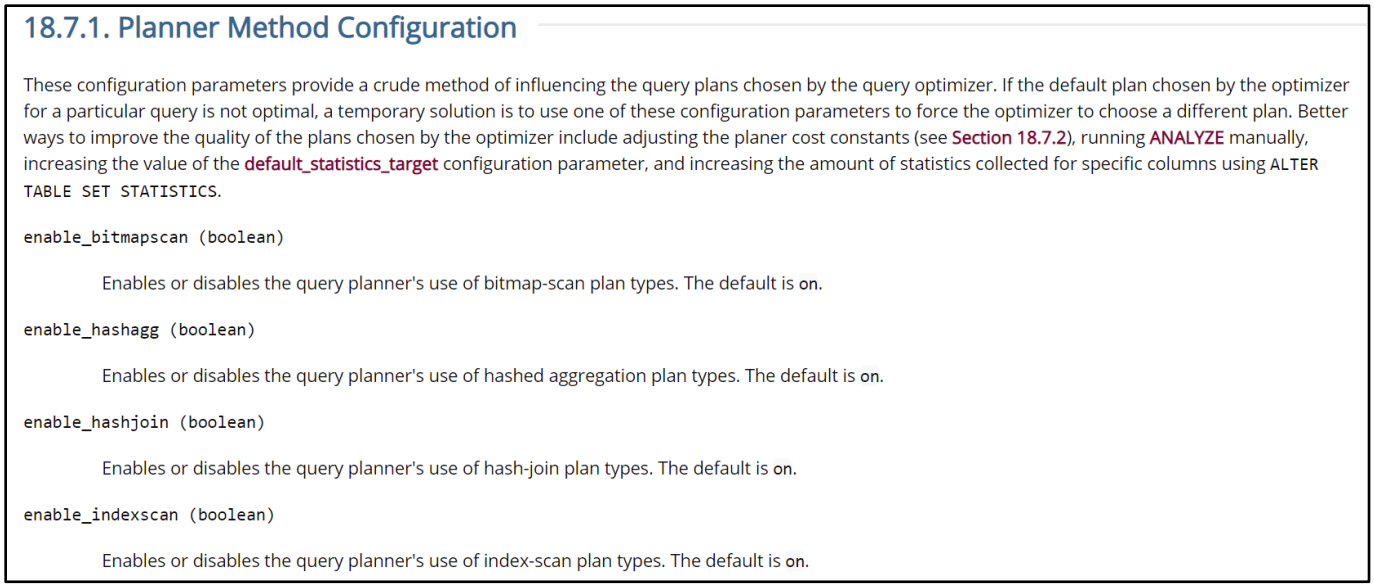
3.1.1 PostgreSQL and Alternative QEPs

While we have attempted to retrieve the alternative QEPs associated with a specific query that the PostgreSQL query optimizer considers in its planning, this has proven impossible. Alternative QEPs almost never reach completion as the Genetic Query Optimizer that PostgreSQL implements will kill off a lot of alternative trees before they can reach completion. Furthermore, alternative QEPs are neither logged nor stored by PostgreSQL. Consequently, out of the many potential plans that the PostgreSQL genetic query optimizer considers, we can only access what PostgreSQL believes to be the most optimal plan through the use of the EXPLAIN query. There are no off-the-shelf toolkits that will allow us to obtain the alternative plans considered by PostgreSQL as well.

3.1.2 Forcing the Planner

We have further attempted to force the query planner to only allow specific types of scans and joins in its plan consideration. This is achieved by disabling and enabling specific combinations of scans and joins to obtain different plan configurations. For example, we could disable hashjoin and block the query planner from considering it when crafting QEPs through the following command:

```
SET LOCAL enable_hashjoin OFF;
```



18.7.1. Planner Method Configuration

These configuration parameters provide a crude method of influencing the query plans chosen by the query optimizer. If the default plan chosen by the optimizer for a particular query is not optimal, a temporary solution is to use one of these configuration parameters to force the optimizer to choose a different plan. Better ways to improve the quality of the plans chosen by the optimizer include adjusting the planer cost constants (see Section 18.7.2), running ANALYZE manually, increasing the value of the default_statistics_target configuration parameter, and increasing the amount of statistics collected for specific columns using ALTER TABLE SET STATISTICS.

enable_bitmapscan (boolean)

Enables or disables the query planner's use of bitmap-scan plan types. The default is on.

enable_hashagg (boolean)

Enables or disables the query planner's use of hashed aggregation plan types. The default is on.

enable_hashjoin (boolean)

Enables or disables the query planner's use of hash-join plan types. The default is on.

enable_indexscan (boolean)

Enables or disables the query planner's use of index-scan plan types. The default is on.

However, this method is also unsatisfactory as the scan or join that is disabled cannot participate in the entire query. This fundamentally changes the QEPs that the query optimizer can consider, and we are left with an ambiguous alternative cost.

Consider the case where we wish to determine the best scanning method for a given relation R. PostgreSQL supports bitmapscan, indexscan, indexonlyscan, seqscan and tidscan. Suppose we were to investigate the costs of scanning R with these various scanning methods by only turning on one of the following scans for each query, and then performing these five queries. We might expect that this will allow us to capture the cost of scanning R using each of the five scanning methods available to us. However, this is certainly not the case. By disabling the other scanning modes for the **entire query,** every relation in the query will be affected, and this fundamentally affects the sequence of joins and scans that must be created in order to minimize cost. Therefore, the selectivity of relation R will differ across each of these five queries because R will likely be scanned in a different sequence, based on the complex interaction between the different relations and operators. This no longer results in a consistent basis of comparison, which is to calculate the different scan or join costs based on the same selectivity of the node.

