



CZ4031 Database System Principles

School of Computer Science and Engineering
AY22/23 Semester 1

Project 2

Connecting SQL Query with Query Plans

Group No: 47

Zhang Yinghao (N2202302J)
Gupta Suhana (U1923230B)
Michelle Lam Su-Ann (U2021548K)
Jiang Yuxin (U2022749J)
Wang Qianteng (U2022039K)

Due date: Nov 13, 2022; 11:59 PM

Table of Contents

1	<i>Introduction.....</i>	3
2	<i>TPC-H Dataset.....</i>	3
3	<i>Details of the Software</i>	4
3.1	Overall File Structure	4
3.2	Libraries Used	4
3.3	Installation Instructions.....	4
3.4	GUI Overall Layout and Features	5
3.4.1	Error Handling.....	5
3.4.2	Query Plan Visualiser Screen – View Tree Feature	5
3.4.3	Query Plan Visualiser Screen – View More Information Feature	6
3.5	Navigating the Software.....	7
3.6	interface.py.....	10
3.6.1	main_account_screen.....	10
3.6.2	QEP_visualizer Class	11
3.6.3	draw_chart Class.....	11
3.7	annotation.py	11
3.7.1	PlanNode Class.....	11
3.7.2	PlanTree Class.....	12
3.8	preprocessing.py	13
3.8.1	Preprocessor Class	13
4	<i>Examples</i>	14
4.1	Query 1.....	15
4.2	Query 2.....	19
4.3	Query 3.....	26
4.4	Query 4.....	33
4.5	Query 5.....	38
6	<i>Limitations.....</i>	45
6.1	Performance.....	45
6.2	Security of Login Interface.....	45
6.3	Limited Retrieval of AQPs	45
6.4	Limited Flexibility of Information Disclosure on Interface.....	45
6.5	Consistency	45
6.6	Support of PostgreSQL	46
7	<i>Contribution</i>	47
8	<i>References</i>	48

1 Introduction

Real-world uses of the relational database management system (RDBMS) involve users writing SQL queries as input to the system. This is followed by the RDBMS query optimiser selecting the optimal (in terms of time) query execution plan (QEP) from a pool of alternative query plans (AQP).

The issue raised during this process results from the separation of the query optimiser and SQL query. In other words, users are not able to see the connection between QEP, AQP and its matching SQL query. This project aims to address this issue by developing a GUI where the input SQL query can be annotated with explanations of each component is executed and why. The TPC-H dataset is used to test software functions.

Please refer to the Instructions.pdf file for the instructions on how to use this software.

2 TPC-H Dataset

The dataset used to test software functionality are derived from the “The Transaction Processing Performance Council”. Specially the TPC-H decision support benchmark is used.

TABLE 1
Details of relations in TPC-H.

Relation	Description	Cardinality
REGION	Continents the countries are located in.	5
NATION	Countries supported in the dataset	25
SUPPLIER	Suppliers’ details (name, address, phone number, ...)	Approx. 10,000
CUSTOMER	Customers’ details (name, address, phone number, ...)	Approx. 150,000
PART	Parts’ details (name, brand, type, size, ...)	Approx. 200,000
PARTSUPP	Details of the supply of a part (supplier, part, availability, cost, ...)	Approx. 800,000
ORDERS	Details of customer orders (customer, order status, price, date, ...)	Approx. 1,500,000
LINEITEM	Details of items in an order (part, supplier, quantity, ...)	Approx. 6,000,000

3 Details of the Software

3.1 Overall File Structure

The main structure of our project is as follows:

3. **project.py** – This is the main file used to invoke all necessary functions from the following three files below.
3. **interface.py** – This file is responsible for the generation of all necessary user interfaces. This includes: (1) Login interface, (2) Query input interface, (3) QEP annotation interface, and (4) AQP annotation interface
3. **annotation.py** – This file is responsible for generating the query plan tree and annotating each node, for every query plan generated.
3. **preprocessing.py** – This file is responsible for connecting to the PostgreSQL server, retrieving the optimal QEP, retrieving the relevant AQPs, and pre-processing the information received before passing it into *annotations.py* for annotations.

3.2 Libraries Used

The main libraries required are as follows:

- psycopg2
 - PostgreSQL database adapter for the Python programming language. [1]
- json
 - To work with JSON files from the PostgreSQL's query optimiser. [2]
- tkinter
 - Library for GUI development
- Pmw
 - Toolkit for building high-level compound widgets, or megawidgets, constructed using other widgets as component parts. [3]

3.3 Installation Instructions

1. Ensure python version is 3.8 and above
2. Pip install psycopg2
 - a. If this install does not work, use Pip install psycopg2-binary
3. Pip install tk
4. Run the project.py file

3.4 GUI Overall Layout and Features

3.4.1 Error Handling

This software has included features to handle errors at the login interface and query statement screen. The possible error messages that the user can encounter are as follows:

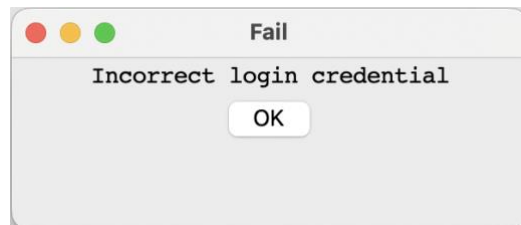


Fig.1. Login error message.

The login error is triggered if the user did not enter the correct login credentials required as shown in Fig.1. This error message is displayed when any one or more of the login credentials entered is wrong. To resolve this, the user will need to enter all the required login fields in the login screen correctly.

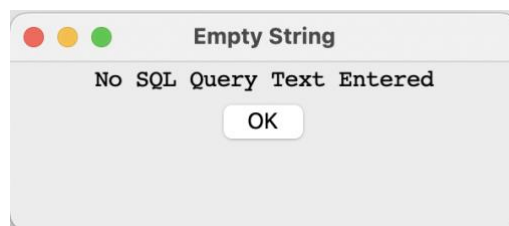


Fig.2. SQL statement error message.

The SQL statement error is triggered if the user attempts to proceed to view the optimal QEP tree or view the AQP tree without entering any query statement as shown in Fig.2. To resolve this, the user will need to enter a query statement into the enter query statement screen.

3.4.2 Query Plan Visualiser Screen – View Tree Feature

The steps to navigate between interfaces and view the optimal query plan or alternate query plan are discussed in Section 3.4. When the user has successfully logged in and entered a valid query, the query plan visualizer screen will display. An example of a node of the overall format of the query plan visualiser's output is shown in Fig.3.

Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

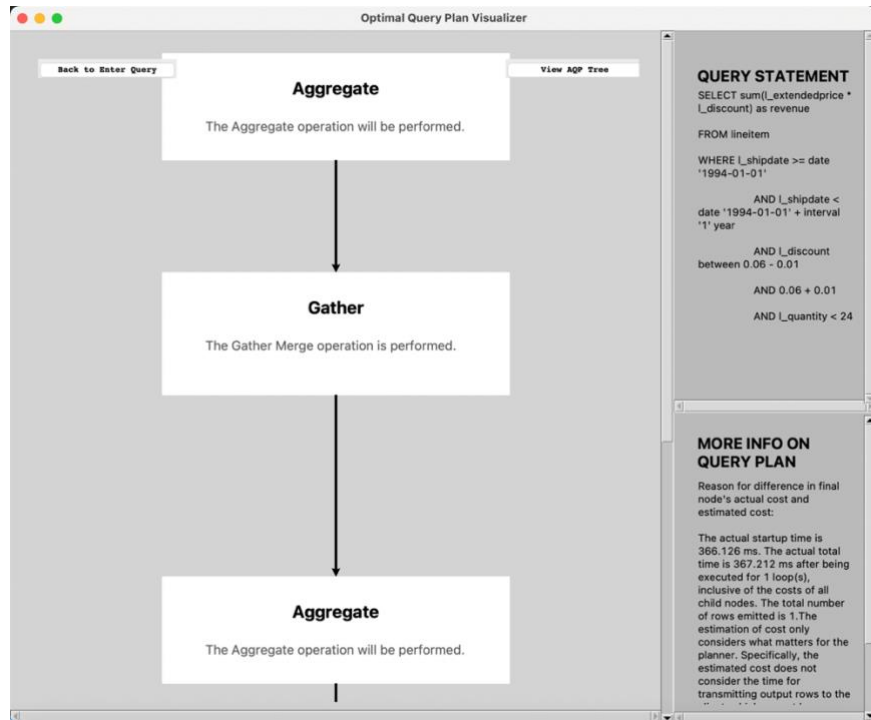


Fig.3. An optimal QEP generated by PostgreSQL query optimiser.

3.4.3 Query Plan Visualiser Screen – View More Information Feature

The user can view more information related to each tree node by hovering the cursor over the white portion of each rectangle as shown in Fig.4.

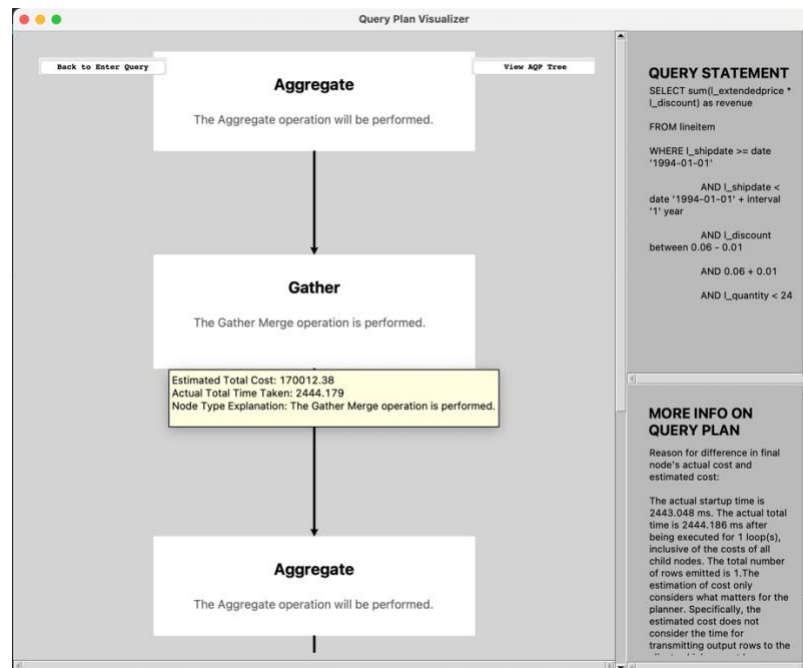


Fig.4. View More Information Feature

More details and explanation about software navigation can be found in section 3.5.

3.5 Navigating the Software

To use the software, the user is prompted to enter their login credentials to connect to the PostgreSQL server (Fig.5). The login credentials required are: (1) Username, (2) Password, and (3) Database Name.

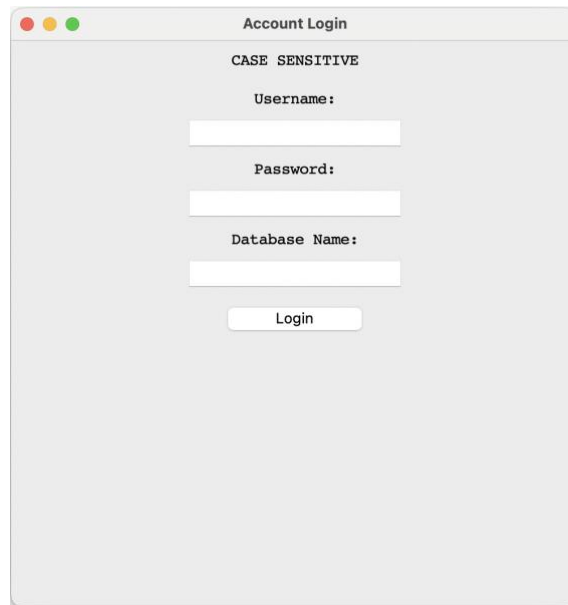
A screenshot of a macOS-style window titled "Account Login". The window has a light gray background and standard macOS window controls (red, yellow, green buttons) in the top-left corner. Inside the window, the text "CASE SENSITIVE" is displayed in a small, all-caps font. Below this, there are three labels: "Username:", "Password:", and "Database Name:", each followed by a white rectangular input field. At the bottom center of the window is a "Login" button.

Fig.5. Login Screen

If any of these login credentials are inputted incorrectly, the user is denied from using the software. An error message will be displayed to inform the user that the login credential entered is incorrect. Once the user enters all login credentials correctly, user gain access to enter a query statement as shown in Fig.6.

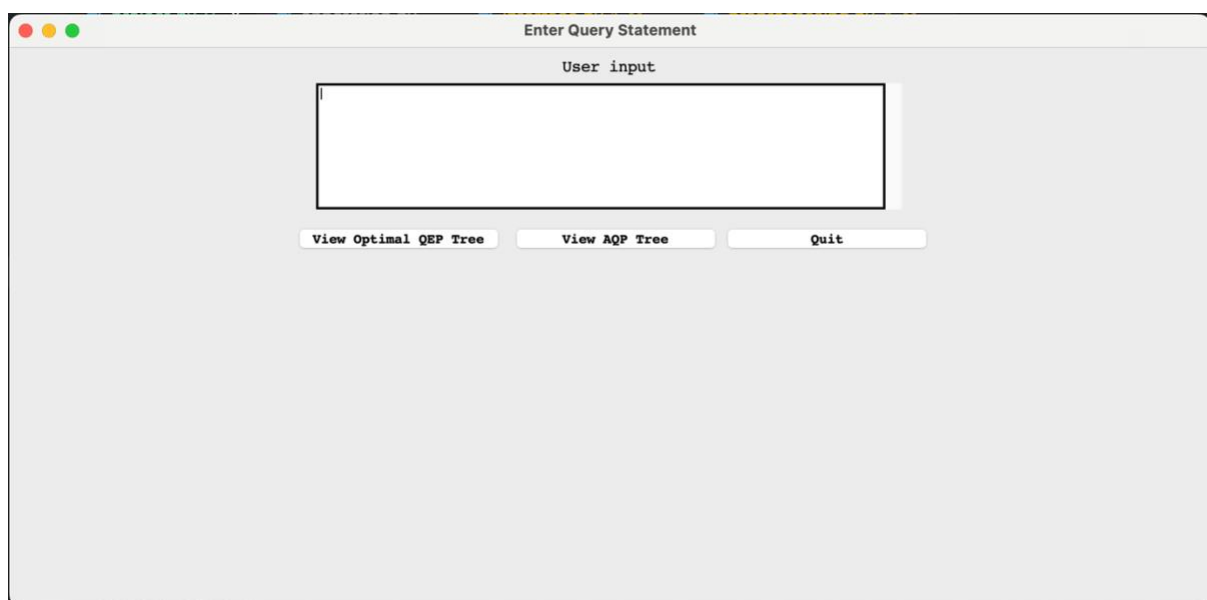
A screenshot of a macOS-style window titled "Enter Query Statement". The window has a light gray background and standard macOS window controls in the top-left corner. Inside the window, the text "User input" is displayed in a small, all-caps font. Below this is a large white rectangular text area for entering a query. At the bottom of the window, there are three buttons: "View Optimal QEP Tree", "View AQP Tree", and "Quit".

Fig.6. Enter Query Statement Screen

Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

The user cannot proceed if no query statement is entered. An error message will be displayed to inform the user that no query statement has been entered yet.

Once the user has entered a query statement, clicking on the “View Optimal QEP tree” button generates and displays an Optimal Query Plan tree with relevant information. The user can also click on “View AQP tree” button to view the Alternate Query Plan tree with relevant information. For example, the user enters a query statement and clicks on the “View the Optimal QEP tree” button (Fig.7).

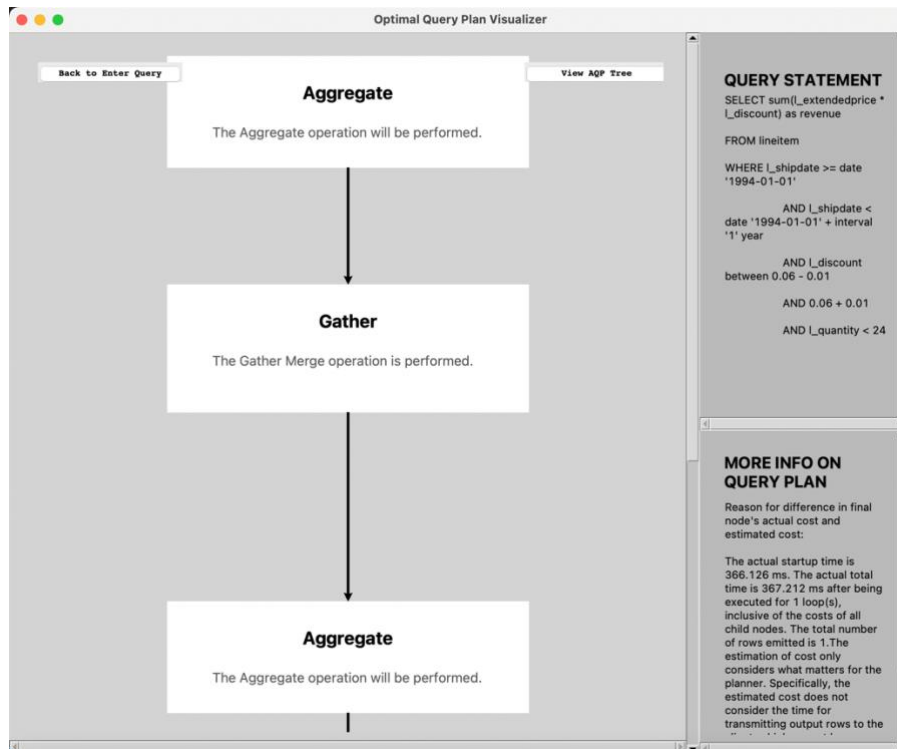


Fig.7. View Optimal QEP Screen (Query Plan Visualizer Screen)

The user can use the respective scrollbars to view the entire tree generated on the left side, and to view more information on the right side. To view more information for each tree node, the user can hover the cursor over the node (i.e., white areas of each rectangle). A tooltip will appear, displaying the detail information for the tree node the cursor is currently hovering over (Fig.8).

Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

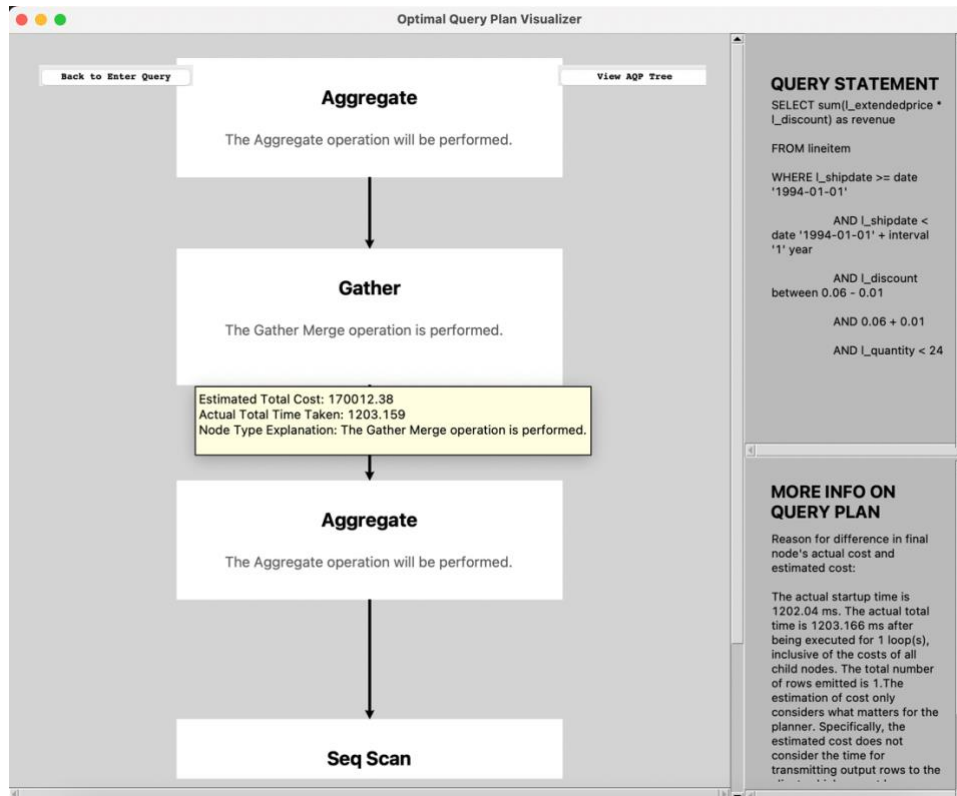


Fig.8. View More Information Tooltip

From the optimal query plan visualizer screen (Fig.7 and Fig.8), the user can also choose to view the AQP tree by clicking on “View AQP Tree” button or go back to the query statement screen by clicking on the “Back to Enter Query” button to enter a new query statement.

A similar GUI interface is displayed if the user clicks the “View the AQP” button from the enter query statement screen (Fig.6) or the “View AQP Tree” button from the optimal QEP screen (Fig.8). The AQP tree and information will be displayed (Fig 9).

Similarly, the user can hover over each tree node to view more information as shown in Fig.8. From the AQP screen, the user can click on the “Back to Enter Query” button to go back to the query statement screen to enter a new query statement.

Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

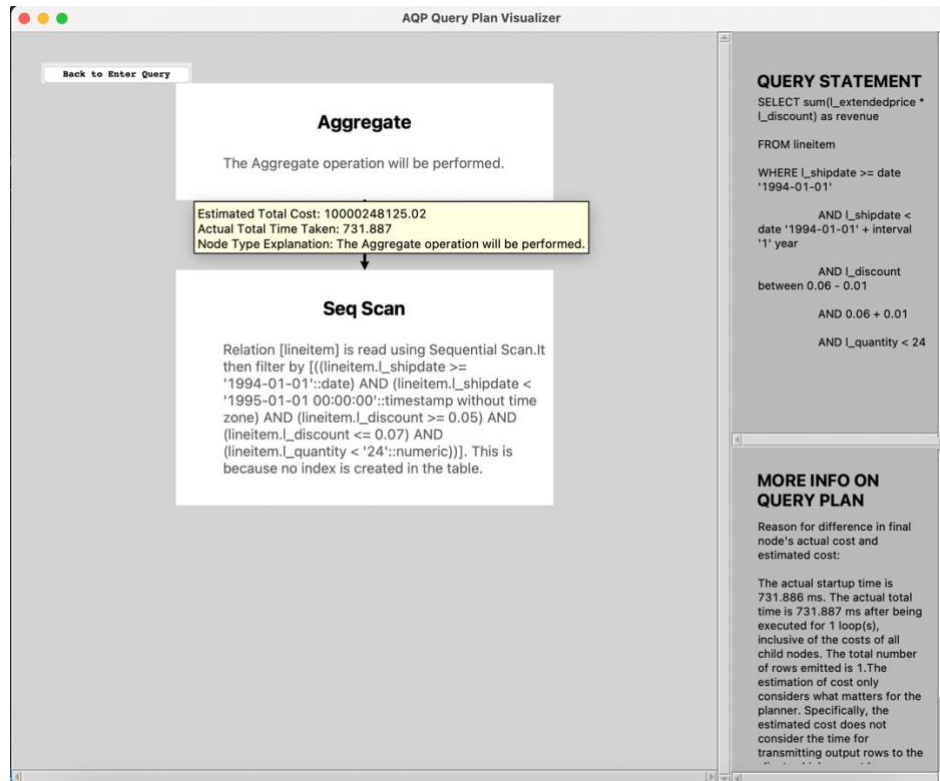


Fig.9. View Alternate Query Plan Screen

Note: It may take some time for the software to process the input query. The user may have to wait for a while before the result is shown on the screen.

3.6 interface.py

3.6.1 main_account_screen

Logic to establish the main login screen. Contains functions:

1. `validatelogin()`
 - verify username, password and database name inputted by user with PostgreSQL DBMS server.
2. `login_fail()`
 - Prompt to user that login failed due to incorrect login credential/s.
3. Functions to remove windows
 - `delete_login_fail()`
 - `delete_main()`
 - `delete_guiforSQL()`
 - `delete_empty_string()`

To generate the QEP visualization, 2 classes are implemented: `QEP_visualizer` class and `draw_chart` class.

3.6.2 QEP_visualizer Class

Each QEP_visualizer class object represents the entire QEP visualizer screen, which contains 3 components – (1) a query tree of the query plan on the left side, (2) a formatted version of the query statement entered on the top right corner, and (3) more information regarding the cost of each node on the lower left corner.

This class contains all the methods involved in generating these 3 components as mentioned above.

3.6.3 draw_chart Class

Each draw_chart class object created represents the query tree that will be generated. It contains 2 main methods – *draw_node()* and *draw_query_flowchart()*.

The *draw_query_flowchart()* method is invoked in the QEP_visualizer class. This method contains the algorithm to generate the query plan tree. It is a recursive function that will draw its root node, and then draw each node's children recursively. Each node will have a maximum of 2 children and a minimum of 1 child. If the node has 2 children, a split is needed to generate a left child node and a right child node. Otherwise, nodes with 1 child will generate a child node directly below it. To determine whether a split is necessary, the number of children for each node must be checked first. Thereafter, each node can continue to be generated recursively.

The *draw_node()* method is called within *draw_query_flowchart()*. This method is responsible for creating each node (i.e., rectangle in the tree) and its relevant information, such as the node type, node explanation, actual cost, and estimated cost.

3.7 annotation.py

To facilitate the annotation function of the GUI, two class are implemented: PlanTree and PlanNode.

3.7.1 PlanNode Class

Each PlanNode object represents a node of the QEP tree outputted by the PostgreSQL. The node maintains its set of child nodes.

When a new PlanNode object is initialised by passing in a query plan, it will iterate through the plan from top to bottom and initialize all child nodes using the *add_children()* function.

The information of each node as shown in Fig.1. is parsed through multiple functions and stored as attributes of the PlanNode object:

1. *parse_plan()*
 - Parse the node type, i.e., the corresponding operator of the node.
2. *parse_cost()*

- Parse the information returned from EXPLAIN command about the estimated cost and runtime cost.
- 3. `parse_misc()`
 - Parse the information returned from EXPLAIN command about miscellaneous information of the query plan, including parent relationship, plan width, and output.
- 4. `parse_filter()`
 - Parse the information returned from EXPLAIN command about the filter used, i.e., selection condition.

To explain the node information, i.e., add annotation to the operators to explain their underlying algorithm, the following functions are implemented:

1. `explain_node_type()`
 - Output a string explain the algorithm or technique selected for the corresponding operator. The function translates the Json file to natural sentences with algorithm-related key attributes and adds explanation of the algorithm based on heuristics of QEPs.

e.g., Sequential Scan:

```
if self.node_type == "Seq Scan":
    explain += "Relation [{}] is read using Sequential Scan.".format(self.query_plan['Relation Name'])
    if self.hasFilter:
        explain += 'It then filter by [{}].'.format(self.filter['Filter'])
    explain += " This is because no index is created in the table."
```

2. `explain_estimated_cost()`
 - Output a string explain the related costs and output of this node, including:
 - Cost accumulated
 - Cost upon completion of the current node
 - Number of output rows
 - Average length of each row

In addition, an `explain_actual_cost()` function is also implemented, with `explain_cost_diff()` to address to the user about the discrepancy in execution time.

3.7.2 PlanTree Class

Each PlanTree object represents the entire QEP of the input query. From the input query, it generates PlanNode and store the head of the tree. QEP information such as node types, cost can be extracted by traversing the tree and calling explain functions of each node.

1. `get_all_node_types()`
 - Extract node type of the PlanNode object stored.
2. `get_more_info()`
 - Extract cost discrepancy information of the PlanNode stored.
3. `get_cost_infor()`
 - Extract estimated and actual cost info of the PlanNode stored.

3.8 preprocessing.py

3.8.1 Preprocessor Class

Each Preprocessor object is used to establish connection with the PostgreSQL DBMS and extract QEP and AQP for the given SQL query.

1. `establish_connection(self, database, user, password)`
 - To verify the if the login recredential used are valid for connection to PostgreSQL database server
2. `connect(self)`
 - Connect to the PostgreSQL database server.
3. `execute_query_qep(self, query)`
 - The algorithm to retrieve the optimal query plan with default settings. With PostgreSQL EXPLAIN command, the PostgreSQL's planner will generate an execution plan. The ANALYZE option causes the statement to be executed. The execution plan will be output in Json format, including information such as the specific scan/join algorithm used and actual run time statistics, etc. This is useful for seeing whether the planner's estimates are close to reality.

```
final_query = "EXPLAIN (analyze, verbose, costs, format JSON)" + query
```
4. `execute_query_qep (self, query)`
 - To help in the retrieval of alternate query plans
5. `execute_query_aqp (self, query)`
 - The algorithm to retrieve the alternate query plan. To do so, the settings are manually configured, such that the query processor will be forced to execute the query without a particular condition. Our algorithm focuses on 6 types of settings, hash join, merge join, nested loop, index scan and sequential scan. If the current optimal query plan generated contains any of these 6 conditions, it will turn off the settings for that condition to generate an alternate query tree. Moreover, our algorithm will only retrieve and return 1 type of alternate query plan.

Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

```
def execute_query_aqp(self, query):
    try:
        query_plan = self.execute_query_qep1(query)
        if query_plan:
            plan_tree = PlanTree(query_plan)
            node_types = plan_tree.get_all_node_types()

            if "Hash Join" in node_types:
                self.cur.execute("SET enable_hashjoin = off")
            elif "Merge Join" in node_types:
                self.cur.execute("SET enable_mergejoin = off")
            elif "Nested Loop" in node_types:
                self.cur.execute("SET enable_nestloop = off")
            elif "Index Scan" in node_types:
                self.cur.execute("SET enable_indexscan = off")
            elif "Seq Scan" in node_types:
                self.cur.execute("SET enable_seqscan = off")
            else:
                print("NO modification has made.")

        return self.execute_query_qep1(query)

    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
        print("Please check the query statement entered: ", query)
        return None
```

○

4 Examples

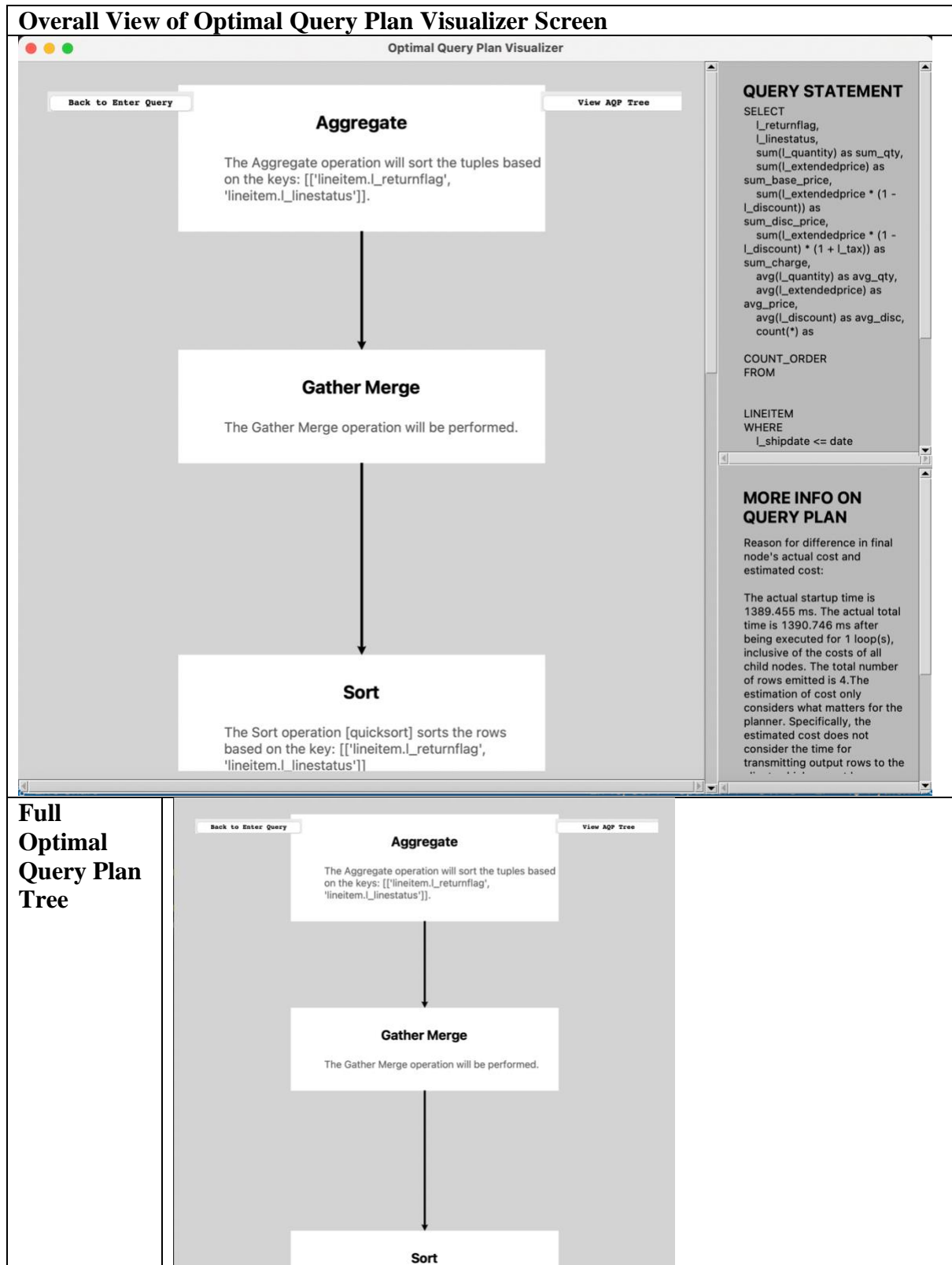
The query statement examples executed below were found on [this website](#). Each of these queries can also be found in the folder 'Queries Tested'.

