The JRS Graphical Editors of Logical and Physical Plans

A. Albano, R. Orsini, C. Valisena

Abstract

The functionalities of the JRS (*Java Relational System*) graphical editors of *logical* and *physical plans* are presented. The graphical editors are used to *define and execute* queries on a relational database represented by two kind of trees: A l*ogical plan* of relational algebra, and a *physical plan* that describes an algorithm to execute a query using the JRS physical operators, a relational DBMS developed in Java as a teaching tool at the University of Pisa, Department of Computer Science.[1]

1 The interactive JRS environment

JRS is a simple, single-user relational DBMS implemented in Java and designed for educational use, which supports a large subset of SQL-92,. The interactive JRS environment provides a unique environment with the possibility of practicing with the following three different methods in order to formulate relational database queries (Figure 1):

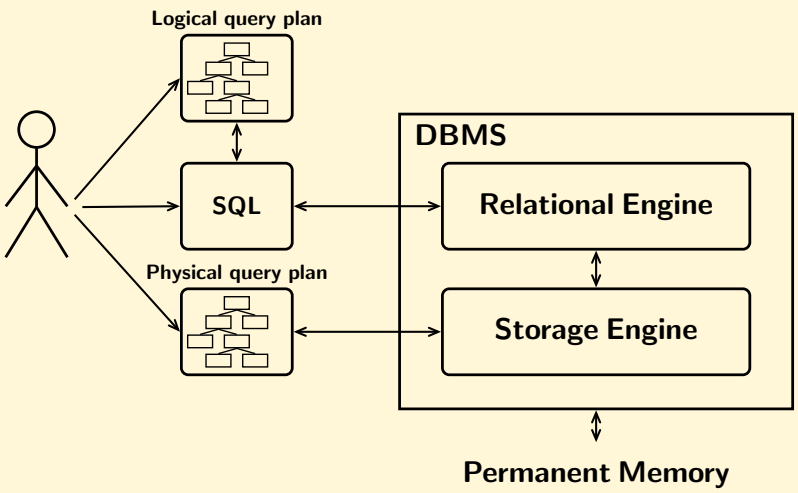


Fig. 1: The **JRS System**

1. A. Albano and C. Valisena, A System to Support Teaching and Learning Relational Database Query Languages and Query Processing, *SEBD 2011, Proceedings of the Nineteenth Italian Symposium on Advanced Database Systems*, pp. 215-225, Maratea, Italy, June 26-29, 2011.

- **SQL**, with the possibility to analyze the query plan produced by the optimizer. The optimizer by default generates **Left-Deep** access plans, and uses a **Greedy Search** optimization technique. A graphical interface allows the user to investigate the effect of other alternatives such as:
  - changing the access plans structure from **Left-Deep** to **Bushy**;
  - changing the optimization level from **Greedy Search** to **Iterative Full Search** or **Full Search**;
  - excluding the use of certain **Physical Operators** to generate access plans, and to exploit the impact of certain operators on the cost of a physical plan.
- **Relational algebra**, with the possibility to define and execute a graphical logical query plan step-by-step, and to see how it can be translated into SQL.
- **Physical algebra**, with the possibility to define and execute a graphical physical plan step-by-step.

2 Editor Window Areas

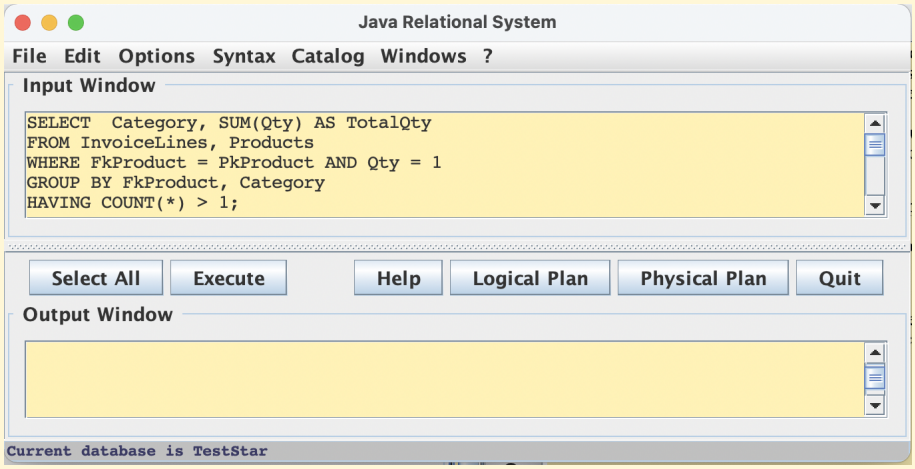


Fig. 2: The main window of the **JRS System**

The two graphical editors for physical and logical plans are activated with the buttons **Logical Plan** and **Physical Plan** from the main JRS window, with a database selected. They have a similar interface to define the nodes of a tree, the arcs between nodes, and to operate on a tree. The differences are in the types of nodes available and how plans are executed. The common characteristics are presented first, and then those specific to each type of tree.