3.1.2 Unable to Pursue

Given the two complications mentioned above, we are unable to pursue the route of investigating alternative QEPs for a given query. This will result in an incomplete analysis as to why the DBMS would have selected a particular QEP over others for the same selectivity space. Nonetheless, because PostgreSQL does not offer the necessary support to achieve this, we must abandon the concept of alternative QEPs for the given selectivity space.

3.2 Alternative QEPs for Neighbouring Selectivity Space

Inspired by Picasso, we are still able to exploit PostgreSQL functionalities to explore the alternative QEPs in the neighbouring selectivity space for a given query. This method addresses the fine-grained plan choices that are made by DBMSes. Modern query optimizers often make fine-grained plan choices. While this may be extremely effective, it may in fact lead to the generation of a query plan that may not have the lowest cost. This is evident from the diagrams generated by PICASSO, some of which exhibit a variety of intricate tessellated patterns, indicating the presence of strongly non-linear and discretized cost models. As earlier analysed, by producing reduced plan diagrams, we could possibly reduce considerable processing overheads associated with query optimization. Furthermore, plan reduction that swallows query plans associated with a small-sized plan by a larger plan may possibly significantly bring down the cardinality of the plan diagram as well without significantly affecting query cost.

The underlying principle is the fact that we could potentially have picked an alternative plan that was considered outside of our selectivity space, implement it, and obtain a lower cost even within our current selectivity space. However, the DBMS does not even consider these alternative plans that exist outside the selectivity of the given query.

A dummy example would be as such:

Selectivity	**<40%**	**40-45%**	**45-100%**
Lowest cost plan	Plan1, cost 100	Plan2, cost 10000	Plan3, cost 500

Our query happens to be for 40% selectivity, and Plan2 at a cost of 10000 is selected. However if we were to instead ignore the DBMS suggestion and forced it to utilise the neighbourhood plan Plan1, we might potentially achieve a cost that is 100 times less if we used Plan1 for the selectivity range of 40-45%, since the selectivity does not vary by much.

In other words, the DBMS' cost estimate might be inaccurate at a certain selectivity level, causing it to select a plan that is appreciably worse than a similar plan that it had chosen at a similar selectivity level.

Henceforth, our strategy will be similar to Picasso, whereby we vary the selectivity values of certain attributes/predicates to investigate the alternative plans that are in the neighbourhood, which might give us a significant cost reduction for executing our query. We can then return an output to explain why the optimal QEP returned was selected among other alternative plans.

3.2.1 Varying Selectivities for Alternative QEPs

Note that henceforth, "Alternative QEPs" refer exclusively to the QEPs in the neighbourhood selectivity space, not the alternative trees that are considered for our specific selectivity space.

3.2.1.1 Neighbourhood Selectivity Space

Picasso explores the selectivity space of 0% to 100%, in default of 10% intervals. For our purposes of finding a potentially better QEP in the neighbourhood of our current selectivity, we believe that this is not practical as it will consider plans that are very far away. This will then exceed our fine-grain decision boundaries, and very likely cross into plans that are unlikely and should not be considered.

Furthermore, we are creating a front-end application and responsiveness is important. If we have to vary four different attributes from 0% to 100% in 10% intervals, we will end up with 11^4 different queries, which causes significant overhead on the DBMS.

Therefore, we will only consider the selectivity space in the range of 20% above and 20% below the input selectivity. For example,

> Original selectivity 45% - consider 30%, 40%, 50%, 60%
> Original selectivity 5% - consider 0%, 10%, 20%

This will restrict our search space, allow us to avoid considering unrealistic plans, and find a potential plan to address the fine-grained QEP problem faster.

3.2.1.2 Attributes Allowed for Selectivity Variations

Referring to Picasso's source code, to vary the selectivity for a given predicate, the following conditions must hold (at least for the TPC-H database):

- Attribute must either be a number or a date

    - Characters and strings are not allowable
- Attribute must have a histogram stored in the database
    - There is no reasonable method to obtain selectivities otherwise
    - Most Common Value tables do not allow us to obtain range selectivities, only selectivities for specific values
- Attribute must participate in a range query
    - Picasso enforces this using **attribute :varies** syntax, thus omitting a specified value for the attribute whose selectivity space we want to vary
    - Equality estimation cannot be estimated with a histogram but rather MCVs and MCFs. Given the nature of our aim to vary selectivity of predicates, it does not make sense to incorporate equality relational operators, hence equality operators are not allowed to be varied
- Attribute must participate in a WHERE clause

We will implement similar constraints in our application and algorithm. Specifically, we have omitted those attributes that do not fulfill the first two conditions in the list above by performing a check on the database for the attributes that a histogram does not exist for. The query to perform this is

SELECT histogram_bounds FROM pg_stats WHERE tablename = xxx AND attname = xxx;

Therefore our remaining list of allowable predicates are as follows:

● c_custkey ● c_acctbal ● l_orderkey ● l_partkey ● l_suppkey ● l_extendedprice ● l_shipdate	● l_commitdate ● l_receiptdate ● n_nationkeyO_orderkey ● o_custkey ● o_totalprice ● o_orderdate	● p_partkey ● p_retailprice ● ps_partkey ● ps_suppkey ● ps_availqty ● ps_supplycost

As such, in our frontend GUI, we will only allow the above predicates to be selectable for variation by the user.