4.1 Query 1

Query:

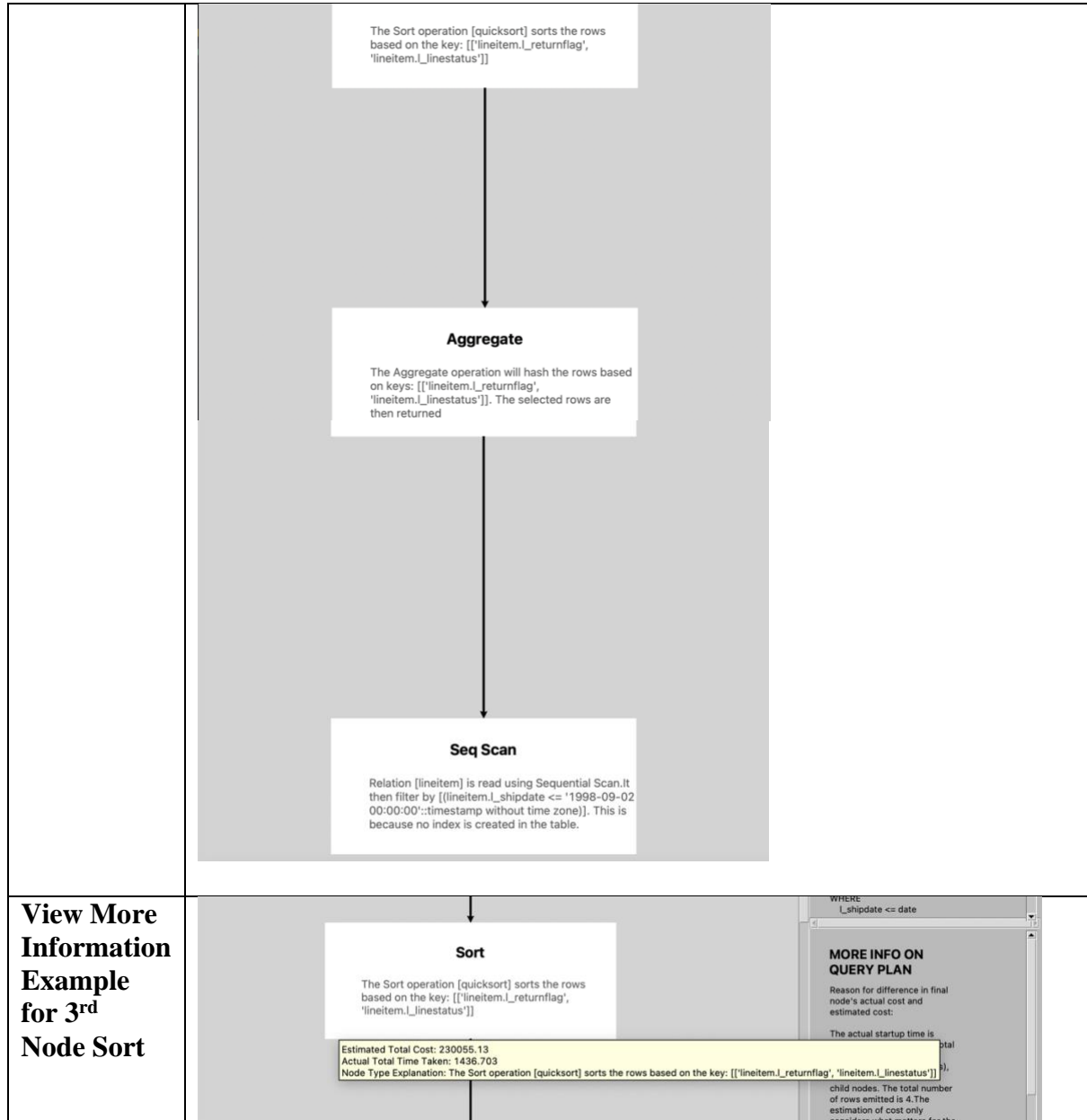
```
SELECT
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
FROM
  lineitem
WHERE
  l_shipdate <= date '1998-12-01' - interval '90' day
GROUP BY
  l_returnflag,
  l_linestatus
ORDER BY
  l_returnflag,
  l_linestatus
```

Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

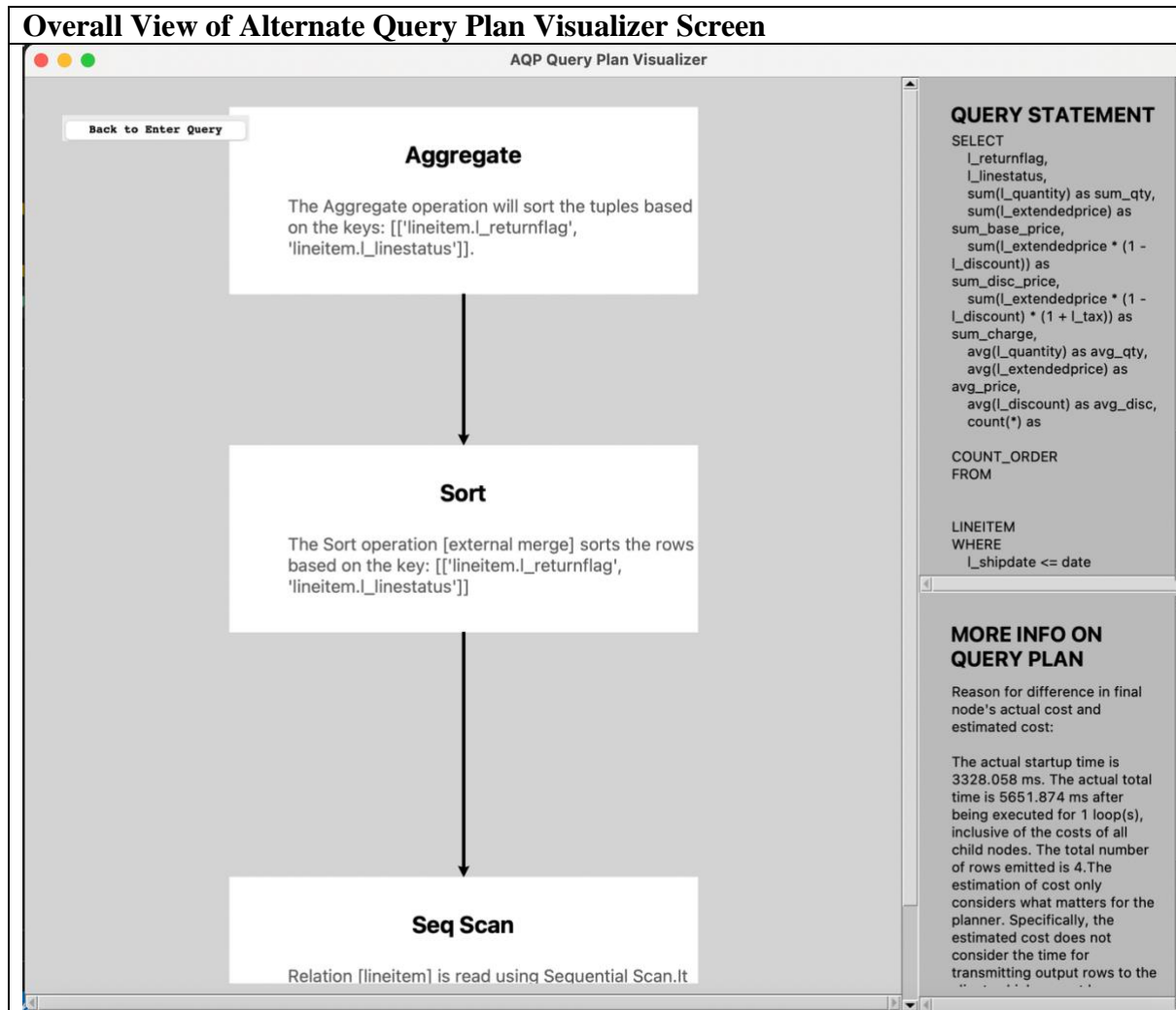


Project 2 22S1 CZ4031

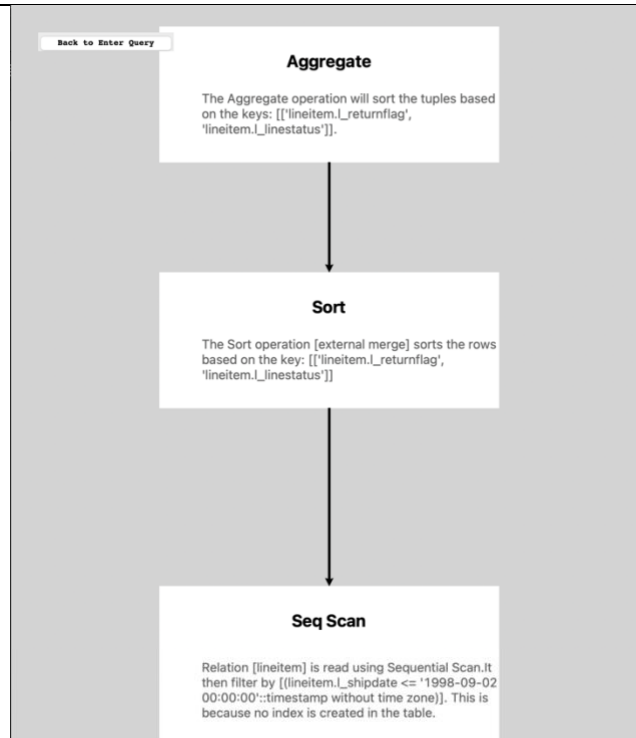
Connecting SQL Query with Query Plans



Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans



Full Alternate Query Plan Tree



Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

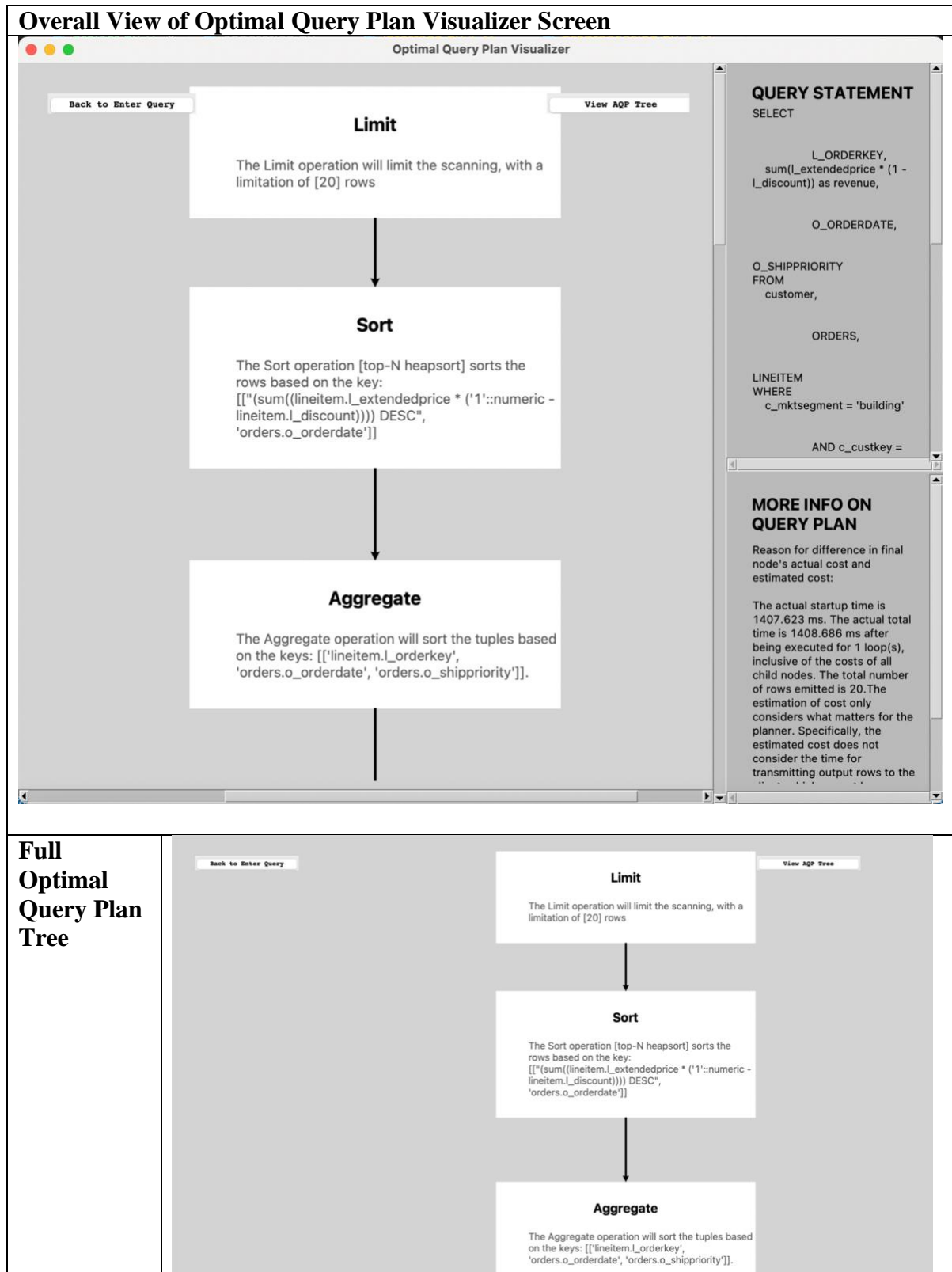
<p>View More Information Example for 1st Node Aggregate</p>	<div> <div>Back to Enter Query</div> <div> <h3>Aggregate</h3> <p>The Aggregate operation will sort the tuples based on the keys: [{"lineitem.l_returnflag", "lineitem.l_linestatus"}].</p> <p>Estimated Total Cost: 10001358170.19 Actual Total Time Taken: 5651.874 Node Type Explanation: The Aggregate operation will sort the tuples based on the keys: [{"lineitem.l_returnflag", "lineitem.l_linestatus"}].</p> </div> <div> <h4>QUERY STATEMENT</h4> <pre>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,</pre> </div> </div>
---	--

4.2 Query 2

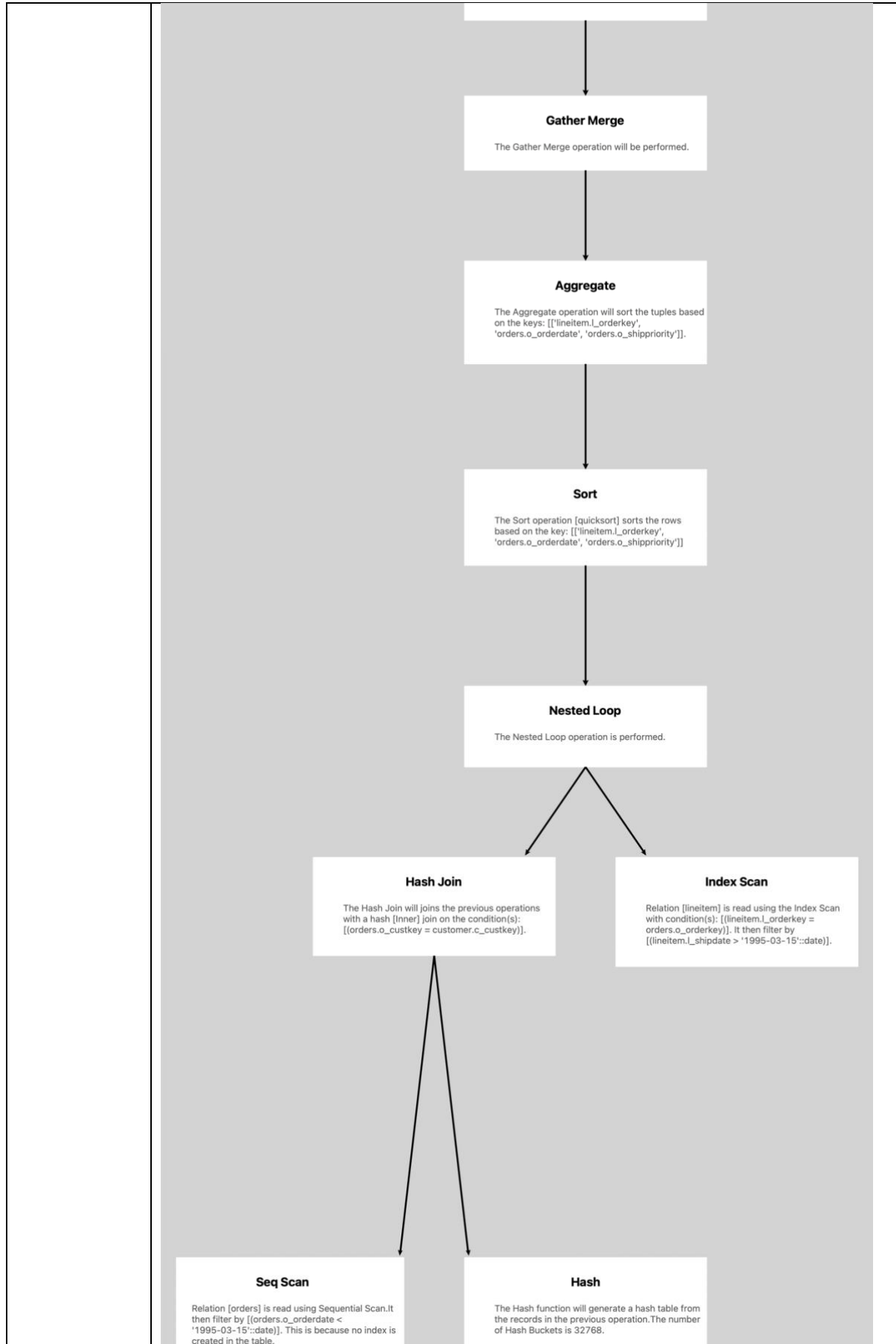
Query:

```
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_orderdate < date '1995-03-15'
  AND l_shipdate > date '1995-03-15'
GROUP BY
  l_orderkey,
  o_orderdate,
  o_shippriority
ORDER BY
  revenue desc,
  o_orderdate
LIMIT 20
```

Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

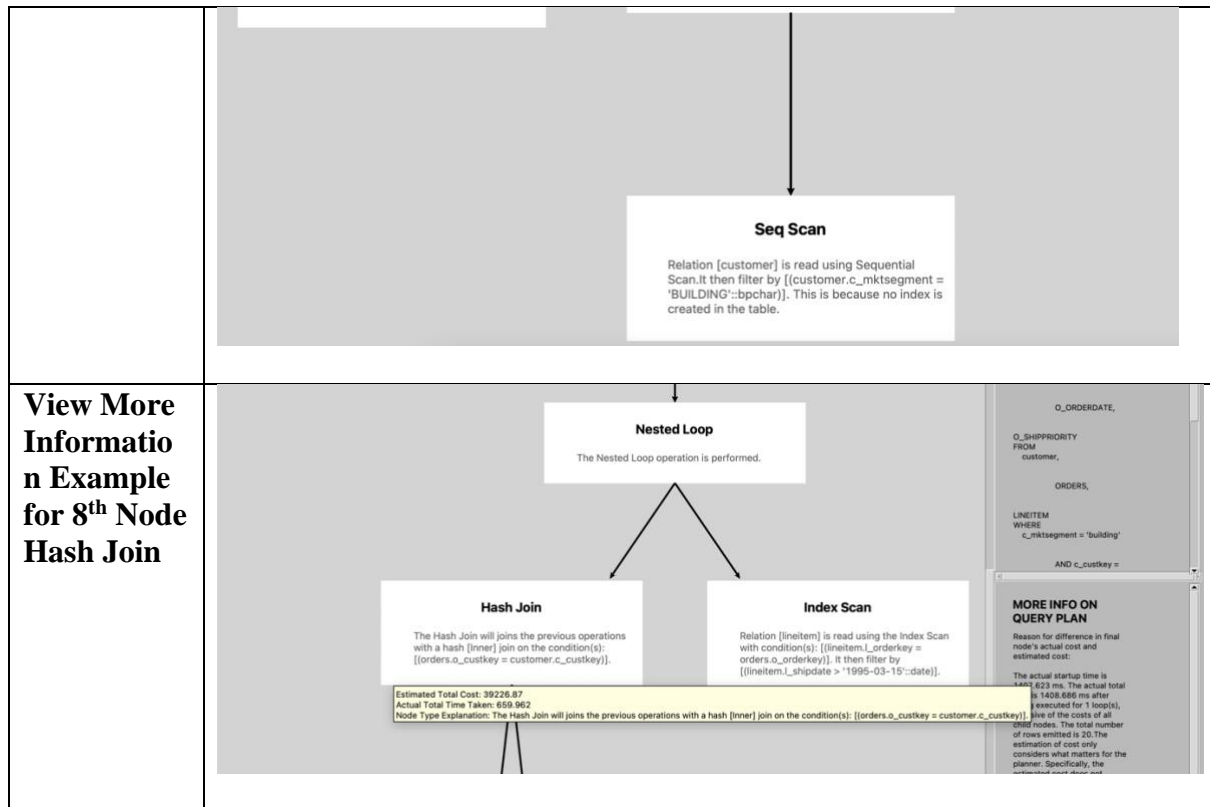


Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

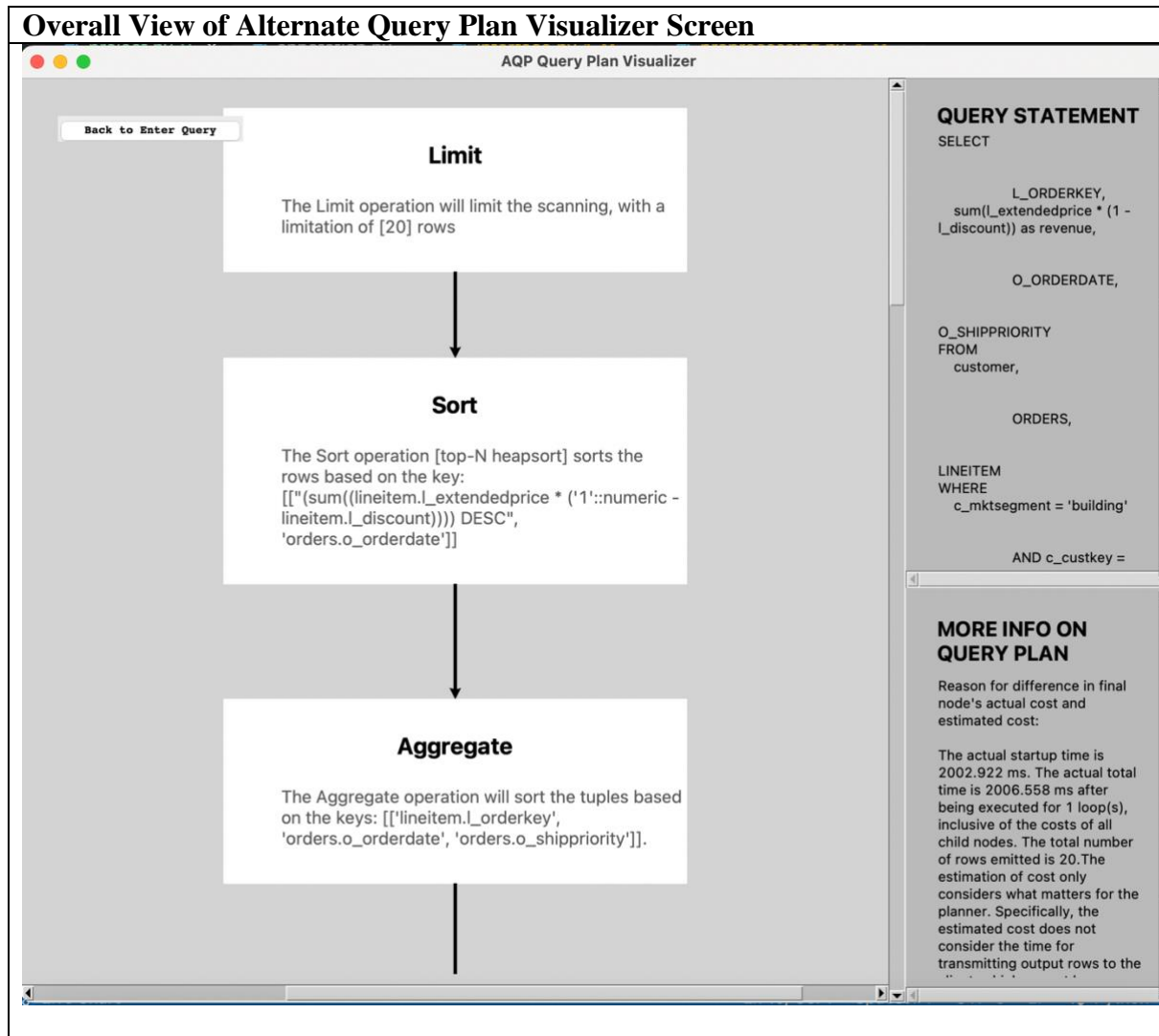


Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

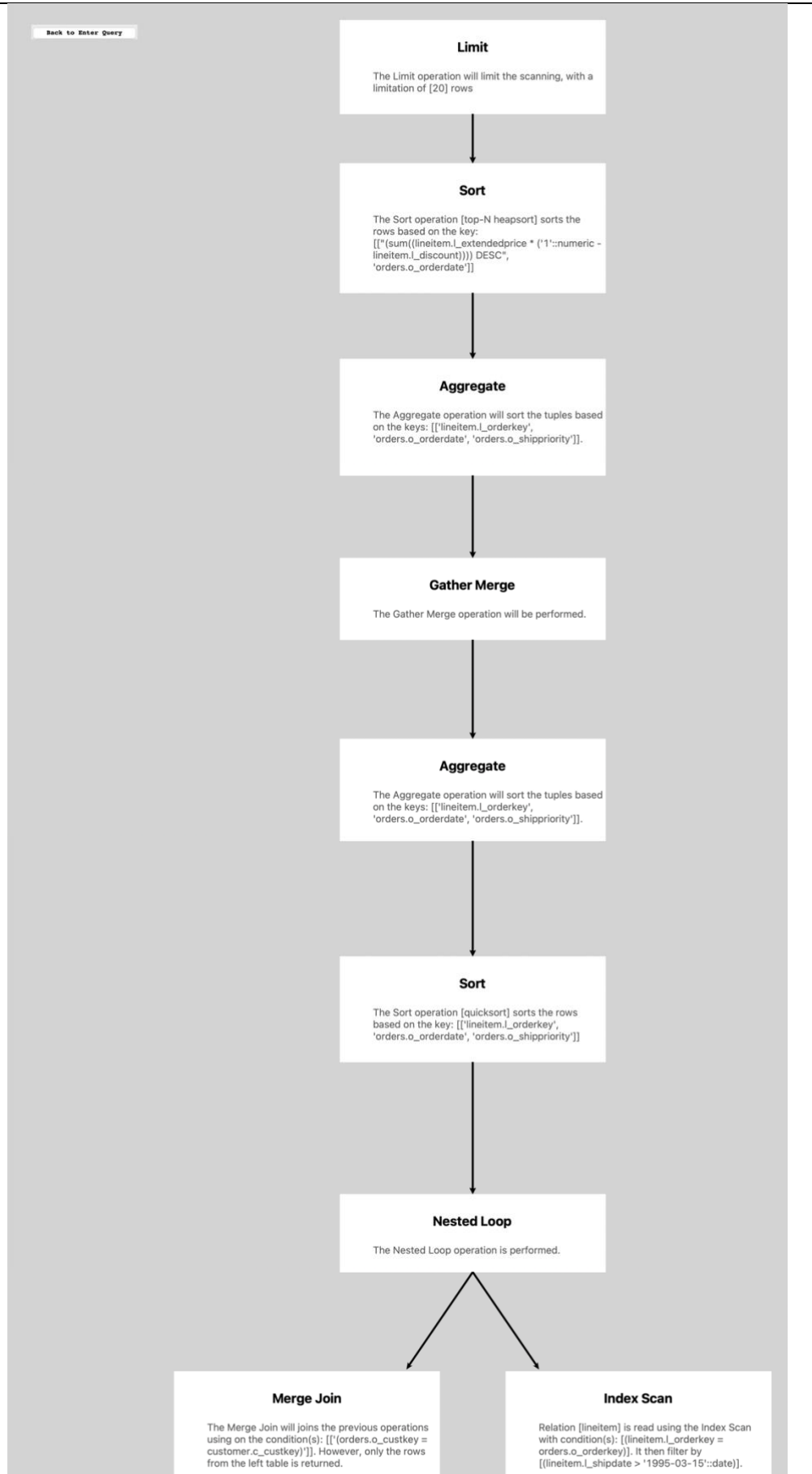


Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans



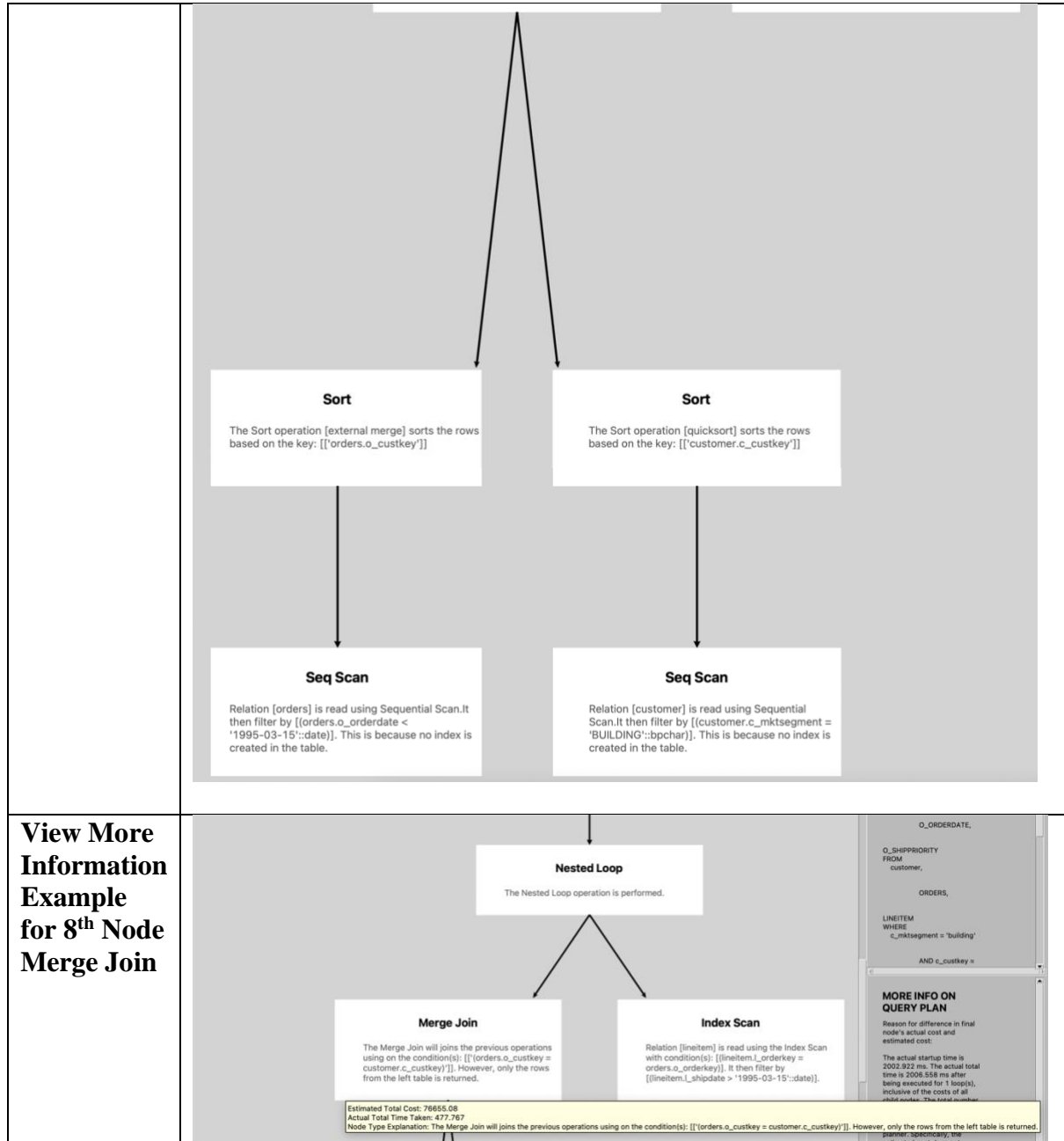
Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

**Full
Alternate
Query Plan
Tree**



Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans



**View More
Information
Example
for 8th Node
Merge Join**

4.3 Query 3

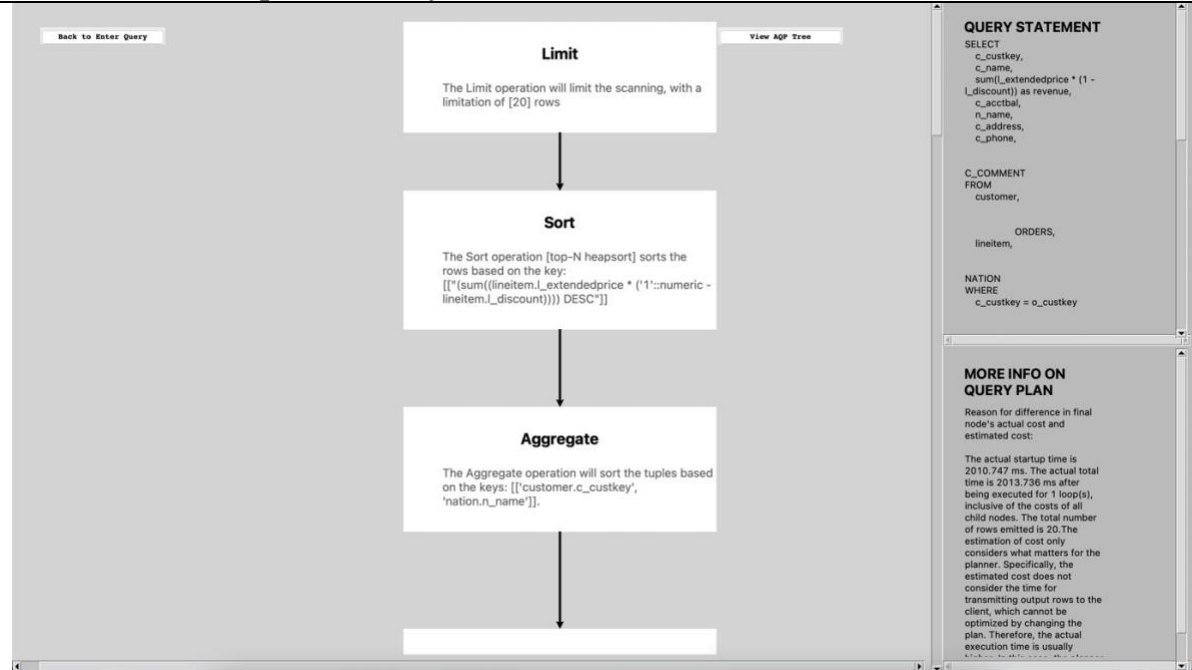
Query:

```
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 >= date '1993-10-01'
  AND o_orderdate < date '1993-10-01' + interval '3' month
  AND l_returnflag = 'R'
  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
LIMIT 20
```