When activated, a graphical editor displays a window divided into three main areas (Figure 3):

- **Control Panel**: An area that contains the buttons to add or remove a node, and to save, load or delete a plan.
- **Logical/Physical Plan Area**: An area to define a tree representation of a plan.
- **Query/Output**: An area that contains the result of a plan execution, which depends on the type of tree.

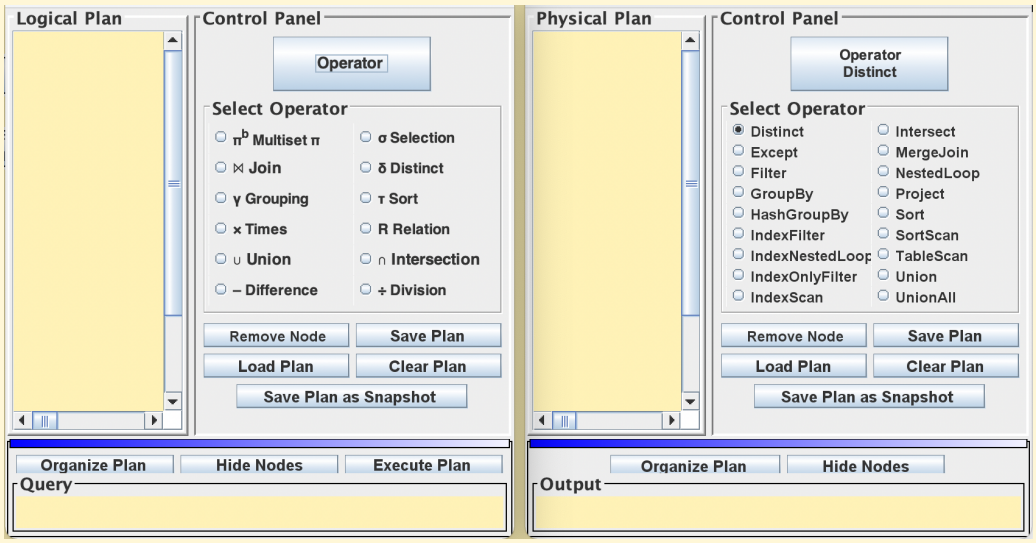


Fig. 3: The **Logical** and **Physical Plan Editors**

2.1 Control Panel

The area is divided into three parts:

- A frame **Select Operator** with the list of the operators that can be used in a plan. An operator is selected with a mouse click. The selection of an operator draws it in the plan area and change the name of the **Operator** button.
- A button **Operator** that shows the selected operator. Once an operator has been selected, clicking on **Operator** draws a new node of the same type in the plan area.
- An area with the following four buttons to operate on nodes and trees:
  - **Remove Node**, to remove a selected node, which is shown in yellow.
  - **Save Plan/Load Plan**, to save a plan on a file or to load a previously saved plane. Since each plan is defined on a JRS database, the loading of a previously saved plan is completed if the database currently in use is the same expected by the plan chosen.
  - **Clear Plan**, to clear the plan area.
  - **Save Plan as Snapshot**, to save a plan as an image in *jpg* format.

2.2 Logical/Physical Plan Area

Once a node has been added to the plan area, the following operations are allowed on it:

- **Selection**, with a click.
- **Moving**: click and hold the node, then drag it to its new location. To move an entire subtree, the same operation is performed on the root, while pressing **SHIFT**.
- **Adding an arc** between two nodes. When the mouse pointer is on a node, several **handles** appear on its edges. To add an arc, the pointer is moved, holding the left mouse button, from a handle on the starting node to the end node, where the mouse button is released.
- **Removing an arc**, by dragging one of its end points.
- **Showing a contextual menu**, with a **right mouse button** click on the node or on a Mac computer by using the **control button**.

2.3 The Result Area

The area is called the **Query** in the case of a logical plan, and **Output** in the case of a physical plane. The area shows the result of evaluating a plan or a subplan rooted in the node selected.

Above the result area there is a **blue bar**, to resize the area with the mouse, and the following buttons:

- **Hide/Show Nodes**, to hide the border of the nodes in order to view the plan as plain text.
- **Execute Plan**, to execute the **plan rooted in the node selected**. Use this feature to analyze the result of a complex plan execution step-by-step, starting from the leaves. The result is printed in the output window and it is shown also in a separate window provided by JRS.
- **Organize Plan**, to redraw the plan tree automatically.
- **Show SQL**, to see in the **Query** area the translation of a logical plan into SQL, which is then used to execute it.

3 Contextual Menu of the Logical Plans Editor

Relational algebra is based on a small number of operators which take one or two relations as operands to yield a relation as result. A query is just an expression involving these operators. The **Logical Plan Editor** allows the definition of query as an expression tree of relational algebra operators.

Any subtree, with relations as leaves, can then be executed to see the results.