3.2.1.3 Obtaining Values for Selectivities

As mentioned, we will only vary selectivities in the range of ±20%. Therefore, we will utilise the histograms for the associated predicates to determine the original selectivity.

Using PostgreSQL's histogram retrieval, we can retrieve the histogram for the required predicates. The histogram divides the range into equal frequency buckets and by locating the bucket that contains the value of our input query, we would be able to calculate the selectivity of the input query's predicate.

We obtain the formula for selectivity estimation from PostgreSQL documentation, and have implemented the selectivity estimation for our current query as such.

```
# get the selectivity for the given attribute value
leftbound = 0
for i in range(num_buckets):
    if attribute_value > histogram[i]:
        leftbound = i

selectivity = (
    leftbound
    + (attribute_value - histogram[leftbound])
    / (histogram[leftbound + 1] - histogram[leftbound])
```

```
            ) / num_buckets

            # inverse logic for the >= and > operators, since prior logic assumes a <= or < operator
            if operator in ["<=", "<"]:
                pass
            elif operator in [">=", ">"]:
                selectivity = 1 - selectivity
```

After obtaining the selectivity of our current attribute, we will then utilise the histogram to search for the required attribute values that will provide us with the selectivity values that we will require. The required logic can be found in the *get_histogram()* function in *api/generate_predicate_varies_values.py*.

We repeat this for every predicate that has been selected by the user to obtain the permutations required to explore the neighbouring selectivity space for our given query.

3.3 Query Processing

3.3.1 Varying Selectivity

To illustrate the various implementations of our algorithm, we will be detailing the entire process using a sample query.

We first take in an input query to generate a query plan. These queries different from Picasso query templates. PICASSO requires that predicates to be varied do not supply an attribute value, and only supply a **': varies'** keyword to vary selectivity.

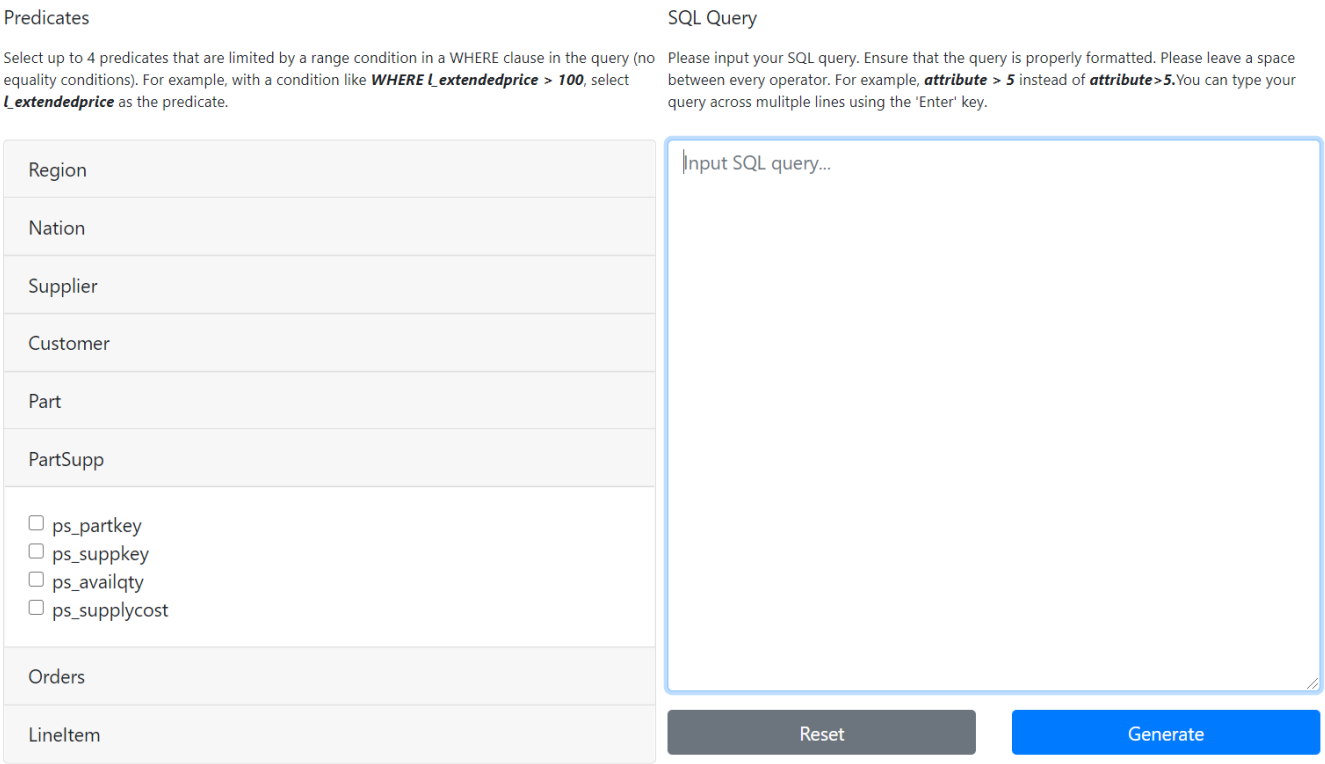
```
select
  c_custkey,
  c_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  c_acctbal,
  n_name,
  c_address,
  c_phone,
  c_comment
from
  customer,
  orders,
  lineitem,
  nation
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate >= '1993-10-01'
  and o_orderdate < '1994-01-01'
  and c_nationkey = n_nationkey
group by
  c_custkey,
  c_name,
  c_acctbal,
  c_phone,
  n_name,
  c_address,
  c_comment
order by
  revenue desc
```

Sample SQL Input Query

```
select
    c_custkey,
    c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
from
    customer,
    orders,
    lineitem,
    nation
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate :varies
    and o_orderdate :varies
    and c_nationkey = n_nationkey
group by
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
order by
    revenue desc
```

Equivalent PICASSO Query

As shown, our input query (left) allows for a specific query selectivity specified by the user, while the one for Picasso only allows for variation across a range of selectivity. This allows us to pinpoint selectivity ranges close to the original query of the user, giving the user more feedback as to how the DBMS makes fine grained decisions depending on estimated costs and cardinalities.



