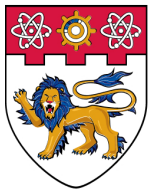


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4031: Database System Principles
AY2022/2023 SEMESTER 1

Project 2

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

Group 49

Group Members:

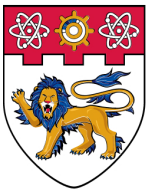
Brendan Ang Wei Jie (U2021332F)

Chia Si Yin Charlene (U2022730D)

Heng Fuwei Esmond (U1922412A)

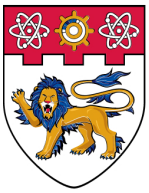
Pande Ashwin (U2023424D)

Zhao PeiZhu (U1923710B)



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

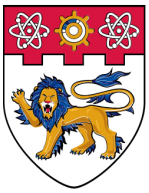
Table of Contents:



NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

1. Contribution of each member

Name of group member	Contribution
Brendan Ang Wei Jie	Preprocessing
Chia Si Yin Charlene	Annotation
Heng Fuwei Esmond	Interface
Pande Ashwin	Interface
Zhao PeiZhu	Annotation



NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

2. Algorithm Design

2.1 Plan Generator

The plan generator is responsible for generating possible Alternative Query Plans (AQP). The algorithm behind it exploits the planner method configuration feature of PostgreSQL to generate AQP based on a user input[1]. The configuration parameters (e.g., enable_hashjoin, enable_nestloop) in this feature provide a way to influence the query optimizer to choose a query plan with certain user-specified physical operators.

To place an emphasis on query components learnt in the CZ4003 module, the configuration parameters chosen are: "enable_bitmapscan", "enable_indexscan", "enable_indexonlyscan", "enable_seqscan", "enable_tidscan", "enable_mergejoin", "enable_nestloop" and "enable_hashjoin". This provides 8 togglable options, generating a total of $2^8 = 256$ possible AQPs. The “best” plan is assumed to be the plan chosen by the PostgreSQL planner with all options toggled on.

To quickly generate these AQPs, a bitmap is used to store the state of each flag, and is incremented on each iteration. Each option is set using the SET operator (e.g SET enable_bitmapscan = “off”). These flags are passed to the EXPLAIN operator to create a query plan. Additional options such as FORMAT JSON are used to create a parsable output for the rest of the application. An example query is as follows:

*EXPLAIN (FORMAT JSON) SELECT * FROM customer c JOIN orders o ON o.o_custkey=c.c_custkey WHERE c.c_custkey < 100 LIMIT 10;*

2.2 Annotation Module Algorithm Design

The annotation module exploits the fact that plan is returned via a xml (or json) like tree structure, and plans are always executed in left-right order among this tree, hence annotation traces this fact with a depth first order to annotate.

This algorithm first decides the type of a plan node. Types are classified as multiple classes: SCAN, JOIN, EMIT, and AUXILIARY. Then the algorithm decides further information to retrieve and what to annotate, based on the exact type of the node. Specific types among each class share a set of common information to retrieve. For example, all specific SCAN types should

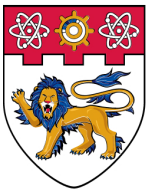
[1] https://www.postgresql.org/docs/current/runtime-config-query.html

have relation type, index type, and scan condition, and all JOIN types should have join filter, join schema, and estimated join size.

It should be noted that not all information needed to annotate a node is already inside that node, for instance, a JOIN node does not include what relations are involved in this join. Hence with depth-first left-right traverse order of a plan tree, information can be passed in a bottom-up manner. Some critical information passed in this manner includes: whether a child node has index; relations involved in a child node; emit mode of a child node, if possible; etc.

Based on this information, a node can then be annotated with a dynamic combination of a few hard coded statements for each type of node. The only exception is AUXILIARY type, which can be immediately annotated upon reaching that auxiliary node. Furthermore, as operations included by a DBMS have many types and part of these types are not critical to plan comprehension, hence these types are annotated as “not supported within this project scope”, such as a MEMOIZE operation.

There are a few constraints for this approach that heavily based on plan structure, though. The first is that this approach can work on a single plan well, but can do nothing to compare different plans. Based on exploitation on generated plans, we find that there is no guarantee that a pair of plans with similar costs can have either similar plan structure or set of selected operations. We argue that comparing any pair of plans may be difficult in nature, but our approach cannot handle it in theory. The second is that since our approach annotates the plan in left-right depth first order, any connection between a pair of nodes that none of them is a parent of another cannot be captured by our annotator. This can produce some issues though especially if there are multiple copies of a same relation involved in a query plan (multiple copies can share some data structure and scan / emit methods), we argue that in most cases a query executor also executes the plan in a same order, hence our annotator can capture reasons in majority cases.