Each node of a logical plan has a different **contextual menu**, that is dynamically created taking into account the attributes of the operands and schema of the database. Each choice of parameters implies an update of the node label to reflect the choices made. In creating a tree, the node parameters must be specified from the leaves to the root: **A contextual menu becomes active only when the node has all the required operands specified**.

A brief description follows of the contextual menu of each operator.

- **Relational Operator** (R)

  The menu has the following options:

  - **Relation**, to select a relation among those defined in the current database, shown in a submenu. **Each database table must be defined with at least one key.**
  - **Tuple Variable**, to define an optional variable that stands for a record of the selected relation.

- **Selection Operator** (σ)

  The menu has the following options:

  - **Condition**, to define a simple selection condition using a submenu.
  - **Enter Condition**, to define a complex selection condition.

  Only one of these two options can be used.

- **Grouping Operator** (γ)

  The menu has the following options:

  - **Groupings**, to select the grouping attributes.
  - **Aggregations**, to select one or more aggregation functions, with or without **Distinct**.

    Each aggregate function must be renamed with the **AS** *operator.*
  - **AS**, to optionally rename grouping attributes.

- **Product Operator** ($\times$)

  This node has no contextual menu.

- **Join Operator** ($\bowtie$)

  The menu has the following options:

  - The entries of the menu σ to define a general join condition.
  - **Natural Join**, to define automatically the join condition using the common attributes of the two operands.

  - **Equi Join**, to define automatically the join condition using the attributes of the primary key and the foreign key of the two operands.

  Only one of these three options can be used.

- **Division Operator** ($\div$)

  This node has no contextual menu.

  The operator has two operand: A relation with two attributes (e.g. $R(A, B)$), without duplicates, and a relation with only the second attribute of the first operand and of the same type (e.g. $S(B)$), without duplicates.

  The result is a relation with the attribute A that consists of the values of A in R that *appear together with all the values of B in S.*

- **Set Operators** ($\cup, \cap, -$)

  These nodes have no contextual menu.

  The operands must be relations of the same type, and they must satisfy the following rules:

  - Both the relations have the same number of attributes.
  - The names of the attributes are the same, and in the same order, in both the relations.
  - Attributes with the same name in both relations have the same type.

  The nodes of a logical plan constructed with the previous operators of the relational algebra have as result a set of records, assuming that the *relations of the leaves of the tree are defined in SQL JRS with at least one key.*

  Instead, the result of the following operator π^b is a *multiset of records*, and it is provided to define logical tree of common SQL queries.

- **Multiset Projection Operator** (π^b)

  The menu has the following options:

  - **Attributes**, to select some of the attributes of a relation, shown in a submenu.
  - **Enter Expression**, to enter expressions instead of attributes.
  - **AS**, to optionally rename attributes or expressions.

  The result of a π^b is a *multiset of records.*

- **Distinct Operator** (δ)

  This node has no contextual menu.
  The operator *must have as operand a π^b*, and eliminates duplicate records from the operand result, i.e. *it is used to turn a multiset into a set.*

- **Sorting Operator** (τ)

  The menu has the following options:

  - **Attributes**, to select the sorting attributes.
  - **Direction**, to select between the ascending or descending order.

The graphical editor allows the use of the operators π^b, δ or τ as the root of an expression tree only. When τ is used, it must be the root because it can be used only to turn the operand result, a set or a multiset, into an ordered set or an ordered multiset.

3.1 Node's Information

A double click with the left mouse button on a node will display a box with the following information:

- **Operator**: The node operation.
- **Table**: The relation name of a leaf node.
- **Condition**: The condition for select and join operators.
- **Result Type**: The operator result type is a set denoted $\{(A_iT_i, ..., A_nT_n)\}$, or a multiset, or a sorted set denoted $\{\{(A_iT_i, ..., A_nT_n)\}\}$.
- **Order**: The order of records in the query result.

4 Contextual Menu of the Physical Plans Editor

A physical plan is an algorithm for executing a query using different evaluation methods, called *physical operators*. Often the physical operators are particular implementations of the operators of relational algebra. They differ in their basic strategy and have significantly different costs. However, there are also physical operators for other tasks that do not involve an operator of relational algebra. The result of the evaluation of a physical plan is in general a multiset of records, which is the answer of the query. The **Physical Plans Editor** allows the definition of query as a tree of physical operators.

As for the logical trees, any physical subtree, with operators on relations as leaves, can be executed to see its results.

Each node of a physical tree has a different contextual menu, which is dynamically created taking into account the attributes of the operands and schema of the database. Each choice of a parameter implies an update of the node label to reflect the choice made. In creating a tree, the node parameters must be specified from the leaves to the root: A contextual menu becomes active only when a node has all the required operands specified.

A brief description follows of the contextual menu of each operator.

4.1 Table Operators

- **TableScan**

  The menu has the following options:
  - **Table**, to select a table among those of the current data base.
  - **Tuple Variable**, to define an optional record variable.

- **IndexScan**

  The menu has the following options:
  - The entries of the menu **TableScan**.
  - **Index**, to select an index among those defined on the selected table.