Our interface allows the user to key in their SQL query and select the predicates that they wish to vary. The rules regarding predicates that can be varied have been outlined in section **3.1.2.1**.

3.3.2 SQL Parsing

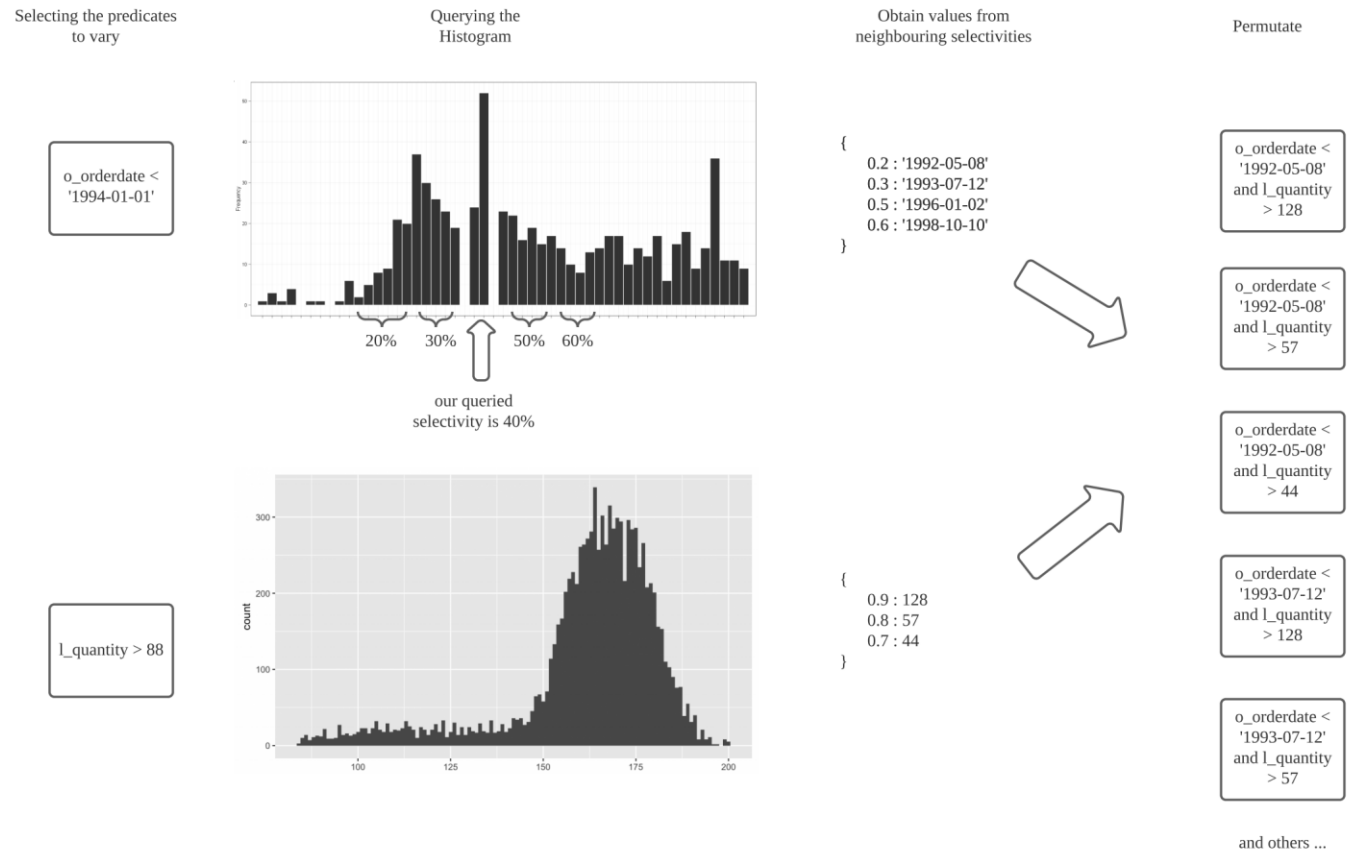
3.3.2.1 Cleaning the input query

Before we can process each selectivity, we'll need to parse the SQL input by the user into a suitable format for our library, *sqlparse*, to use.

Through a series of transformations, we ensure that the query is prettified and formatted (like adding spaces between operators), and that the *sqlparse* library is able to handle relatively low levels of nesting, the majority of SQL query syntax, and multiple conditions.

Once this is completed, the library aids us in finding the location of the predicates in each query. With this information, we can obtain each *key* and *value* pair, which will be passed on to the histogram class to use.