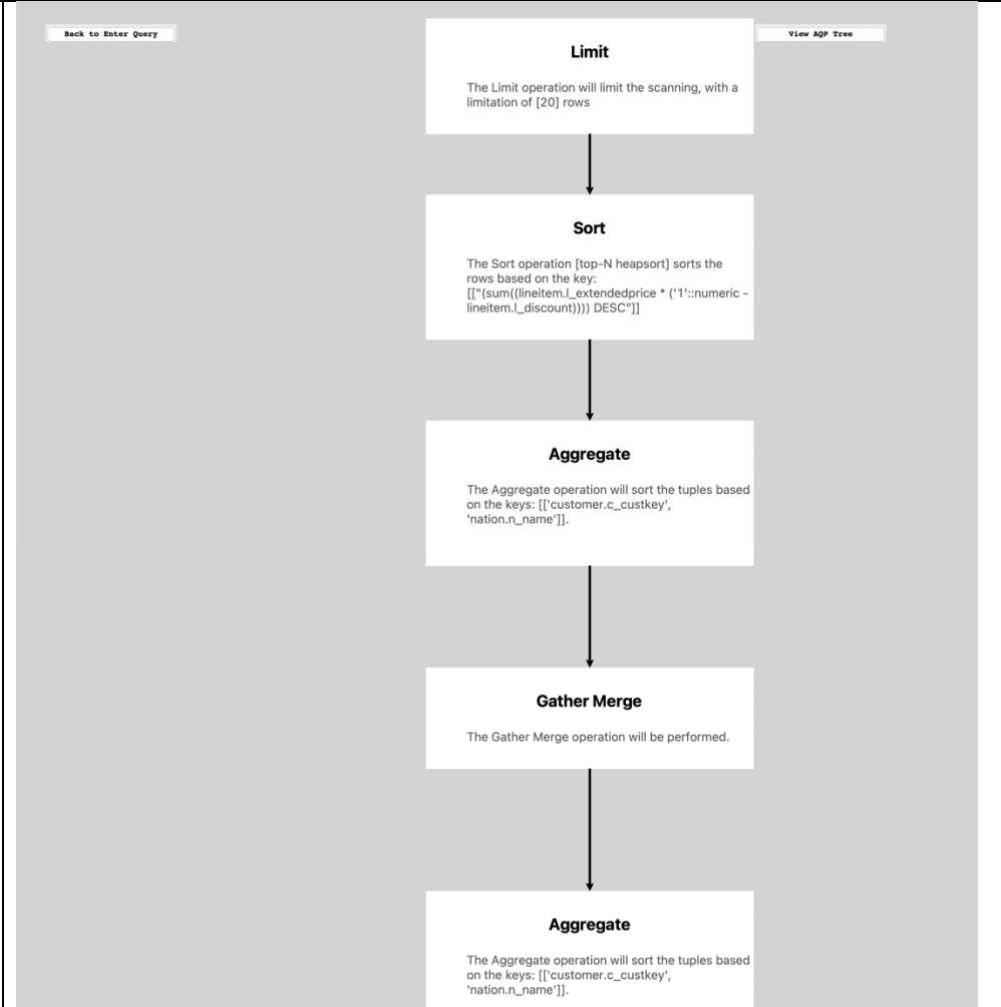
Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

Overall View of Optimal Query Plan Visualizer Screen

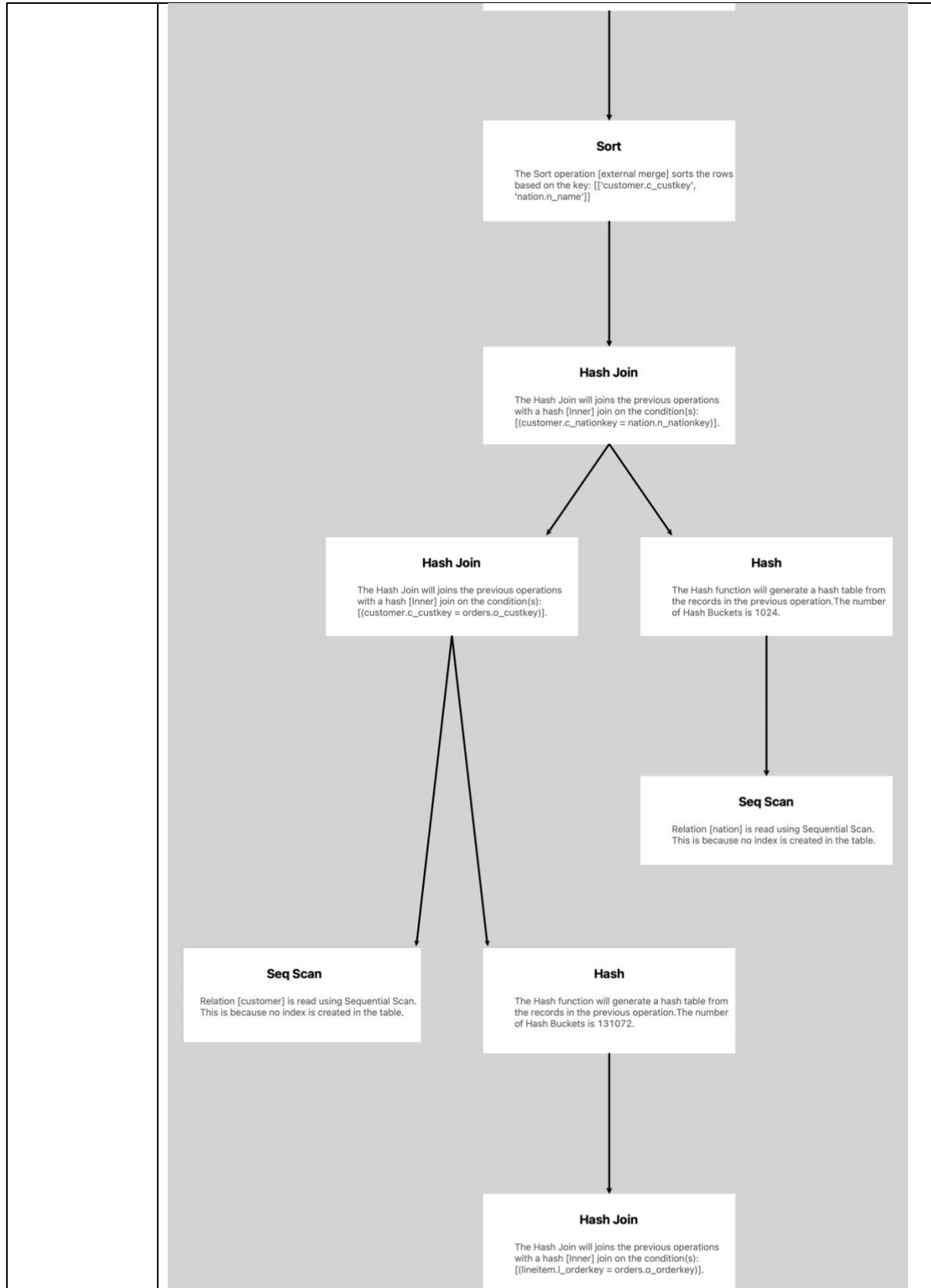


Full Optimal Query Plan Tree



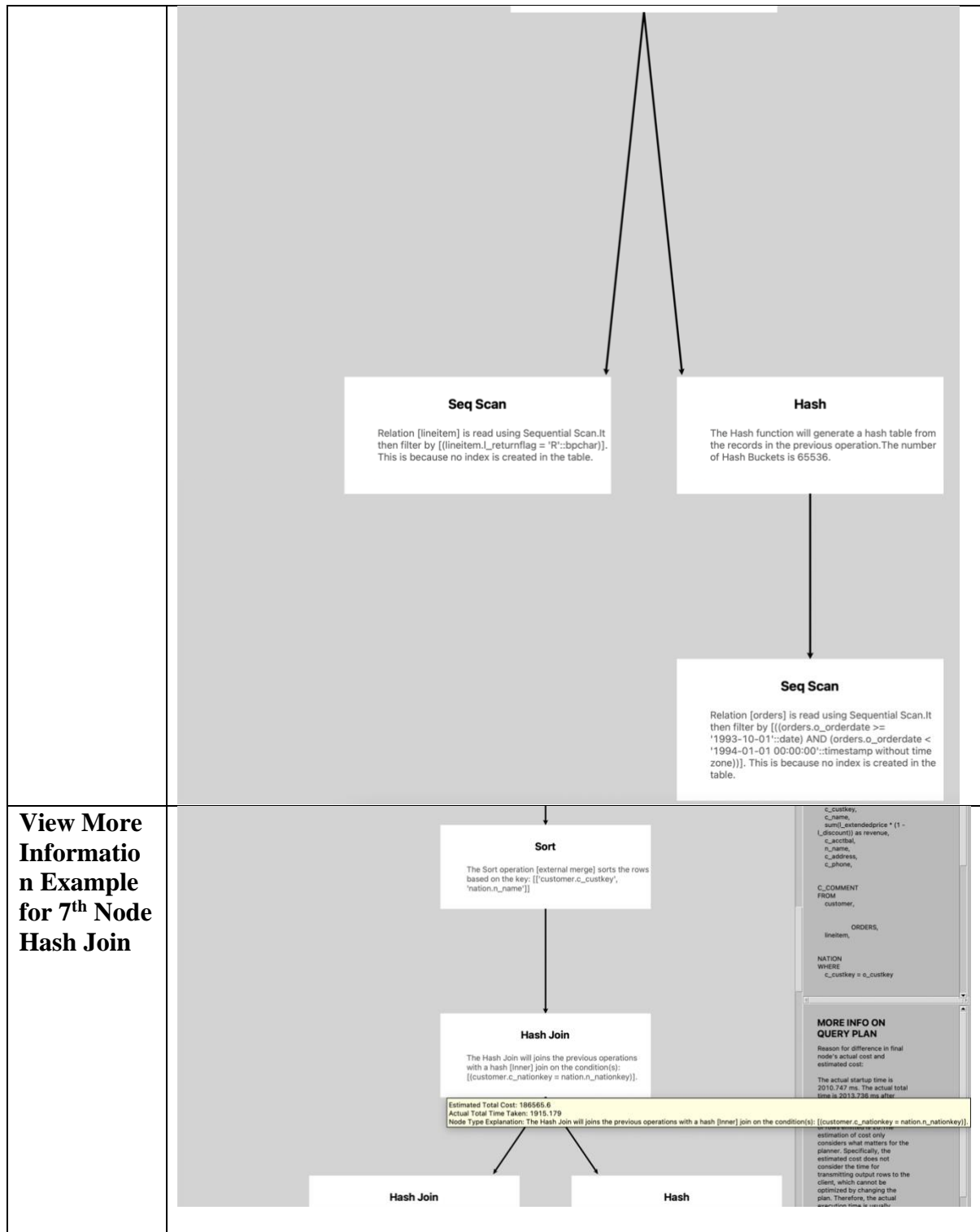
Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