- **SortScan**

  The menu has the following options:
  - The entries of the menu **TableScan**.
  - The entries of the following menu **Sort**.

4.2 Sort Operator

- **Sort**

  The menu has the following options:
  - **Attributes**, to select the sorting attributes.
  - **ORDER**, to define the sorting criteria.

4.3 Projection Operator

- **Project**

  The menu has the following options:

- **Attributes**, to select some of the attributes of a relation, shown in a submenu.
- **Enter Expression**, to enter expressions instead of attributes.
- **AS**, to optionally rename attributes or expressions.

The result of a **Project** operator is a *multiset*.

- **Distinct**

  This node has no contextual menu.
  The operator eliminates duplicate records from the operand result, that must be ordered. Since the operator requires the input records fully sorted, it checks if the previous operator is Sort or SortScan, otherwise notifies the user of a possible problem.

4.4 Selection Operator

- **Filter**

  The menu has the following options:
  - **Condition**, to define a simple selection condition using a submenu.
  - **Enter Condition**, to define a complex selection condition.

  Only one of these two options can be used.

- **IndexFilter**

  The menu has the following options:
  - **Table**, to select an indexed table among those of the current database.
  - **Tuple Variable**, to optionally define a record variabile.
  - **Index**, to select an index among those defined on the table.
  - **Condition**, to define a simple selection condition on the index attributes using a submenu.
  - **Enter Condition**, to define a complex selection condition on the index attributes.
  - **Join Condition** ψ, to specify that the join condition must be automatically generated because the operator will be used as internal operand of an **IndexNestedLoop** operator.

  Only one of the last three options can be used.

- **IndexOnlyFilter**

  The menu has the following options:
  - **Index, Condition, Enter Condition, Join Condition** ψ as for the menu of **IndexFilter**.
  - **Project**, to select the index attributes of the result if the index is multi-attribute.

4.5 Grouping Operator

- **GroupBy**

  The menu has the following options:
  - **Groupings**, to select the grouping attributes.
  - **Aggregations**, to optionally select one or more aggregation functions, with or without **Distinct**.
    Each aggregate function must be renamed with the **AS** *operator.*
  - **AS**, to optionally rename grouping attributes.

The operator requires that the operand records must be sorted on the grouping attributes. For this reason it checks the operand, and if it is not a **Sort** or **SortScan** it notifies the user of a possible problem.

- **HashGroupBy**

  The menu has the following options:
  - The entries of the menu **GroupBy**, but note that *the operator does not require that the operand data must be sorted on the grouping attributes.*

4.6 Join Operators

- **NestedLoop**

  The menu has the following options:
  - **Condition**, to define a simple join condition using a submenu.
  - **Enter Condition**, to define a general join condition on the index attributes.
  - **Equi Join**, to define automatically the join condition using the attributes of the primary key and the foreign key of the two operands.

  Only one of these three options can be used.

- **IndexNestedLoop**

  The menu has the options of **NestedLoop**, with the constraint that the internal operand must be a node **IndexFilter** or **IndexOnlyFilter**, or a **Filter** applied to a **IndexFilter** or to a **IndexOnlyFilter**.

  An index filter of the internal operand must be defined with the option **Join Condition** ψ, *and* ψ *is then automatically rewritten using the* **Equi Join** *option of the* **IndexNestedLoop**.

- **MergeJoin**

  The menu has the options of **NestedLoop**, with the constraint that the result of the operands are ordered on the join attributes, a key for the external operand. Again a test is made for node children and if they are not **Sort** or **SortScan** the user is notified of possible problems.

4.7 Set Operators

- **Union, Union All, Intersect, Except**

  These node have no contextual menu.

  The operands of result of the operators **Union**, **Intersect**, **Except**, must be *sorted multiset of the same type*, that is they must satisfy the following rules:
  - Both the multisets have elements with the same number of attributes.
  - The names of the attributes are the same, and in the same order, in both the relations.
  - Attributes with the same name in both multisets have the same type.

  Again a test is made for node children and if they are not **Sort** or **SortScan** the user is notified of possible problems.

  The result of the operators **Union**, **Intersect**, **Except** is a ordered set, while the result of **Union All** is a multiset.

4.8 Node's Information

A double click with the left mouse button on a node will display a box with the relevant subset of the following information:

- **Operator**: The node operation.
- **Table**: The table name of a leaf node.
- **Index**: The index name, if the operator uses one.
- **Attributes**: The attributes of the index in use.
- **Condition**: The condition for select and join operators.
- **Result Type**: The operator result type, a multiset denoted $\{\{(A_iT_i, ..., A_nT_n)\}\}$.
- **Order**: The order of records in the query result.
- **Result Size**: The estimate number of records of the plan result.
- **Cost**: The estimate the number of pages read from or written to disk to produce the result.[2]

2. The goal of the cost and result size estimations is not to predict the exact values, but they are the estimations used by the optimizer to select a query plan using the information available in the DBMS catalog.