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

3. Annotation

3.1 Explaining Join Methods

The different types of join methods we have learnt in this course are Hash Join, Sort-Merge Join and Nested Loop joins. Hash Join is used when both relations are relatively large and there is an equality operator involved. Sort-Merge joins are used when one or more relations are sorted and are similar to Hash Join. Nested Loop joins if one relation has a few rows and the equality operator is involved.
Since a single kind of join can involve multiple cases (for example, a nested loop join can have 0,1, or 2 operands that have index, and different looping and fetching strategies can be used), several kinds of information are needed to be maintained along tree structure. This includes: whether a node has an index; what relations are involved to produce this node; what scan strategies are used by an operand (sometimes, information about a join can be distributed to child nodes; for example, a heap fetch is usually the result of a bitmap scan, thus a following nested loop join must use index on relation.). Based on these information available, join nodes are dynamically annotated. If its child contains information directly related to it, then relevant children are also annotated with join node information.

3.2 Explaining Access Methods

The different types of access methods we have learnt in this course are Sequential scan, Index scan and Bitmap Scan. Sequential scan is used when a large portion of relation is retrieved and if no index is created on the tables. Index scan is used if indexes are available and only a few rows of the relation are retrieved. Bitmap scan is used if the portion retrieved is larger than a few rows but not so large than Sequential scan is used. Bitmap Heap Scan uses index bitmap generated by the condition and fetch corresponding pages stored in the heap. Bitmap Index Scan is used to generate a bitmap based on indexes of selected attributes. Another scan used in PostgreSQl is TID scan, which is selected if there is TID in the query predicate.
This is in theory, though. In some cases a scan node is in a whole with parent join node, but in this case annotating a scan node based on subsequent join node is the responsibility of join node. Hence, scan nodes can focus on annotating based on relation characteristics only.

3.3 Explaining Additional Operations

There are additional operations which may appear in the generated plan, two of which are Sort and Aggregate. Sort is used when output or relation is required to be sorted either due to an ORDER BY clause or because there is a following sort-merge join. Aggregate is used to compute single results from multiple input rows. Some of the aggregate functions include COUNT, SUM, AVG, MAX and MIN.

While these most common operations are the most cases, there are a few other auxiliary operations, such as LIMIT, UNIQUE, LOCKROWS, etc. These operations all have fixed purpose, hence can be directly annotated by fetching their functionality.

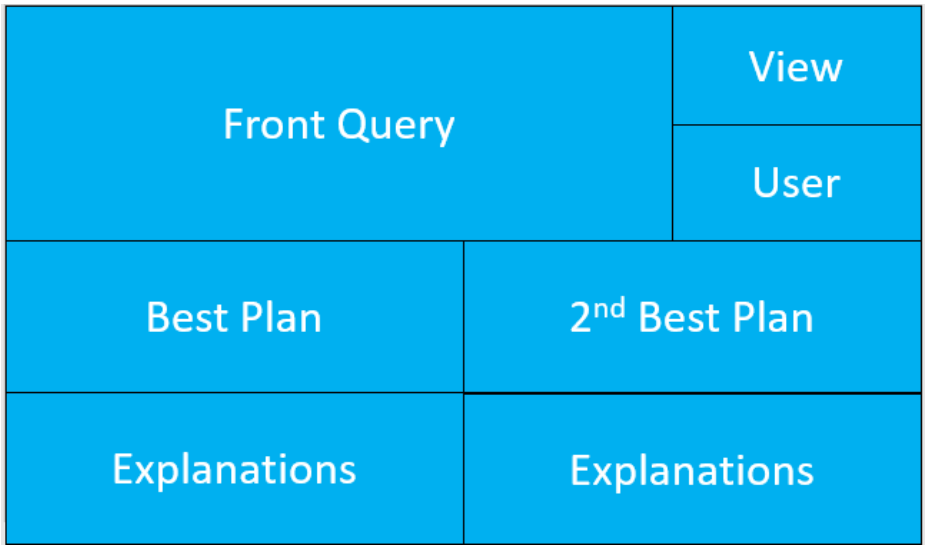
3.4 Example of Annotation using the SQL Query in 3.1

Based on the SQL Query in Appendix 7.1, the best plan and 1 alternate plan is chosen to be annotated. Here, the annotation is elaborated based on the nodes of the plan generated and the filters of child nodes. This output is then passed to the interface which will output the explanation according to the nodes. The output can be found in Appendix 7.2

4. GUI

4.1 Design

This following is a mockup of the GUI we implemented

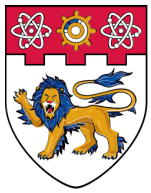


The first level displays the front query as well as view and user. which demonstrates the query as stated in 2.1.