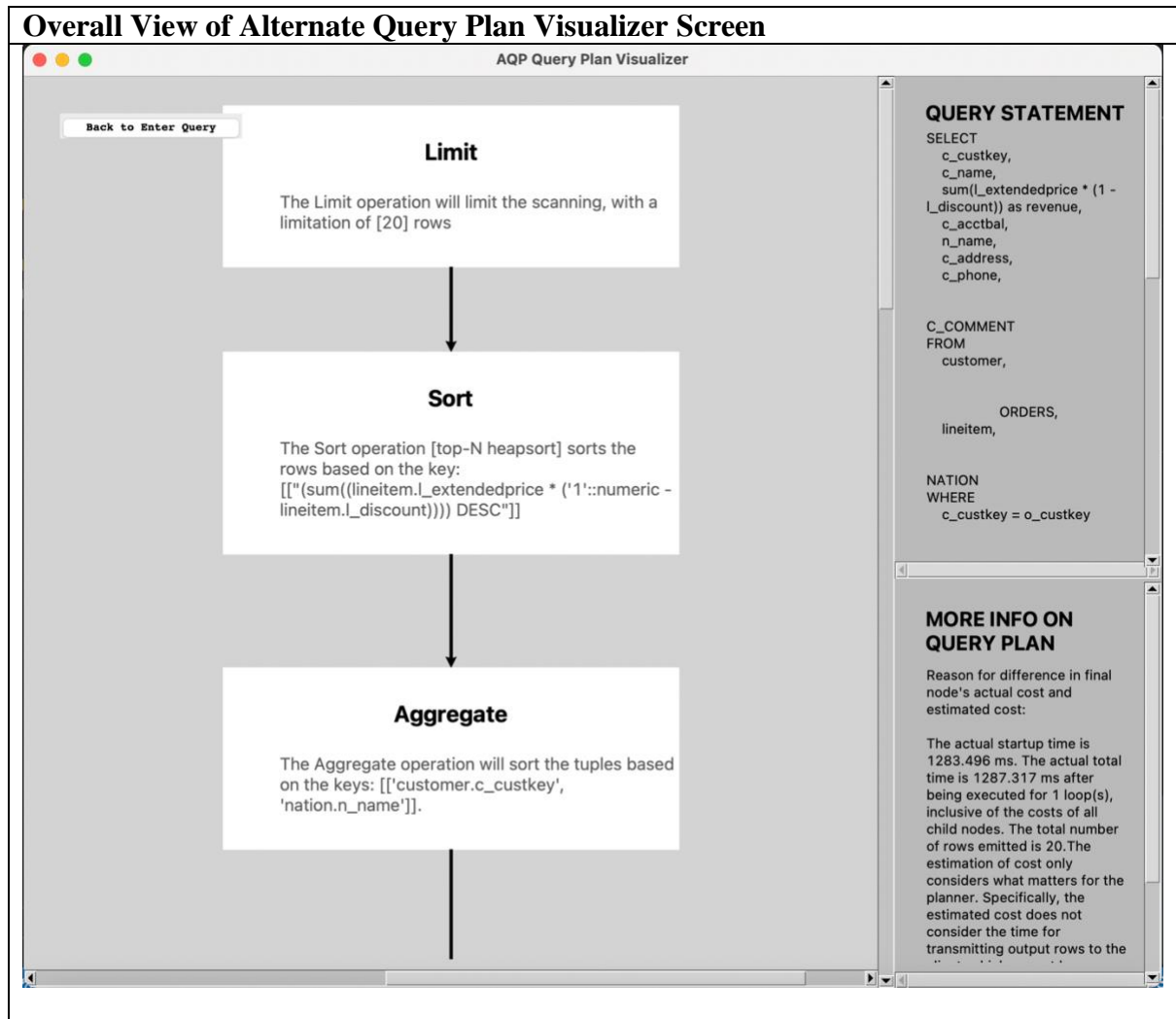


Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

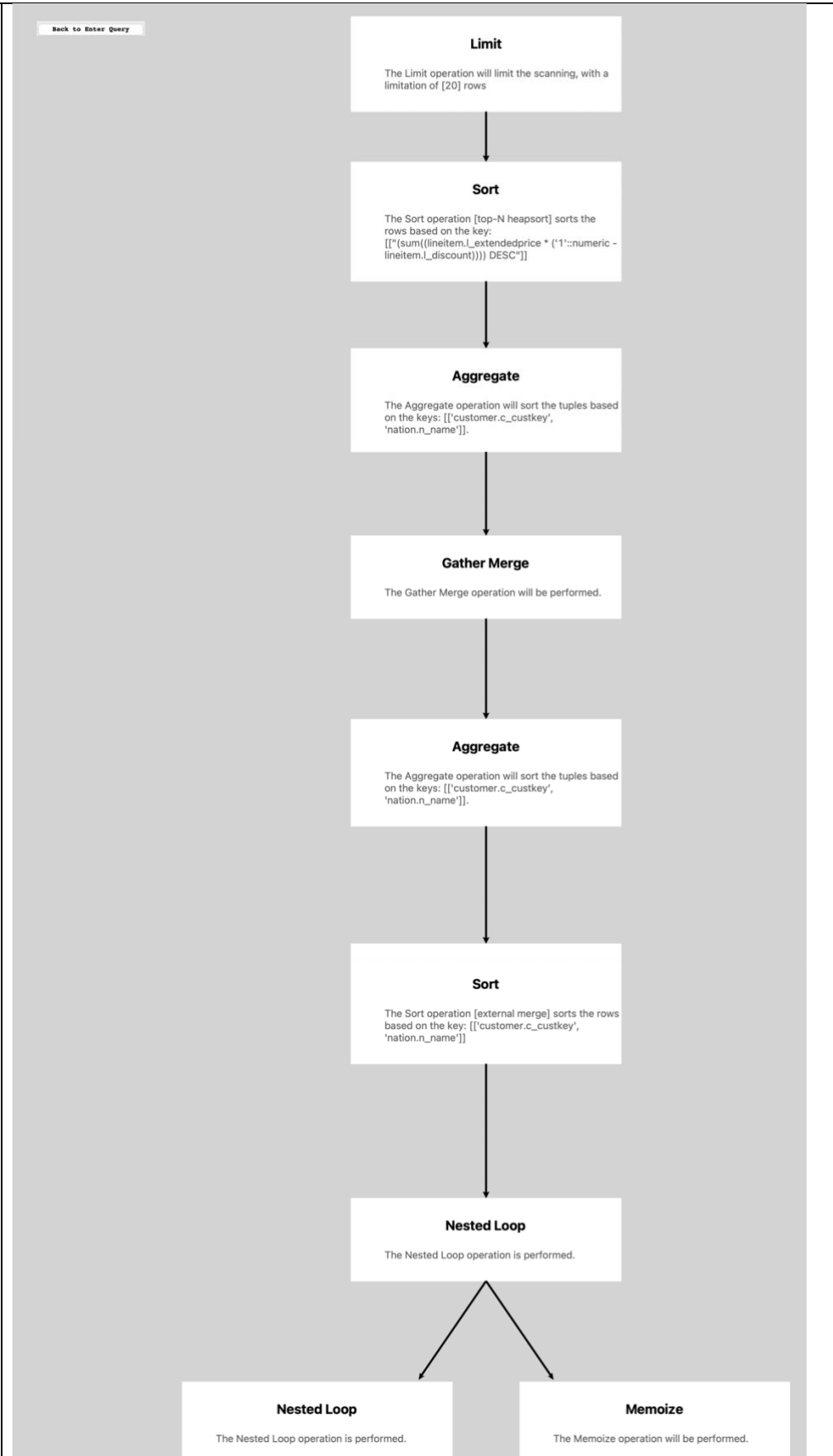


Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans



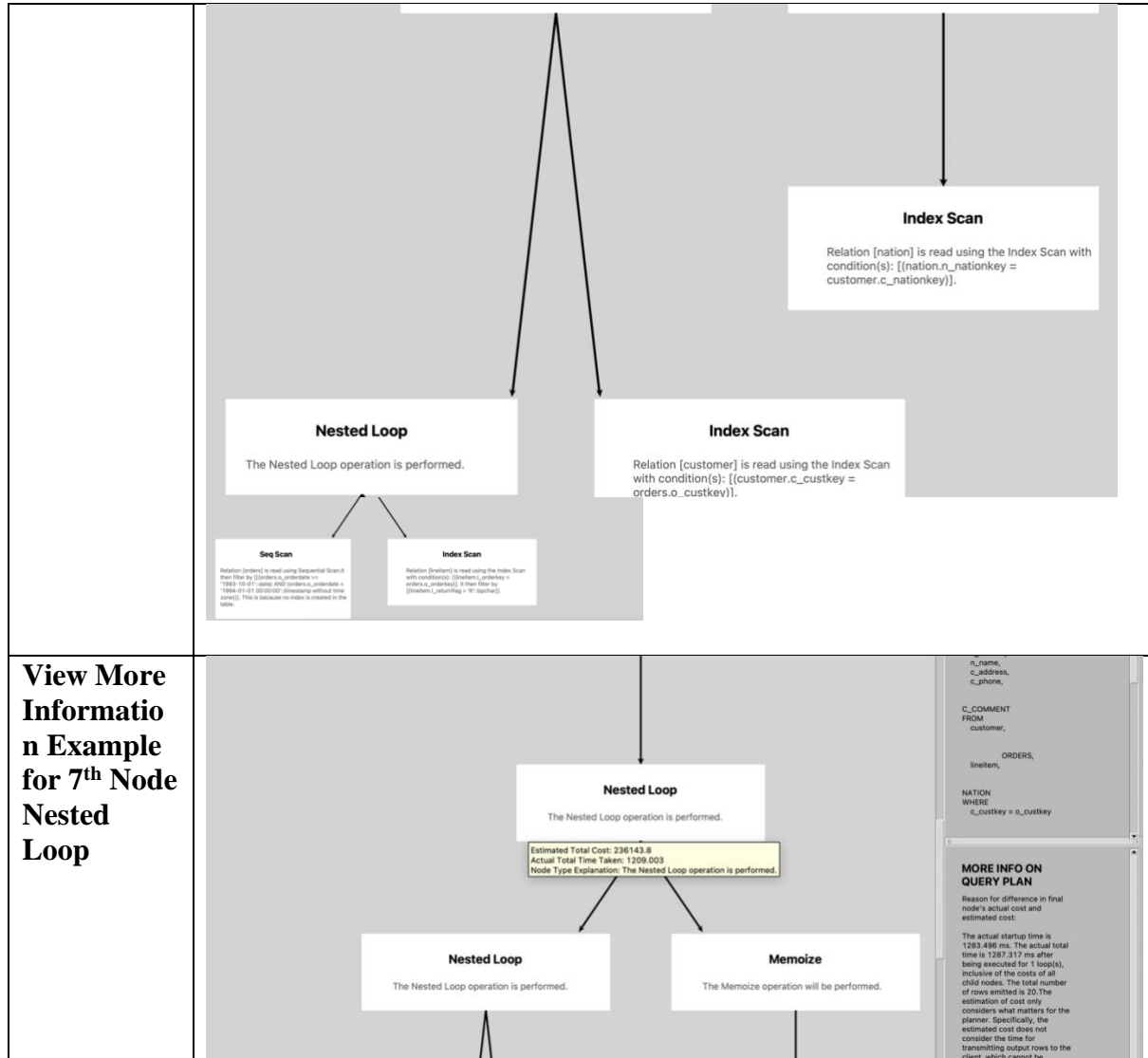
Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

**Full
Alternate
Query Plan
Tree**



Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

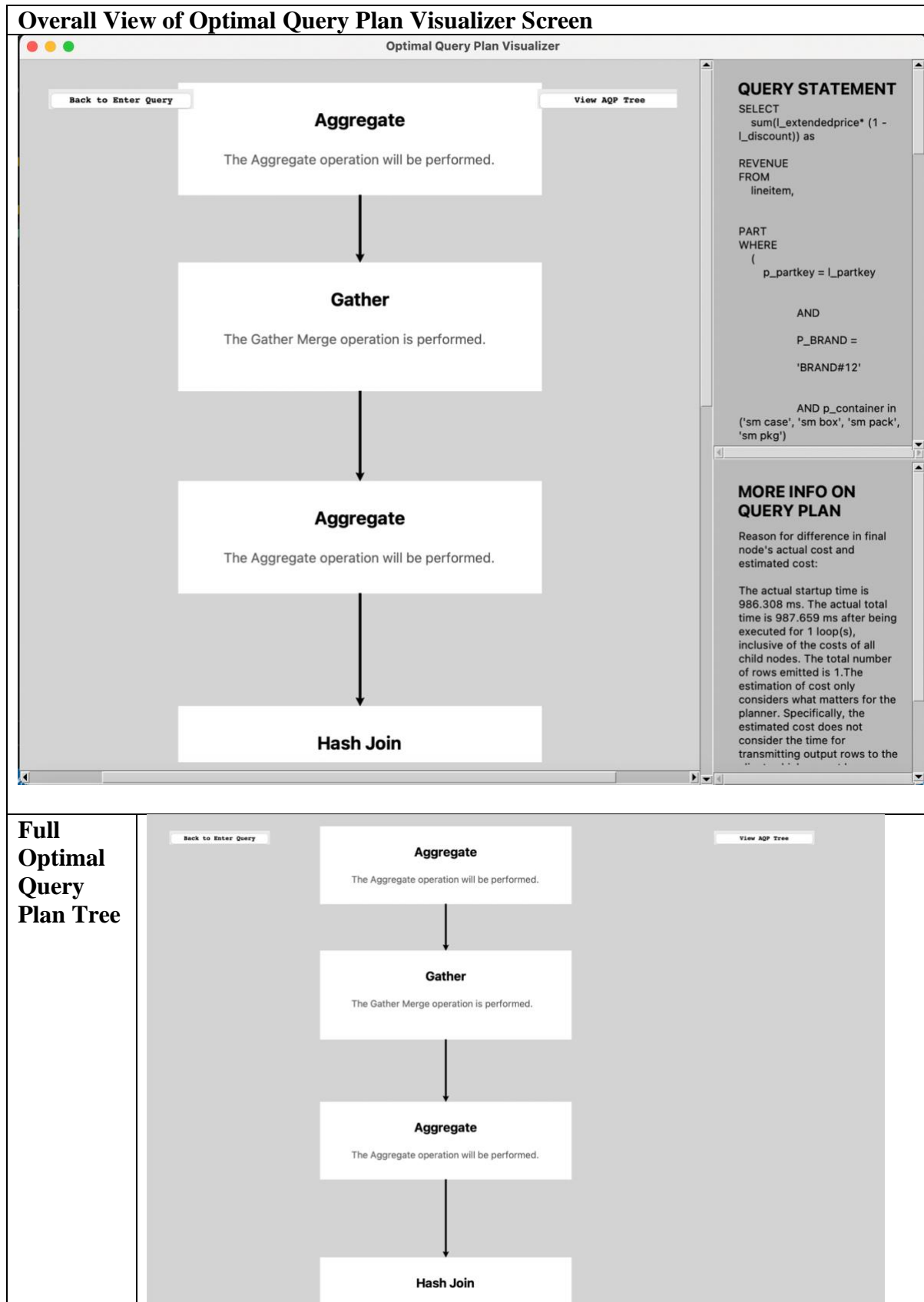


4.4 Query 4

Query:

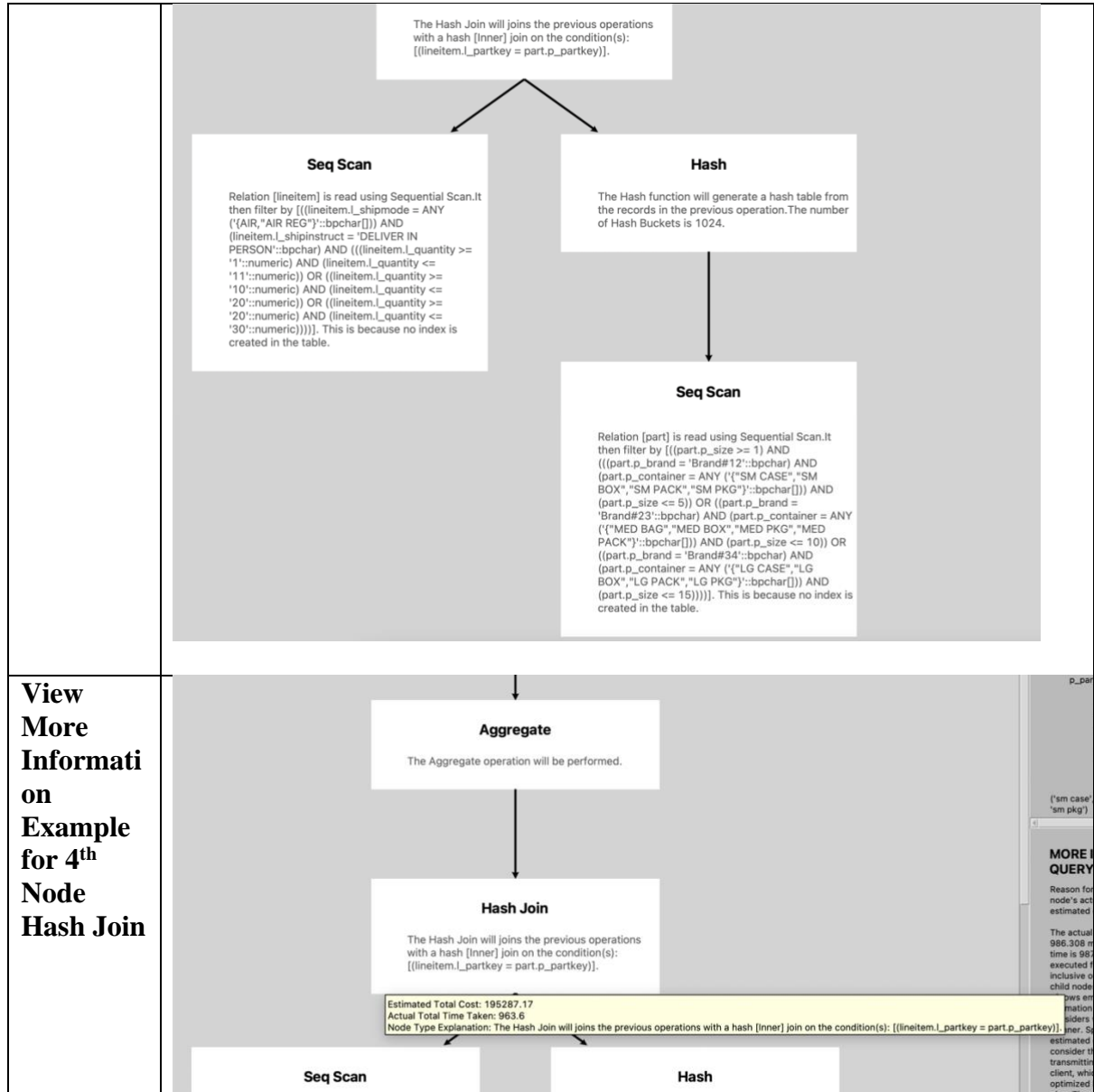
```
SELECT
  sum(l_extendedprice* (1 - l_discount)) as revenue
FROM
  lineitem,
  part
WHERE
  (
    p_partkey = l_partkey
    AND p_brand = 'Brand#12'
    AND p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
    AND l_quantity >= 1 AND l_quantity <= 1 + 10
    AND p_size between 1 AND 5
    AND l_shipmode in ('AIR', 'AIR REG')
    AND l_shipinstruct = 'DELIVER IN PERSON'
  )
  OR
  (
    p_partkey = l_partkey
    AND p_brand = 'Brand#23'
    AND p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
    AND l_quantity >= 10 AND l_quantity <= 10 + 10
    AND p_size between 1 AND 10
    AND l_shipmode in ('AIR', 'AIR REG')
    AND l_shipinstruct = 'DELIVER IN PERSON'
  )
  OR
  (
    p_partkey = l_partkey
    AND p_brand = 'Brand#34'
    AND p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
    AND l_quantity >= 20 AND l_quantity <= 20 + 10
    AND p_size between 1 AND 15
    AND l_shipmode in ('AIR', 'AIR REG')
    AND l_shipinstruct = 'DELIVER IN PERSON'
  )
)
```

Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

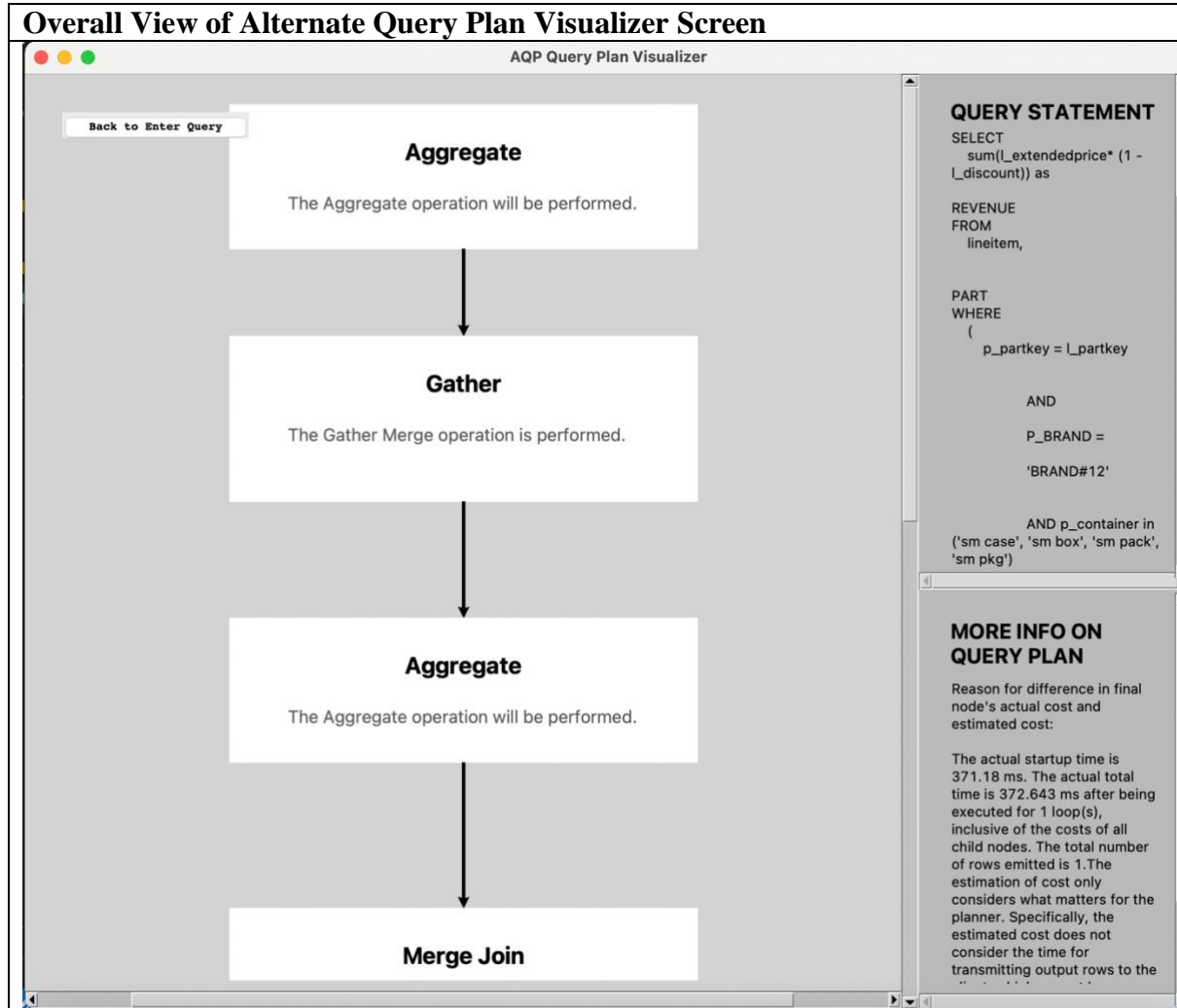


Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans



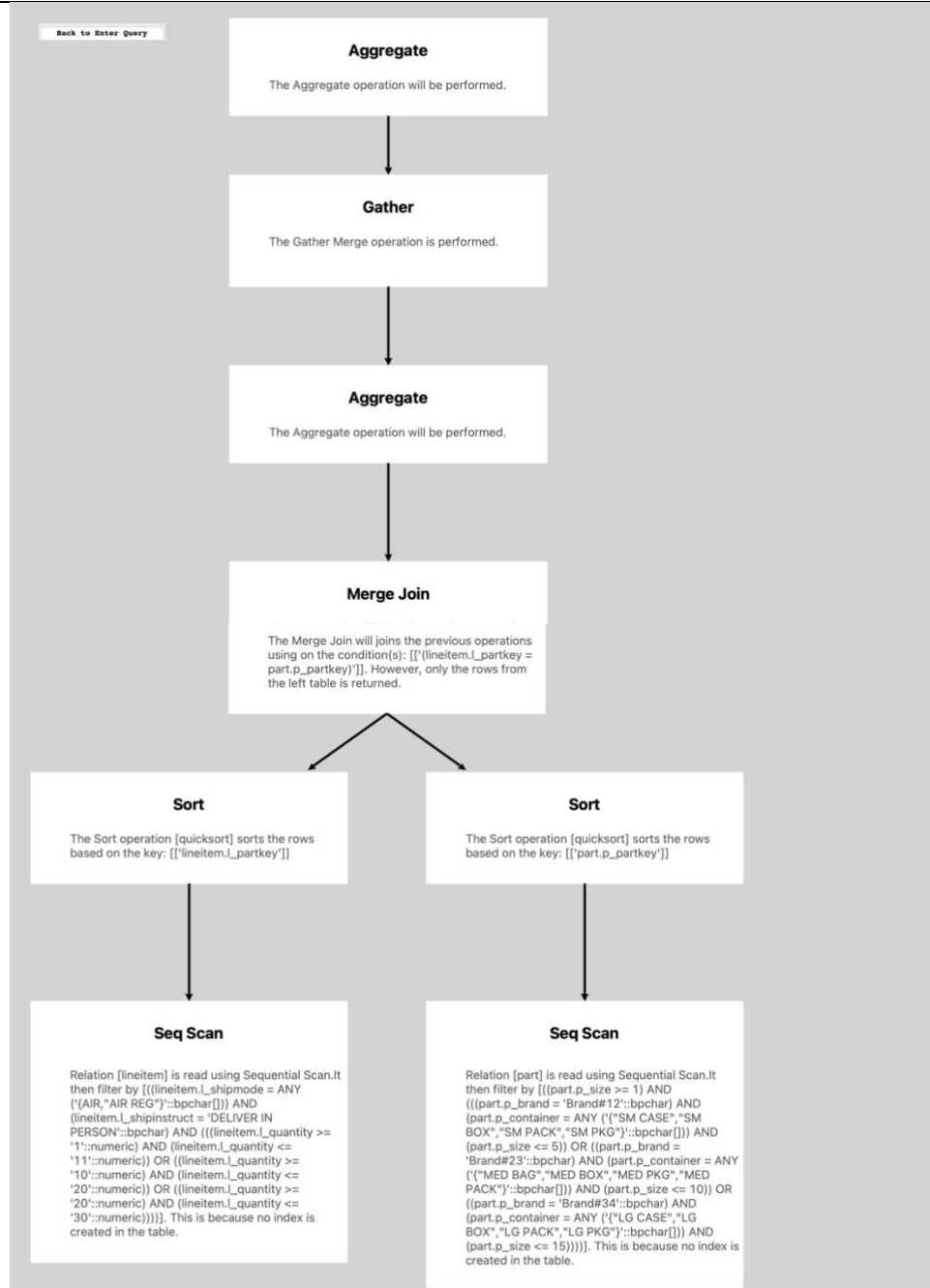
Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans



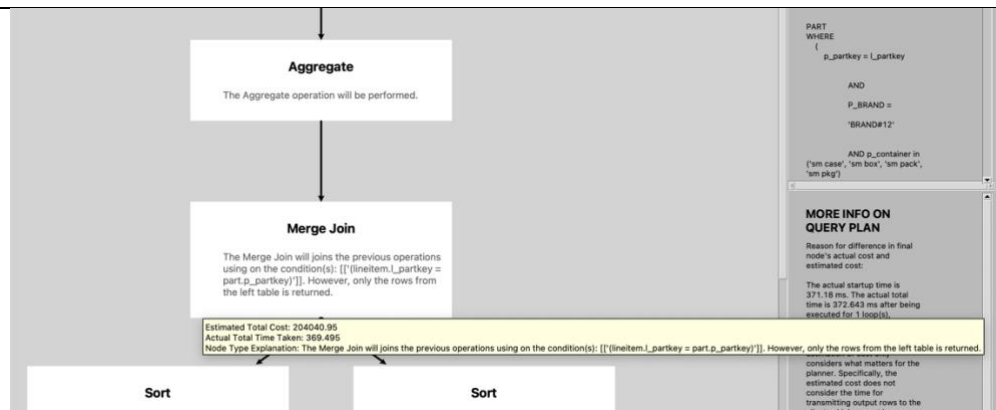
Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

Full Alternate Query Plan Tree



View More Information Example for 4th Node Merge Join



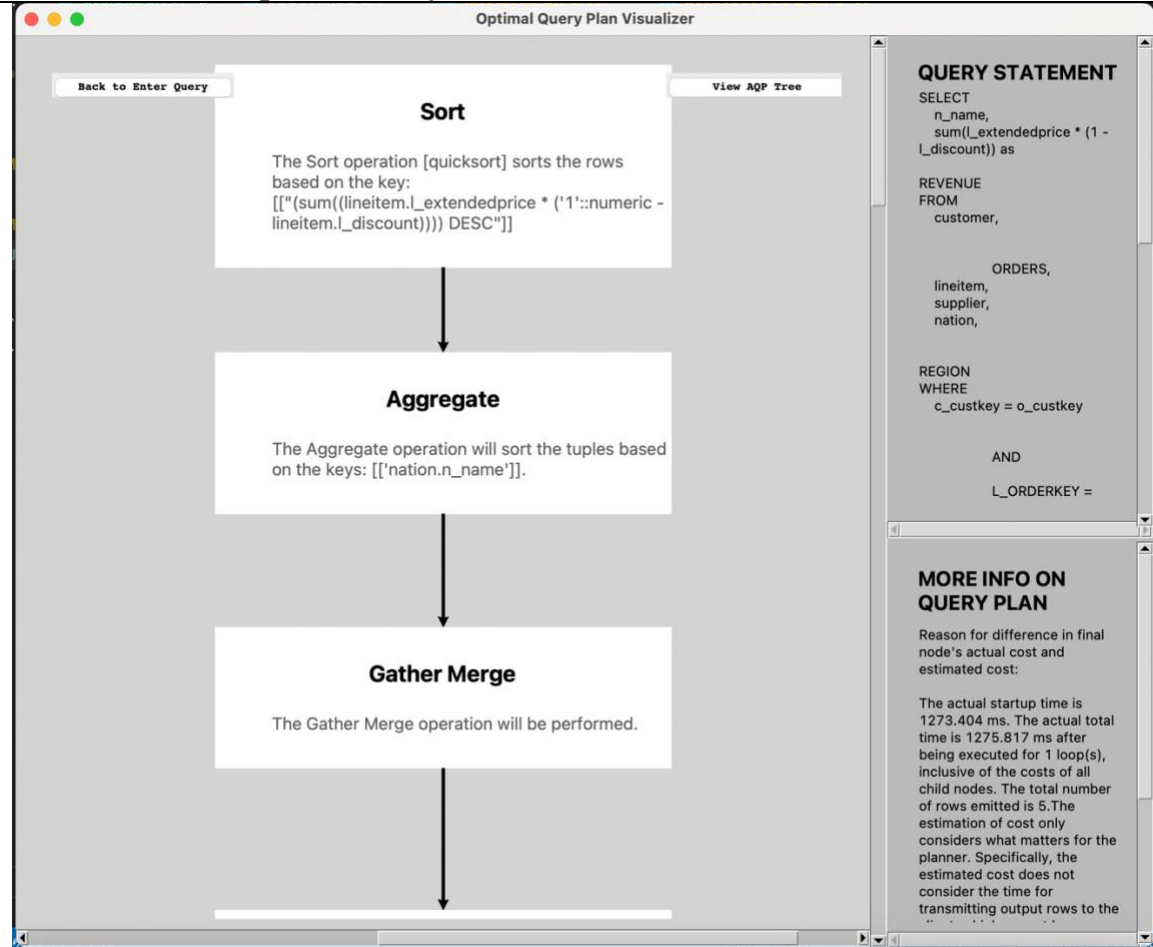
4.5 Query 5

Query:

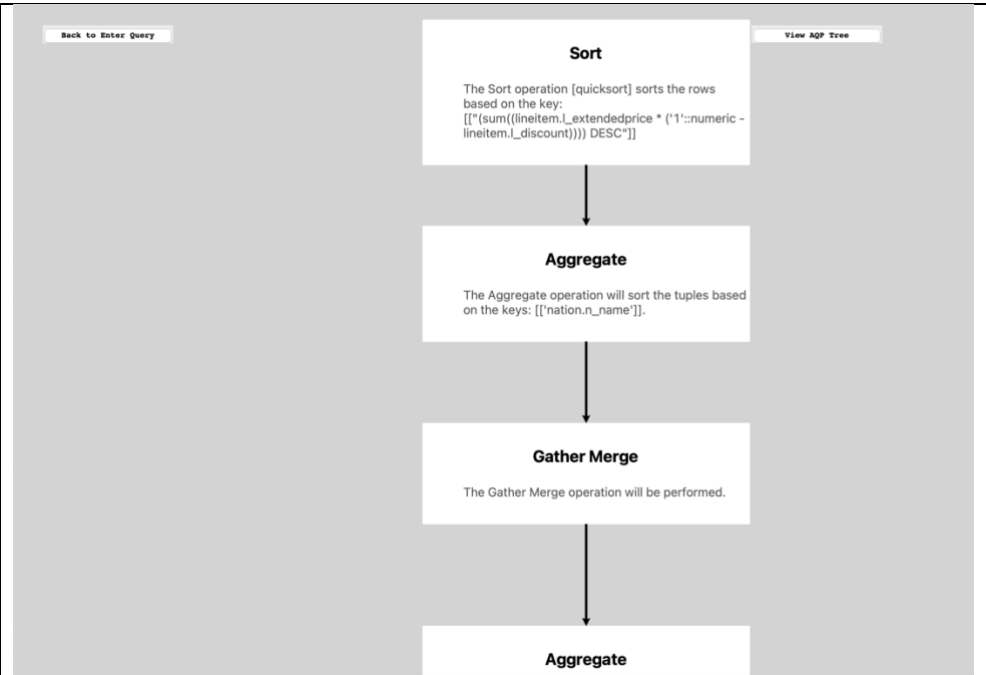
```
SELECT
  n_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue
FROM
  customer,
  orders,
  lineitem,
  supplier,
  nation,
  region
WHERE
  c_custkey = o_custkey
  AND l_orderkey = o_orderkey
  AND l_suppkey = s_suppkey
  AND c_nationkey = s_nationkey
  AND s_nationkey = n_nationkey
  AND n_regionkey = r_regionkey
  AND r_name = 'ASIA'
  AND o_orderdate >= date '1994-01-01'
  AND o_orderdate < date '1994-01-01' + interval '1' year
GROUP BY
  n_name
ORDER BY
  revenue desc
```

Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

Overall View of Optimal Query Plan Visualizer Screen

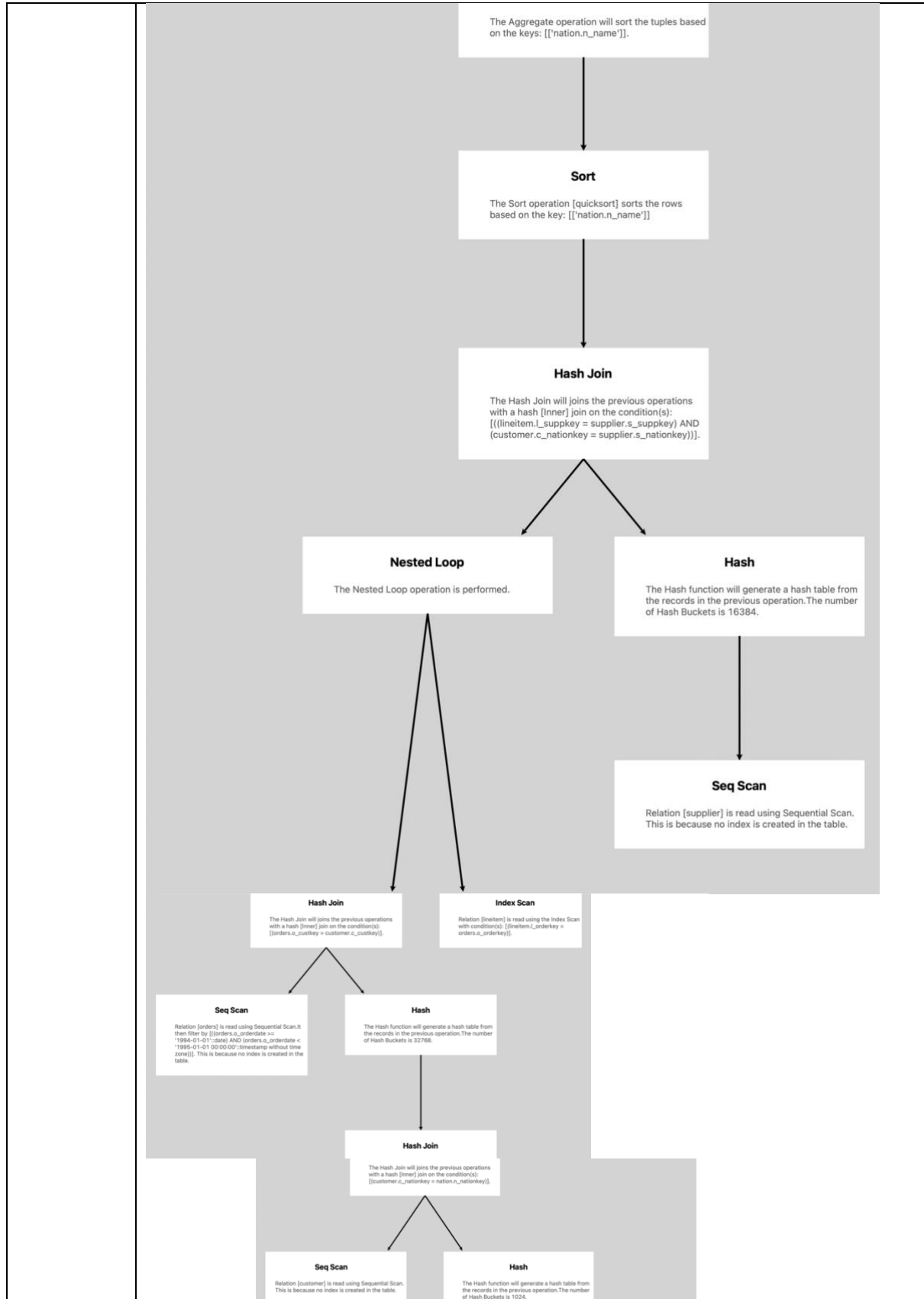


Full Optimal Query Plan Tree



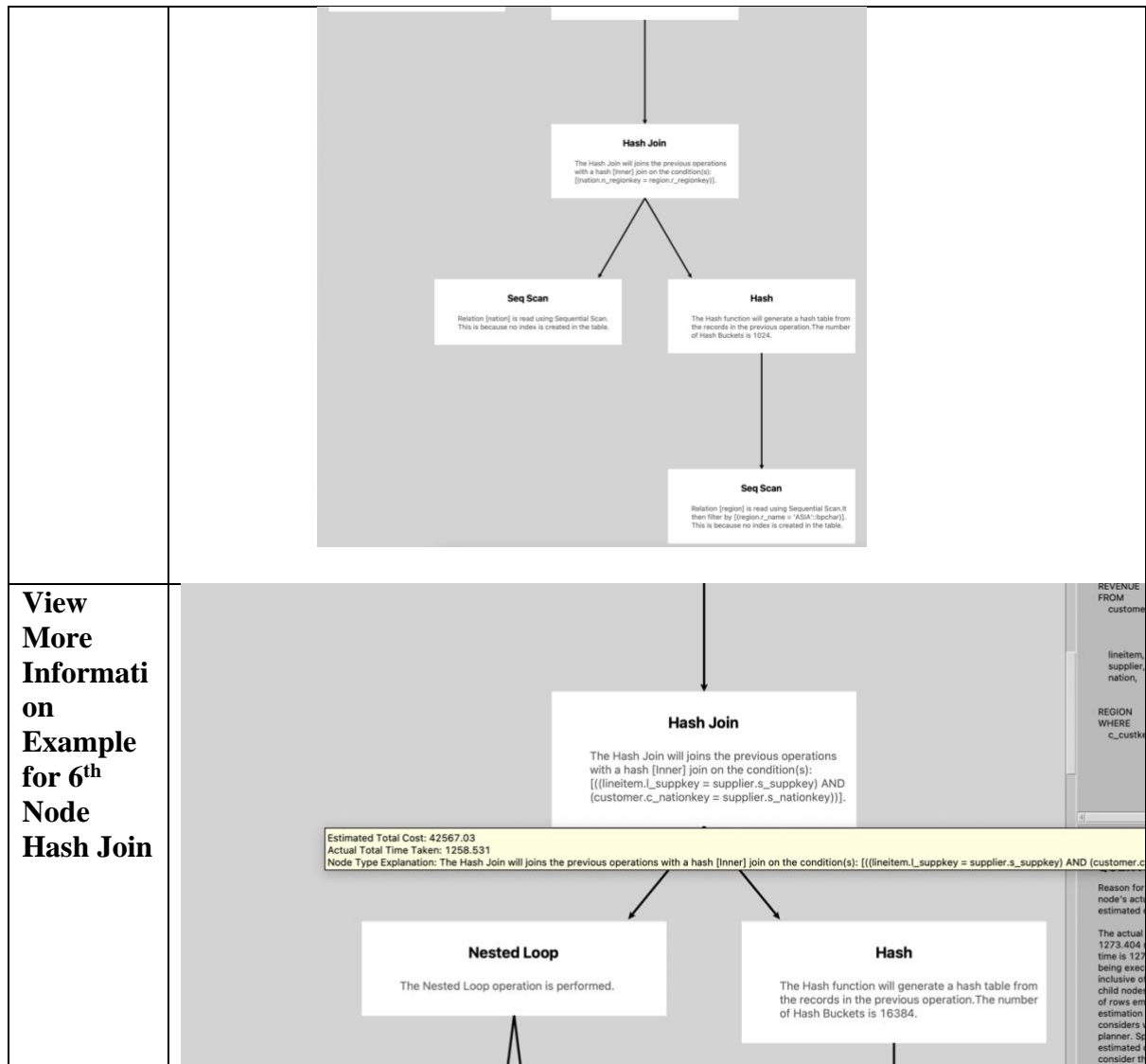
Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

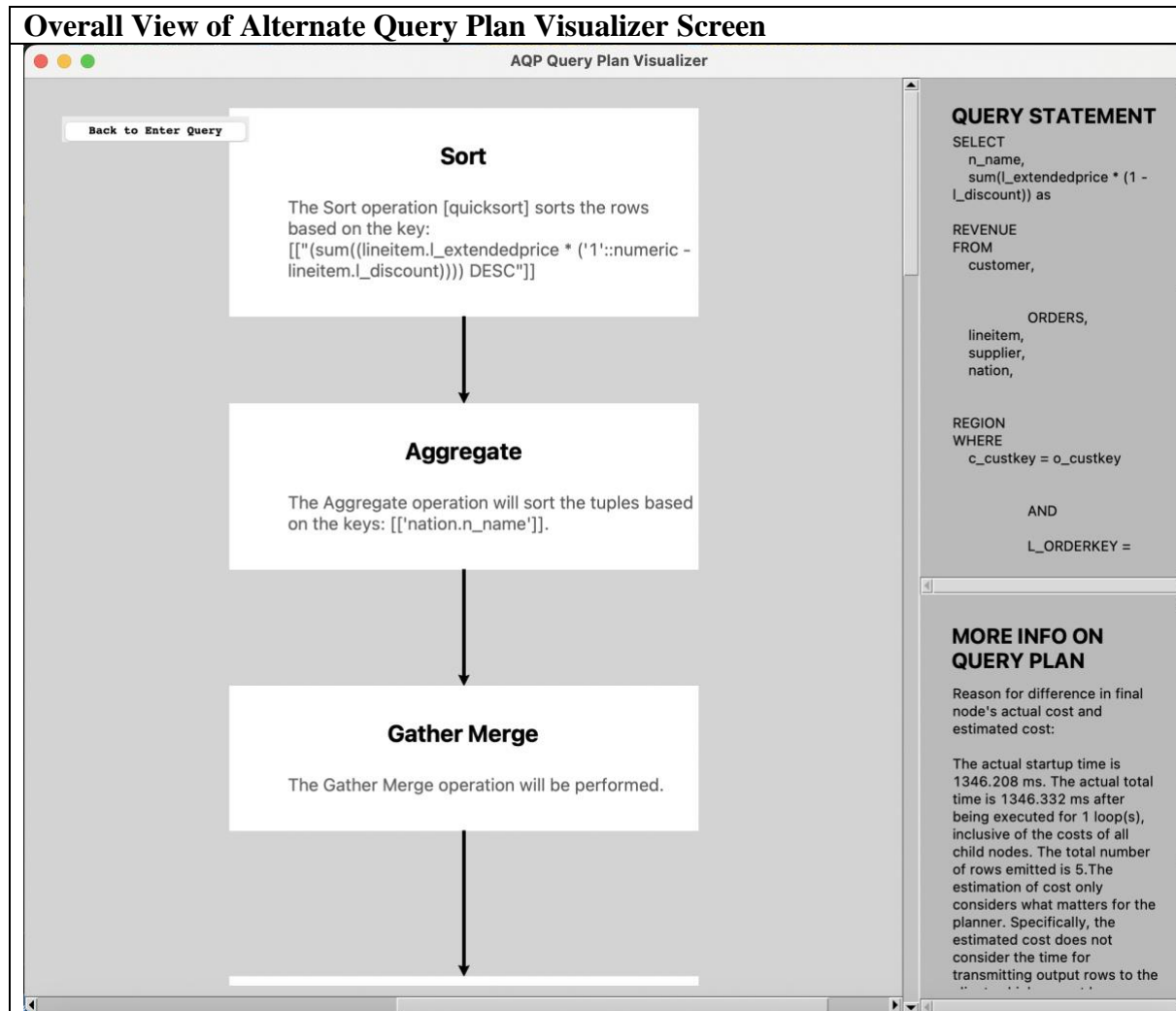


Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans

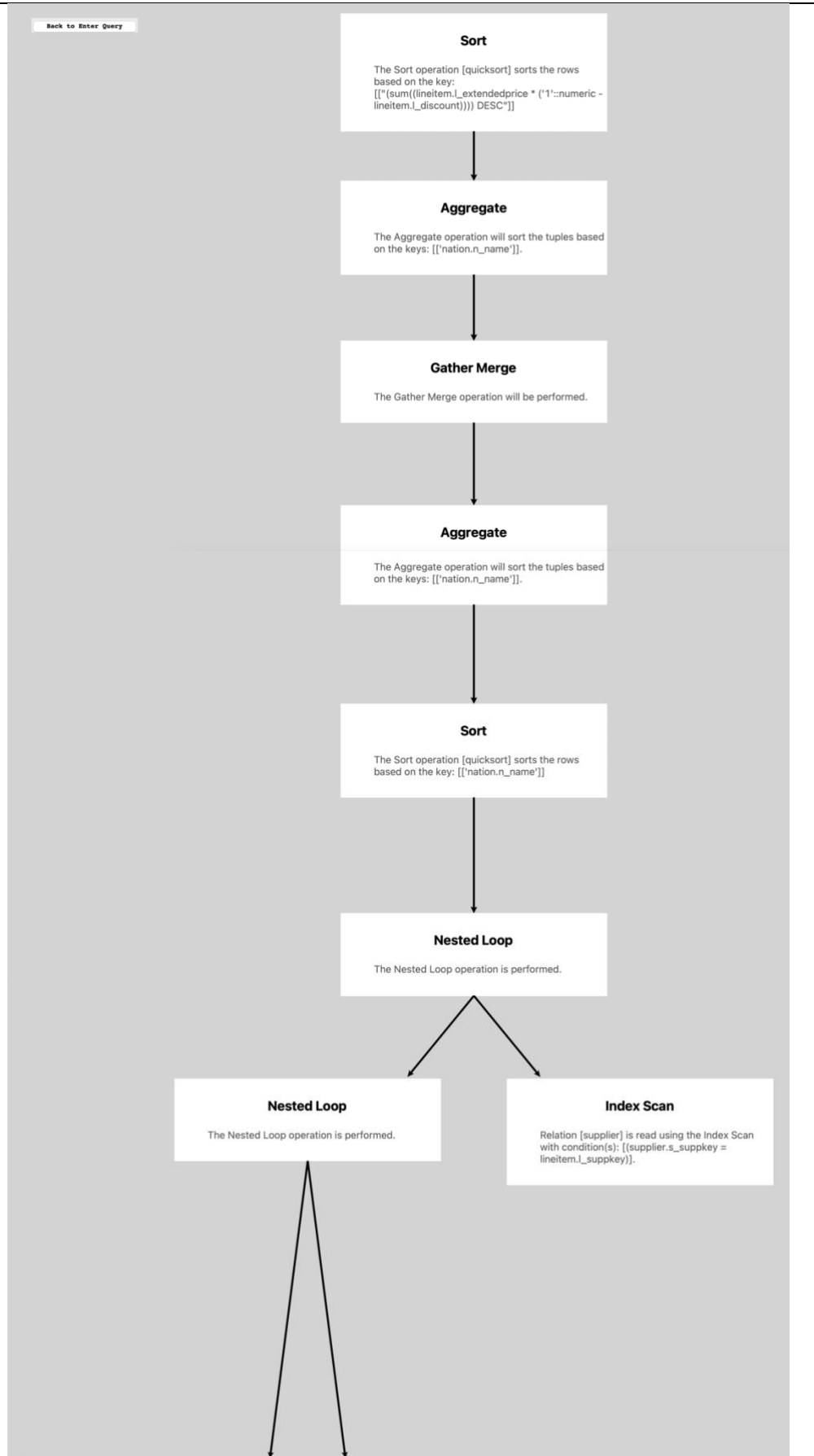


Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans



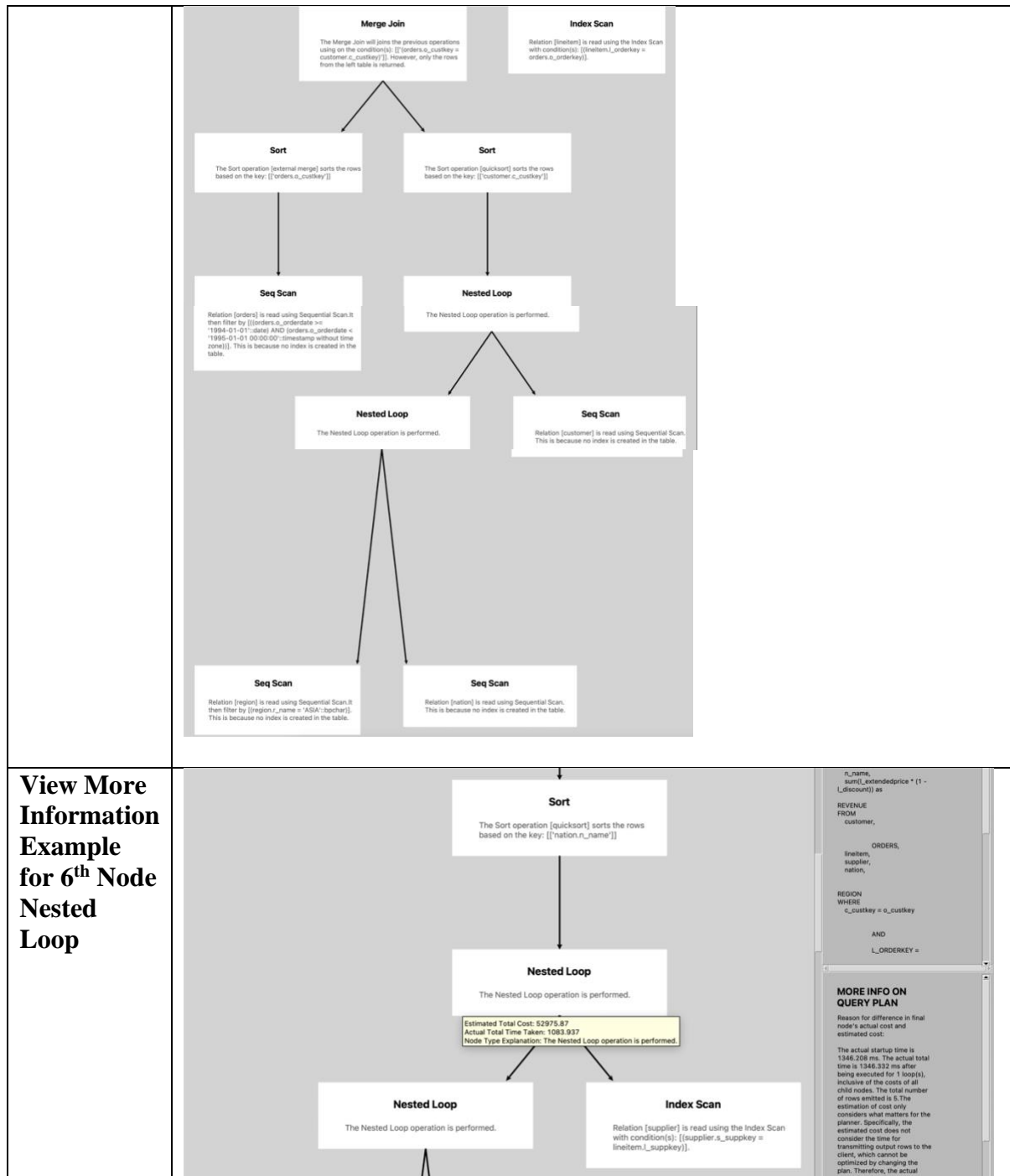
Project 2 22S1 CZ4031
Connecting SQL Query with Query Plans

**Full
Alternate
Query Plan
Tree**



Project 2 22S1 CZ4031

Connecting SQL Query with Query Plans



6 Limitations

6.1 Performance

For the current version of the software to retrieve and analyse the alternative query plan, a long processing time is required. The software needs to tweak the settings of the Postgres database system. The performance of the software is limited by the Postgres database system.

6.2 Security of Login Interface

Data encryption is implemented on user input. This includes login credentials such as passwords and SQL query statement. User inputs are vulnerable to data leaks, hence should not supply sensitive information to the software interface.

6.3 Limited Retrieval of AQPs

To retrieve AQPs, the settings of query processor are manually configured to force the execution of the query without using a certain condition (e.g., hash join, merge join, nested loop, index scan and sequential scan). While there are many different combinations of AQPs that can be generated, our algorithm only focuses on generating one type of AQP. Thus, our analysis and comparison are limited with respect to the AQP pool. However, forcing the query planner to disable multiple operation algorithms may compromise its performance significantly. Therefore, only one AQP is given in this software.

6.4 Limited Flexibility of Information Disclosure on Interface

This GUI can retrieve the underlying algorithm to annotate the SQL query in the form of an execution plan tree. The current GUI version is limited to the functionality of retrieving and displaying information from the QEP returned by PostgreSQL DBMS.

In addition, the current user interface for the “Query Plan Visualizer Screen” do not provide the functionality to hide and show information. Majority of the information is all displayed to the user all together such as “more info” and “query statement”. Further improvements may be achieved by using buttons to show and hide various information.

6.5 Consistency

This software is focused on query annotation purpose rather than query execution. Hence the consistency of output results is not guaranteed to be satisfactory for all possible SQL queries on various databases. Intensive testing is required at later stages to test for corner cases and explore other cases.

6.6 Support of PostgreSQL

To install and execute the GUI, user must pip install psycopg2 to support PostgreSQL server connection with python. This is not a convenient option for all users. In addition, as PostgreSQL is an open source, free DBMS, it lacks warranty for its security and liability of user inputs.

7 Contribution

Name	Contribution
ZHANG YINGHAO	Report structuring, writing, editing
GUPTA SUHANA	Report writing and UI implementation
MICHELLE LAM SU-ANN	Report writing and UI implementation
JIANG YUXIN	Report writing and annotation implementation
WANG QIANTENG	Report writing and annotation implementation

8 References

- [1] “PSYCOPG2,” *PyPI*. [Online]. Available: <https://pypi.org/project/psycpg2/>. [Accessed: 12-Nov-2022].
- [2] “Tkinter - Python interface to TCL/Tk,” *tkinter - Python interface to Tcl/Tk - Python 3.11.0 documentation*. [Online]. Available: <https://docs.python.org/3/library/tkinter.html>. [Accessed: 12-Nov-2022].
- [3] “PMW,” *PyPI*. [Online]. Available: <https://pypi.org/project/Pmw/>. [Accessed: 12-Nov-2022].