5 Examples of using the Graphical Plan Editor

The examples will be given using the relational database schema in Figure 4 and the following query to retrieve the category of products sold singly more than once, and the total quantity sold:

```
SELECT      Category, SUM(Qty) AS TotalQty
FROM        InvoiceLines, Products
WHERE       FkProduct = PkProduct AND Qty = 1
GROUP BY    FkProduct, Category
HAVING      COUNT(*) > 1;
```

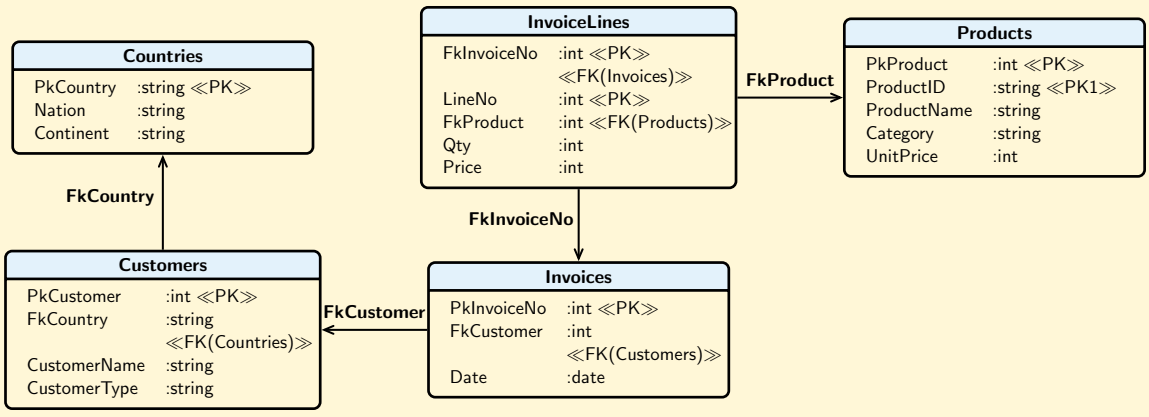
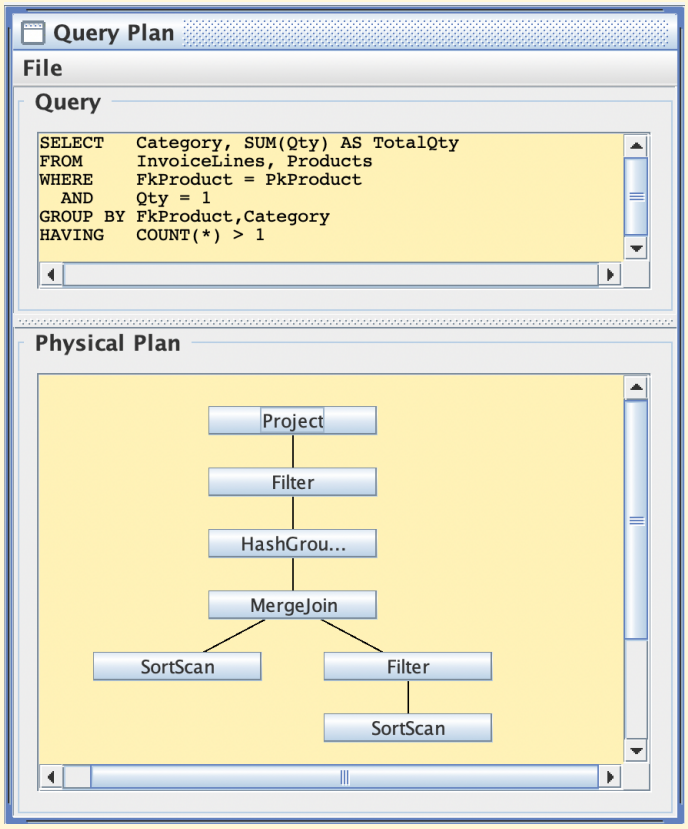


Fig. 4: The relational database schema

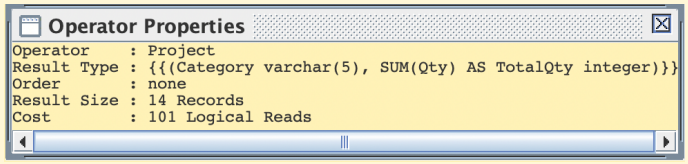
Let us first select the option **Show Physical Plan**, located in the **Options** menu, and then execute the query. We then get the query result and the *physical query plan* generated by the cost-based JRS query optimizer to execute the query. The physical query plan is represented by a tree of physical operators in a separate window (Figure 5a). The JRS *cost-based query optimizer* estimates the costs of alternative query plans and chooses an efficient final plan. This is done using the metadata available on the database, such as the size of each relation, the existence of certain indexes, and the number of different values for an attribute when an index is present.

Clicking on a node of the plan produces a **Physical Operator Properties** window with information regarding the operator involved, the estimated number of records produced by the operator, and the estimated cost of the operation (Figure 5b).