3.3.2.2 Acquiring Selectivity of Predicates from Input Query



After taking in a clean input query, we first identify the predicates whose selectivities are to be varied before identifying the datatype of the predicate and its equality comparator.

```
[1] datatype:  date
[1] {'relation': 'orders', 'attribute': 'o_orderdate', 'conditions': {('>=', "'1993-10-01'"): {'queried_selectivity': 0.7334615384
615384, 'histogram_bounds': {'0.4': datetime.date(1995, 12, 20), '0.30000000000000004': datetime.date(1996, 8, 8), '0.199999999999
99996': datetime.date(1997, 4, 3), '0.09999999999999998': datetime.date(1997, 12, 2)}}, ('<', "'1994-01-01'"): {'queried_selectivi
ty': 0.30291666666666667, 'histogram_bounds': {'0.2': datetime.date(1993, 4, 29), '0.4': datetime.date(1994, 8, 24), '0.3': dateti
me.date(1993, 12, 25), '0.5': datetime.date(1995, 4, 19)}}}}
```

```
selectivity of query:  0.7334615384615384
selectivities_required:  [0.6, 0.7, 0.8, 0.9]
```

Sample output for input query

As seen in the code snippet, we return the closest 2 selectivities in increments of 0.1 that are less than the selectivity of input query as well as the closest 2 selectivities in increments of 0.1 that are greater than the selectivity of input query. As seen in the sample input query selectivity of 0.733, the new selectivities returned are 0.6, 0.7, 0.8 and 0.9 which are the 2 closest selectivities lesser and more than 0.733.

3.4 Generating Permutations of QEPs with New Selectivities of Predicates

With the new selectivities for every predicate that has been selected by the user, we can now generate a series of permutations based on the *keys* and *new values.* These will be fed back into the original SQL query.

Each alternative query will then be executed to obtain alternative QEPs, graphs and explanations.

3.4.1 Using EXPLAIN

It is worth noting that we will be generating the query plan using EXPLAIN instead of EXPLAIN ANALYZE as the former generates the query plan by estimating the cost while the latter executes the query. Given our goal of retrieving a QEP with estimated cost before generating alternative QEPs to compare estimated cost and returning an optimal plan, EXPLAIN ANALYZE might generate too much overhead and take a long time to return all possible permutations of QEPs. Furthermore, the DBMS makes its decision based on estimations, not actual runtime costs since it is prohibitively expensive to perform an actual execution for every possible alternative QEP. Hence, EXPLAIN is more appropriate.

3.4.2 Query Output

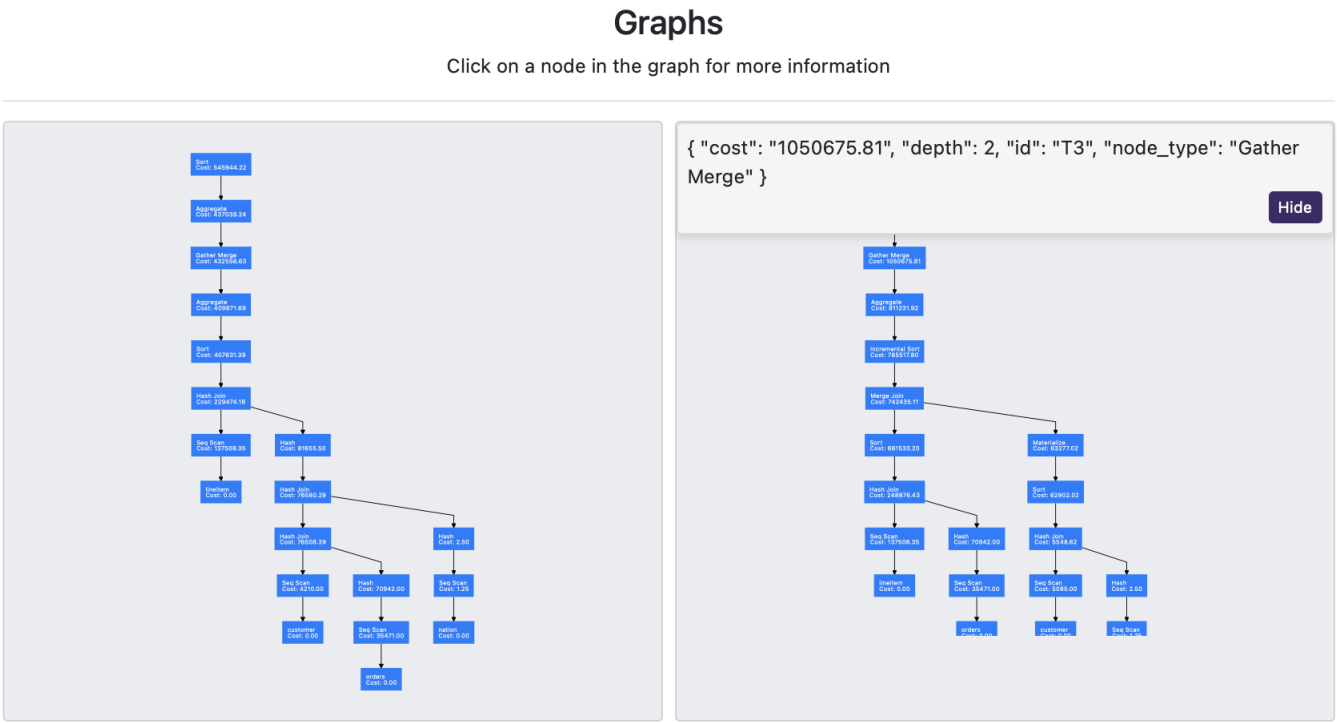
```
def execute_plan(qep_sql_string):
    try:
        # Get the optimal qep
        qep = query(qep_sql_string, explain=True)
        qep = json.dumps(ast.literal_eval(str(qep)))

        graph, explanation = visualize_explain_query(qep)

        qep = json.loads(qep)
        return qep, graph, explanation
    except CustomError as e:
        raise CustomError(str(e))
    except:
        raise CustomError(
            "Error in execute_plan() - Unable to get QEP, graph, explanation."
        )
```