The second level displays the 1st best plan and the 2nd best plan. which demonstrates the best plans as stated in 3.4

The third level displays the explanations for each of the best plan The explanations contain the reasons why the best plan is the best optimal plan for the query.

The GUI is designed using the Tkinter Python library.



NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

5. Software Limitations

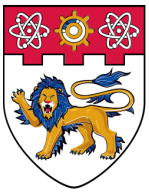
5.1 Plan Generator

The AQP generator has a set of well-defined limitations. Firstly, the module made use of only 8 toggle flags out of the possible 20 options available. This meant that there are possibly 2^{20} alternative plans (although unlikely) which were not explored. We felt that this was a necessary design choice in order to focus on the operators learnt in the classroom, while also striking a balance with the time complexity of the algorithm.

Additionally, the module executes only the EXPLAIN command on its own, without the use of ANALYZE. Together, the ANALYZE option causes the statement to be actually executed, not only planned. This will allow actual run time statistics to be added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned. This is useful for seeing whether the planner's estimates are close to reality. In this project, one of the annotations we wanted to display was for the “best plan” chosen by the PostgreSQL planner. Without the use of ANALYZE, our software ignored any real time costs in favor of theoretical costs. In this way, we can provide a more standard output but we recognize that the output may not be as accurate.

5.2 Annotation Module

The annotation module parses through the plan like a tree structure, and annotates each node based on the information provided in the node. Although we were able to annotate the best plan and alternate plan and recorded the output in the log file, we were not able to compare each pair of nodes for both plans and find the differences between each pair. This is because after many trials and errors of different sql queries and different alternative plans for each query, we found plans can have very different structures and that two nodes of different plans are unlikely to have similar structure or policies. Hence, we decided to output the annotation of the best plan and alternate plan side-by-side so as to manually view the comparison.



NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

6. Installation Guide

6.1 Installation Requirements

We used Python for this project and the version should be 3.9 or higher for the “project.py” file to run.

Install the relevant packages using `pip install "psycopg[binary]"` and “pip install sqlparse” and “pip install tk” (Tkinter library for frontend) if needed. Then run “project.py” and an interface should pop up.

Run project.py but typing `python project.py --conn “*your connection string*”`
e.g. `python project.py --conn "host=localhost dbname=tpch port=5432 user=postgres password=123`
The conn argument takes in your connection string to the tpch database.

6.2 Source Code

References:

https://personal.ntu.edu.sg/assourav/papers/VLDB-22-MOCHA.pdf

www.postgresql.org/docs/9.2/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS

7. Appendix

7.1 Example AQP Generated

```
DEBUG:root:debug cur plan:  {
   "Plan": {
      "Async Capable": false,
      "Group Key": [
         "s.s_suppkey"
      ],
      "Node Type": "Aggregate",
      "Parallel Aware": false,
      "Partial Mode": "Simple",
      "Plan Rows": 150,
      "Plan Width": 36,
      "Plans": [
         {
            "Async Capable": false,
            "Inner Unique": true,
            "Join Type": "Inner",
            "Node Type": "Nested Loop",
            "Parallel Aware": false,
            "Parent Relationship": "Outer",
            "Plan Rows": 170,
            "Plan Width": 24,
            "Plans": [
               {
                  "Async Capable": false,
                  "Inner Unique": false,
                  "Join Type": "Inner",
                  "Node Type": "Nested Loop",
                  "Parallel Aware": false,
                  "Parent Relationship": "Outer",
                  "Plan Rows": 170,
                  "Plan Width": 8,
                  "Plans": [
```

```
{
   "Alias": "s",
   "Async Capable": false,
   "Index Name": "supplier_pkey",
   "Node Type": "Index Only Scan",
   "Parallel Aware": false,
   "Parent Relationship": "Outer",
   "Plan Rows": 150,
   "Plan Width": 4,
   "Relation Name": "supplier",
   "Scan Direction": "Forward",
   "Startup Cost": 10000000000.15,
   "Total Cost": 10000000046.4
},
{
   "Alias": "ps",
   "Async Capable": false,
   "Node Type": "Bitmap Heap Scan",
   "Parallel Aware": false,
   "Parent Relationship": "Inner",
   "Plan Rows": 1,
   "Plan Width": 8,
   "Plans": [
      {
         "Async Capable": false,
         "Index Cond": "(ps_suppkey = s.s_suppkey)",
         "Index Name": "partsupp_pkey",
         "Node Type": "Bitmap Index Scan",
         "Parallel Aware": false,
         "Parent Relationship": "Outer",
         "Plan Rows": 1,
         "Plan Width": 0,
         "Startup Cost": 0.0,
         "Total Cost": 1.45
      }
   ],
   "Recheck Cond": "(ps_suppkey = s.s_suppkey)",
```

```
            "Relation Name": "partsupp",
            "Startup Cost": 1.45,
            "Total Cost": 5.46
          }
        ],
        "Startup Cost": 10000000001.59,
        "Total Cost": 10000000866.81
      },
      {
        "Alias": "p",
        "Async Capable": false,
        "Node Type": "Bitmap Heap Scan",
        "Parallel Aware": false,
        "Parent Relationship": "Inner",
        "Plan Rows": 1,
        "Plan Width": 24,
        "Plans": [
          {
            "Async Capable": false,
            "Index Cond": "(p_partkey = ps.ps_partkey)",
            "Index Name": "part_pkey",
            "Node Type": "Bitmap Index Scan",
            "Parallel Aware": false,
            "Parent Relationship": "Outer",
            "Plan Rows": 1,
            "Plan Width": 0,
            "Startup Cost": 0.0,
            "Total Cost": 0.18
          }
        ],
        "Recheck Cond": "(p_partkey = ps.ps_partkey)",
        "Relation Name": "part",
        "Startup Cost": 0.18,
        "Total Cost": 4.19
      }
    ],
    "Startup Cost": 10000000001.77,
```

```
            "Total Cost": 10000001574.94
          }
        ],
        "Startup Cost": 10000000001.77,
        "Strategy": "Sorted",
        "Total Cost": 10000001577.66
    }
}
```


7.2 Example Annotated AQP

```
DEBUG:root:Annotated best plan:  {
    "Plan": {
        "Async Capable": false,
        "Contain Relation": [
            "supplier",
            "partsupp",
            "part"
        ],
        "Explanation": "A general aggregation node without specifying aggregation method.",
        "Group Key": [
            "s.s_suppkey"
        ],
        "Node Type": "Aggregate",
        "Parallel Aware": false,
        "Partial Mode": "Simple",
        "Plan Rows": 150,
        "Plan Width": 36,
        "Plans": [
            {
                "Async Capable": false,
                "Contain Relation": [
                    "supplier",
                    "partsupp",
                    "part"
                ],
                "Explanation": "Node type Nested Loop: Only one operand of this index join has index scan on original relation, and the one without using index has size 170.Without knowing distinct values on join condition for the operand that uses index, since this size is small, heuristically index join performs better as only few index fetches are expected on relation with index. \n This
```