As usually happens in relational DBMSs, the standard way to evaluate a query with **Group By** is to first retrieve the records required by the operator, and then to execute it with the physical operator **HashGroupBy** in order to produce the final result.



a)



b)

Fig. 5: The Optimized **Physical Query Plan**

A Logical Plan

Let us define the query using the relational algebra, as shown in Figure 6.

By clicking on **Hide Nodes** the logical plan is shown in the traditional form of an algebraic *expression tree*.

Double clicking on a node opens a **Logical Plan Node Information** window with information about the logical operator and the result type.

Any plan node can be selected by clicking. Users can then proceed as follows:

- Clicking on **Execute Plan** produces the query result in the query result window.
- Clicking on **Show SQL** displays in the query result window the SQL query generated by the **Logical Plan Editor**, for *the subtree of the selected plan node*. For example, the SQL query generated, when the selected node is the plan root, is the same shown in Figure 5a.
  The SQL query generated for a logical plan is generally not a single **SELECT**, but a **SELECT** that uses temporary views, because JRS does not allow the use of a subquery in a **FROM** clause. Moreover, in order to avoid the use of views, in the current implementation, the algorithm for generating the SQL query to execute a logical plan does not exploit rewriting rules.

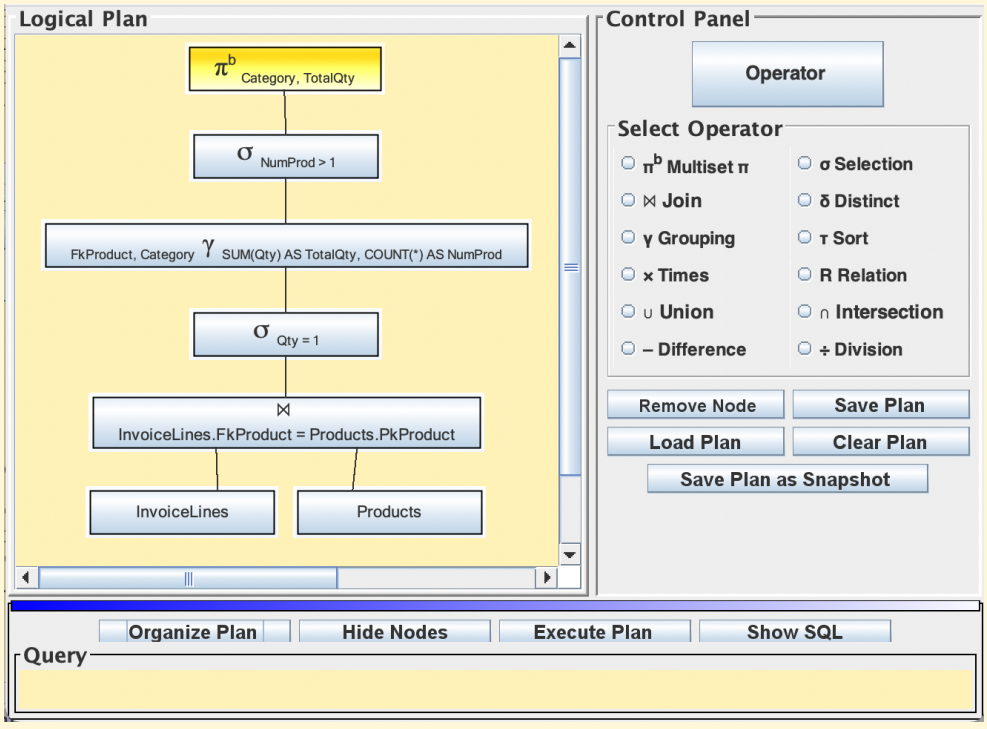


Fig. 6: A **Logical Query Plan**

Another Logical Plan

Let us try a different logical plan on the basis of the following result:

Proposition 5.1: *Let $\alpha(X)$ be the set of columns in X and $R \overset{\bowtie}{}_{C_j} S$ an equi-join using the primary key p_k of S and the foreign key f_k of R. R has the* invariant grouping *property*

$$ {}_A\gamma_F(R \overset{\bowtie}{}_{C_j} S) \equiv \pi^b_{A \cup F}(({}_{A \cup \alpha(C_j) - \alpha(S)}\gamma_F(R)) \overset{\bowtie}{}_{C_j} S) \tag{1}$$

if the following conditions are true:
1. *$A \rightarrow f_k$, with A the grouping columns in $R \overset{\bowtie}{}_{C_j} S$.*
2. *Each aggregate function in F uses only columns from R.*

This property of doing the group-by before a join is called *invariant grouping* since the operator can be brought forward by modifying the grouping attributes only, but the transformation may need an additional projection in order to produce the final result. In general, with a big table R the performance of a join query with grouping and aggregation is improved by doing the group-by before the join.

If there are selections on R, the operator γ is done before a join on the selections on R (Figure 7).