Code Snippet to get QEP ,graph and explanation

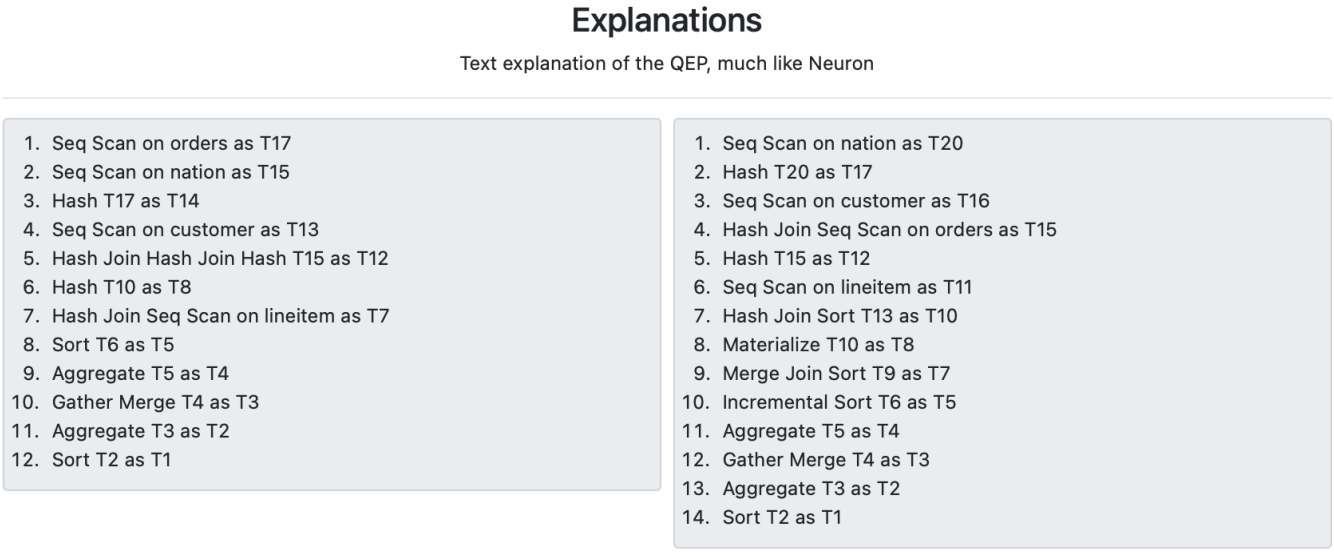
As seen in the code snippet, function *execute_plan()* takes in an input query in the form of an SQL string and returns a QEP, its graph to be visualized and a textual explanation.



Sample Visualization of Queries compared

QEPs returned are visualized using network graphs in sequence of node types returned by the QEP from the root all the way to the leaf nodes, traversed using breadth-first-search. Non-leaf nodes display types of operations such as Sort, Joins or Scans as well as its corresponding cost while leaf nodes display the name of the relation accessed. As seen in the figure, each node when clicked on has a unique id (e.g T3) which will be useful for understanding the explanation generated as well as other information such as the cost of performing the operation, the depth of the node from root as well as the node type.

The user is further able to select alternative plans from the top dropdown input to compare different plans against each other, and all the graphs and explanations will be dynamically changed.



Sample Explanations of Queries compared

Accompanying this query visualization graph is a text sequential explanation of how relations are accessed and subsequently scanned or joined to produce the output at the end. Nodes referred to will utilise the specific unique ID of the same node within the graphical visualization. This provides a more intuitive method to understanding the sequential execution of the various node actions within the QEP. This feature is very much like the query explanation feature in Neuron and will likely help with understanding the QEP better.

3.5 Selecting Optimal QEP

3.5.1 estimated_cost_per_row Metric

After generating new QEPs based on new selectivities returned, we will select an optimal QEP to return, using the metric of estimated cost per row, given by :

$$estimated\_cost\_per\_row = (startup\ cost + total\ cost) / rows\ returned$$

where startup cost and total cost are estimates given by EXPLAIN.