nested loop join has no filter, it can because either conditions are pushed down to scans or relations are small so a NL join is preferred.",
"Inner Unique": true,
"Join Type": "Inner",
"Node Type": "Nested Loop",
"Parallel Aware": false,
"Parent Relationship": "Outer",
"Plan Rows": 170,
"Plan Width": 24,
"Plans": [
{
"Async Capable": false,
"Contain Relation": [
"supplier",
"partsupp"
],
"Explanation": "Node type Nested Loop: This nested loop join has two operands that does index scan on original relation, hence this nested loop is a zig-zag join on both indices.\n This nested loop join has no filter, it can because either conditions are pushed down to scans or relations are small so a NL join is preferred.",
"Inner Unique": false,
"Join Type": "Inner",
"Node Type": "Nested Loop",
"Parallel Aware": false,
"Parent Relationship": "Outer",
"Plan Rows": 170,
"Plan Width": 8,
"Plans": [
{
"Alias": "s",
"Async Capable": false,
"Contain Relation": [
"supplier"
],
"Explanation": "Node type Index Only Scan: This is an Index Only Scan node.\nThere is no condition on supplier index scan. Without knowing predecessors of this node, this index scan is likely to be selectedas there is likely to be a join that has condition on this relation, or has build relation very small so a join fetching probe relation index is prefered. \nThis node that has index scan in it child nodes is used by its direct parent node as nested loop join.",
"Has Child With Index": true,

```
            "Index Name": "supplier_pkey",
            "Node Type": "Index Only Scan",
            "Parallel Aware": false,
            "Parent Relationship": "Outer",
            "Plan Rows": 150,
            "Plan Width": 4,
            "Relation Name": "supplier",
            "Scan Direction": "Forward",
            "Startup Cost": 10000000000.15,
            "Total Cost": 10000000046.4
        },
        {
            "Alias": "ps",
            "Async Capable": false,
            "Contain Relation": [
                "partsupp"
            ],
            "Explanation": "Node type Bitmap Heap Scan: A bitmap heap scan is always a direct parent of bitmap index scan.\nIt uses index bitmap generated by condition (ps_suppkey = s.s_suppkey) and fetch corresponding pages stored in the heap. \nThis node that has index scan in it child nodes is used by its direct parent node as nested loop join.",
            "Has Child With Index": true,
            "Node Type": "Bitmap Heap Scan",
            "Parallel Aware": false,
            "Parent Relationship": "Inner",
            "Plan Rows": 1,
            "Plan Width": 8,
            "Plans": [
                {
                    "Async Capable": false,
                    "Contain Relation": [],
                    "Explanation": "Node type Bitmap Index Scan: A bitmap index scan is used to generate a bitmap based on indices of selected attribute(s).\nIt has condition (ps_suppkey = s.s_suppkey).\nBit map index scan always has a bitmap heap scan node as its immediate parent node.",
                    "Index Cond": "(ps_suppkey = s.s_suppkey)",
                    "Index Name": "partsupp_pkey",
                    "Node Type": "Bitmap Index Scan",
                    "Parallel Aware": false,
                    "Parent Relationship": "Outer",
                    "Plan Rows": 1,
```

```
                        "Plan Width": 0,
                        "Startup Cost": 0.0,
                        "Total Cost": 1.45
                    }
                ],
                "Recheck Cond": "(ps_suppkey = s.s_suppkey)",
                "Relation Name": "partsupp",
                "Startup Cost": 1.45,
                "Total Cost": 5.46
            }
        ],
        "Startup Cost": 10000000001.59,
        "Total Cost": 10000000866.81
    },
    {
        "Alias": "p",
        "Async Capable": false,
        "Contain Relation": [
            "part"
        ],
        "Explanation": "Node type Bitmap Heap Scan: A bitmap heap scan is always a direct parent of bitmap index scan.\nIt uses index bitmap generated by condition (p_partkey = ps.ps_partkey) and fetch corresponding pages stored in the heap. \nThis node that has index scan in it child nodes is used by its direct parent node as nested loop join.",
        "Has Child With Index": true,
        "Node Type": "Bitmap Heap Scan",
        "Parallel Aware": false,
        "Parent Relationship": "Inner",
        "Plan Rows": 1,
        "Plan Width": 24,
        "Plans": [
            {
                "Async Capable": false,
                "Contain Relation": [],
                "Explanation": "Node type Bitmap Index Scan: A bitmap index scan is used to generate a bitmap based on indices of selected attribute(s).\nIt has condition (p_partkey = ps.ps_partkey).\nBit map index scan always has a bitmap heap scan node as its immediate parent node.",
                "Index Cond": "(p_partkey = ps.ps_partkey)",
                "Index Name": "part_pkey",
                "Node Type": "Bitmap Index Scan",
```

```
                        "Parallel Aware": false,
                        "Parent Relationship": "Outer",
                        "Plan Rows": 1,
                        "Plan Width": 0,
                        "Startup Cost": 0.0,
                        "Total Cost": 0.18
                      }
                    ],
                    "Recheck Cond": "(p_partkey = ps.ps_partkey)",
                    "Relation Name": "part",
                    "Startup Cost": 0.18,
                    "Total Cost": 4.19
                  }
                ],
                "Startup Cost": 10000000001.77,
                "Total Cost": 10000001574.94
              }
            ],
            "Startup Cost": 10000000001.77,
            "Strategy": "Sorted",
            "Total Cost": 10000001577.66
        }
    }
```