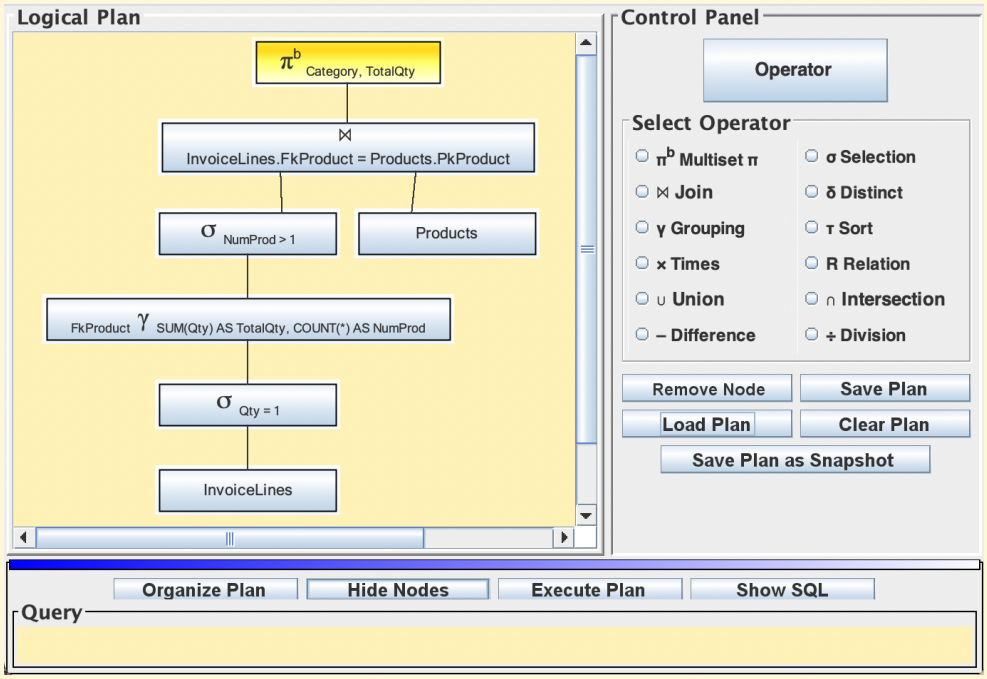


Fig. 7: Another **Logical Query Plan**

Since the query optimizer does not consider the possibility of doing the group-by before the join, the **Logical Plan Editor** generates the following SQL code to execute the logical plan:

```
WITH      LTV1 AS
(
SELECT    FkProduct, SUM(Qty) AS TotalQty, COUNT(*) AS NumProd
FROM      InvoiceLines
WHERE     Qty = 1
GROUP BY  FkProduct
HAVING    COUNT(*) > 1
)

SELECT    Category, TotalQty
FROM      LTV1, Products
WHERE     FkProduct = PkProduct;
```

In this case, the generated query uses a view as a left operand of the join, thus forcing the optimizer to generate two separate plans, one for the view and another for query. However it is interesting to check if the query generated by a logical plan that brings forward the group-by before a join is more efficient.

A Physical Plan

Let us now design a physical plan that exploits carrying out the group-by before a join with the operator **GroupBy**, and uses the operator **IndexNestedLoop** with an index on the **Products** primary key (Figure 8).

Clicking on **Hide Nodes** displays the physical plan in the traditional form of an *iterators tree*.

A double click on a node of the plan displays information about the operator involved, the estimated number of rows produced by the operator, and the estimated cost of the operation (Figure 9, Figure 10, Figure 11).

As it happens with a logical plan, any physical plan node can be clicked on in order to execute the subtree with the selected node as root node.

Finally, in Figure 12 is shown the physical plan with the use of the operator **HashGroupBy**, which produces the same result, ordered differently, and, due to the small size of the database tables, has the same costs for the physical operators.