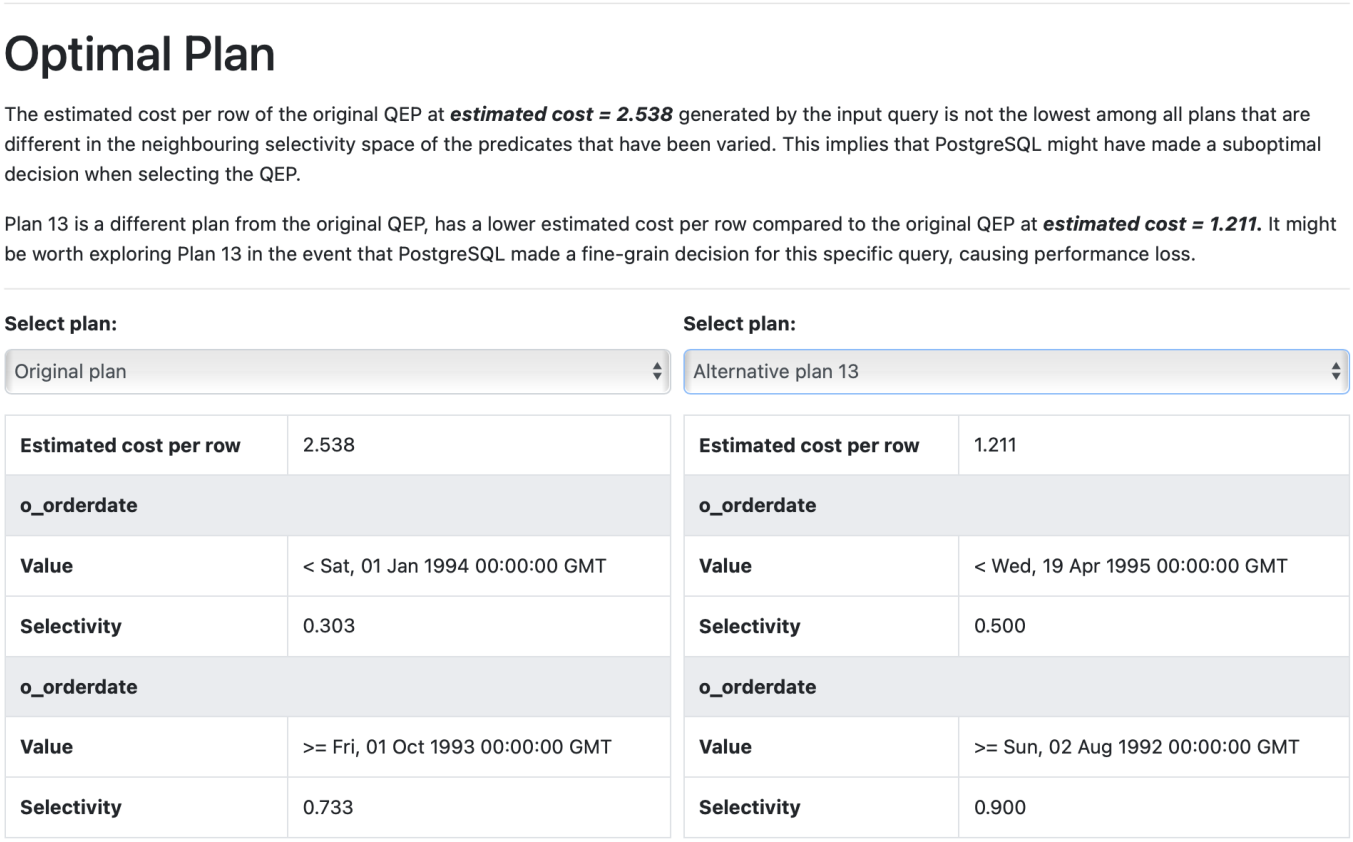
Startup cost is the estimate of the cost that will be incurred before the query will output its first tuple. Total cost is a misnomer - it actually refers to the estimate of the runtime cost before the query will complete and output every tuple required. Thus, we can imagine the sum of startup cost and total cost as the grand total cost of the entire query.

Given the variation of selectivity which in turn affects the cardinality of the QEP, we choose to use these metrics for an estimated cost as they are relative to the cardinality returned and hence serve as a better basis

of comparison rather than using total cost metric. An example to further support the rationale for selecting such metrics is this:

Assuming an input query with selectivity 0.5 has two alternative QEPs returned, one of which has 0.4 selectivity while the other 0.6 selectivity. The conventional query optimiser will pick the one with a lower total cost which in this case we assume to be that of 0.4 selectivity. However, as selectivity varies, cardinality varies and the QEP with 0.6 selectivity could have a much higher cardinality than that of the 0.4 selectivity QEP. Using our metrics, the 0.6 selectivity QEP could have a lower cost per row returned and be more effective in selection of optimal QEPs.

3.5.2 Interface



Plan comparison in Visual interface

As seen in the plan comparison, different plans can be selected for comparison. Information such as the estimated cost per row as well as the specific selectivities of predicates used for generation of QEP is also displayed. Based on the estimated cost per row, a recommendation is also made under the heading “Optimal Plan” to indicate a better alternative to the plan recommended by the RDBMS which may be suboptimal due to its fine-grained decision. We default to show the “best alternative” plan if it exists. We will show the original QEP that has been selected by the DBMS query planner on the left as comparison.

3.5.3 Potentially Better Alternatives

If an alternative plan possesses a lower *estimated_cost_per_row* as compared to the original plan, it will be highlighted under the “Optimal Plan” segment. However, it is important to note that even if an alternative plan has a lower *estimated_cost_per_row*, it does not automatically imply that it will be a better plan to implement for our original query, given that the selectivity space of the two queries are fundamentally different. It simply flags out an alternative QEP as a potentially cheaper plan.