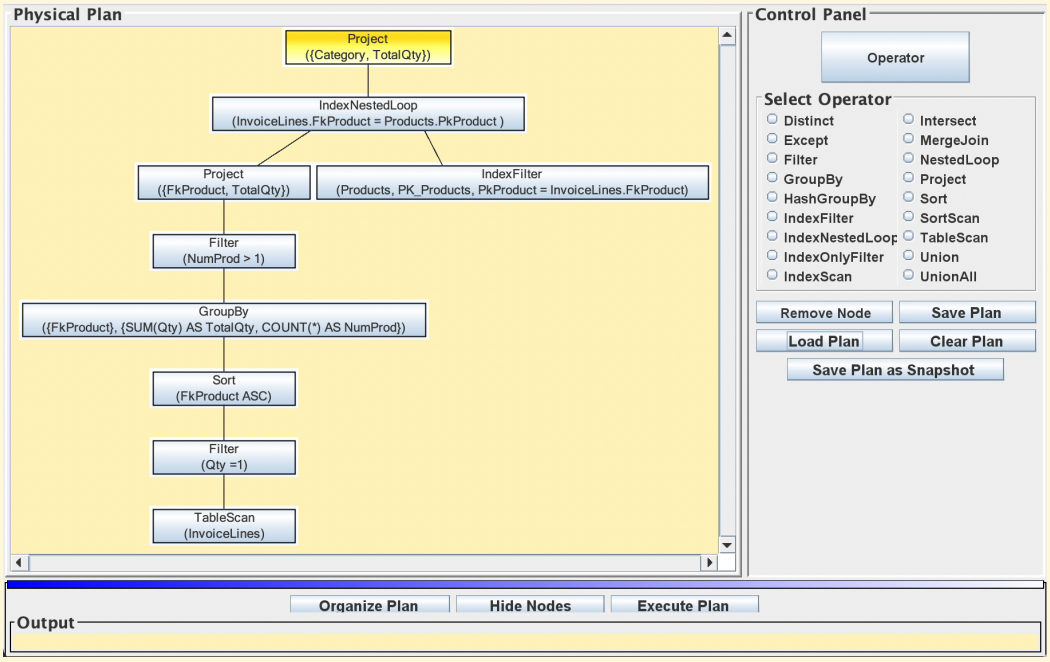


Fig. 8: A **Physical Query Plan** with **GroupBy**

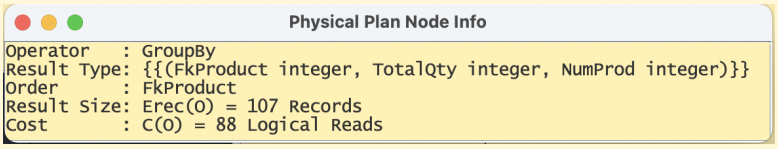


Fig. 9: Information about the **Physical Operator GroupBy**

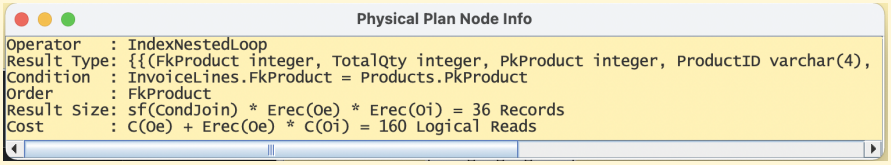


Fig. 10: Information about the **Physical Operator IndexNestedLoop**

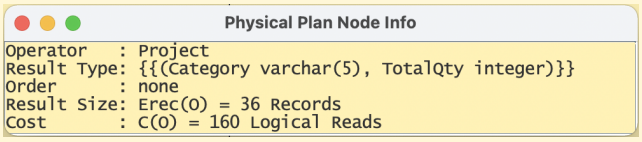


Fig. 11: Information about the **Physical Operator Project**

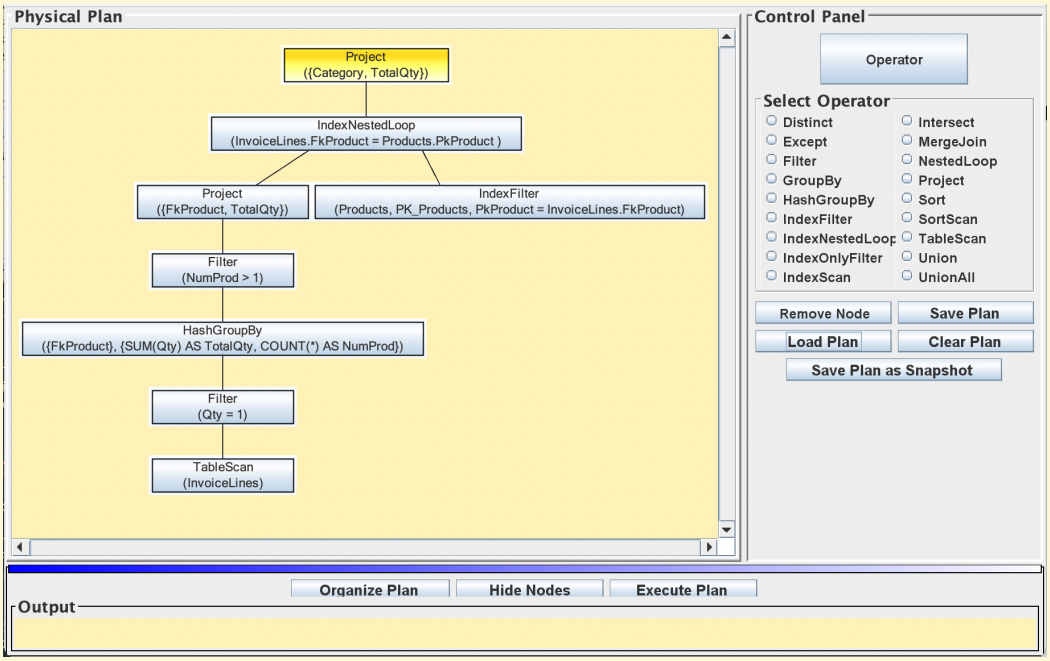


Fig. 12: Another **Physical Query Plan** with **HashGroupBy**