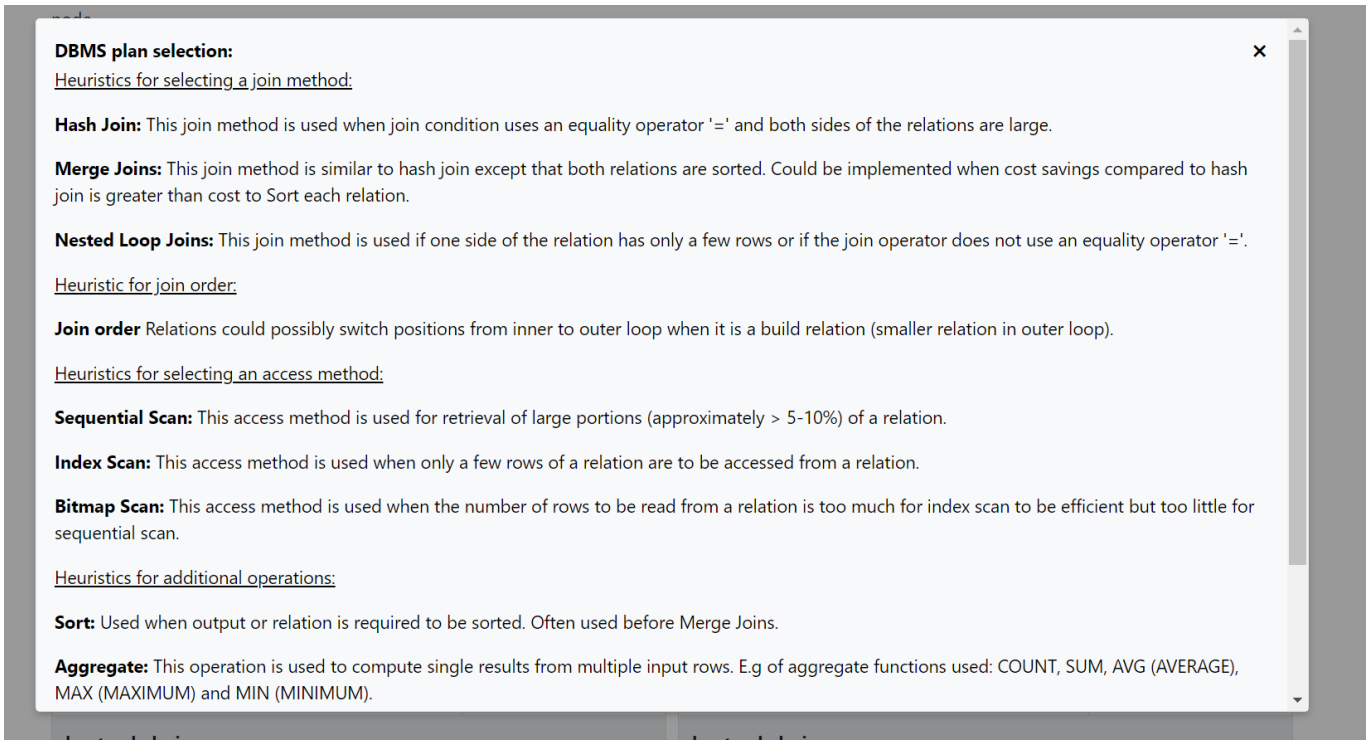
The cost estimate is but an estimate - actually executing our current query using an alternative QEP might yield significantly different results. However, QEPs with lower *estimated_cost_per_row* are more likely to be those that might prove the DBMS query optimizer has made a very fine-grained decision.

However, PostgreSQL does not grant us the ability to force the DBMS to execute a specific QEP over another. Therefore, there is no way we can test out our theory.

3.6 Heuristics of QEPs

Optimal QEPs are generated based on *estimated_cost_per_row* and may potentially return a significant number of different plans, some of which adopting a completely different sequence or combination of join and scan operations. Therefore, rather than generate an explanation based on differences between plans, we seek to provide a comprehensive list of heuristics providing users an explanation of why certain join or scan methods were chosen as well as an understanding of various operations that appear in the visualized query plan.

We provide a popup text in our application to help users have an overview of some of the heuristics associated with QEPs.



The heuristics compiled for explanation to users are as shown in the following subsections:

3.6.1 Heuristics for selecting a join method:

Hash Join: This join method is used when join condition uses an equality operator ‘=‘ and both sides of the relations are large

Merge Joins: This join method is similar to hash join except that both relations are sorted. Could be implemented when cost savings compared to hash join is greater than cost to Sort each relation.

Nested Loop Joins: This join method is used if one side of the relation has only a few rows or if the join operator does not use an equality operator ‘=‘.

3.6.2 Heuristic for join order

Join order → Relations could possibly switch positions from inner to outer loop when it is a build relation (smaller relation in outer loop)

3.6.3 Heuristics for selecting an access method:

Sequential Scan: This access method is used for retrieval of large portions (approximately > 5-10%) of a relation.

Index Scan: This access method is used when only a few rows of a relation are to be accessed from a relation.

Bitmap Scan: This access method is used when the number of rows to be read from a relation is too much for index scan to be efficient but too little for sequential scan.

3.6.4 Heuristics for additional operations

Sort: Used when output or relation is required to be sorted. Often used before Merge Joins

Aggregate: This operation is used to compute single results from multiple input rows. E.g of aggregate functions used: COUNT, SUM, AVG (AVERAGE), MAX (MAXIMUM) and MIN (MINIMUM).

Gather Merge: This operation indicates that the portion of the plan below it is run in parallel.

Hash: This operation is used to hash rows of a subtree on their join key values into a hash table before Hash Join.

Incremental Sort: This operation is used to accelerate sorting data when data sorted from earlier parts of a query are already sorted.

3.7 Test Queries

We have provided a set of queries that you can use to test our system. These queries can be found in the folder *sample_queries.*

3.8 Conclusion

While we have been unable to pursue the route of retrieving alternative QEPs of a given query, our algorithm of generating alternative QEPs based on selectivities obtained from the input query has proven that there are other query plans with lower estimated cost per row and that the DBMS might indeed make fine-grained decisions in its query optimization.

Although we are unable to determine if these alternative queries would actually be more optimal than the suggested optimal QEP from Postgresql for a specific selectivity, our application will still serve as a good way for users to understand query execution plans. It provides both a clear graphical representation of the QEP, as well as a textual explanation in the same page.

Furthermore, it provides comparisons against some other QEPs in the neighborhood, which can allow the user to think deeply about the differences between the plans, and whether the estimated cost difference between the two plans makes sense, giving insights into why the DBMS chose that particular optimal plan at that specific selectivity.