

A System to Support Teaching and Learning Relational Database Query Languages and Query Processing

A. Albano, C. Valisena

University of Pisa, Department of Informatics,
Largo B. Pontecorvo 3, 56127 Pisa, Italy

Abstract. The importance of relational algebra in a database course is widely recognized to facilitate teaching and learning of SQL. From our experience we have also found it very useful for the students to understand the basics of query processing in terms of execution plans. However currently there are no specific tools to make the process of learning relational algebra and execution plans an interesting and stimulating activity. The features of the JRS (Java Relational System) graphical editors of query plans are presented. The graphical editors are used to define and execute queries on a database represented by two kinds of trees: A *logical plan* of relational algebra, and a *physical plan* that describes an algorithm to execute a query using the physical operators of the relational DBMS developed in Java as a teaching tool.

1 Introduction

How to use a relational database system is one of the main topics covered in an undergraduate computer science course, and it is also very common in several other curricula. A database course usually includes conceptual modeling, relational data model and relational database design, relational algebra, SQL, how to build database applications, and the basics of query processing in order to understand how to make physical design decisions.

There is almost universal agreement that knowledge of relational algebra is fundamental to teach and learn SQL as a query language. 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. To make a relational algebra expression more readable, it is usually represented as an expression tree of relational algebra operators, called a *logical plan*.

More advanced database courses are system-oriented to deal with data structures to organize tables and indexes, external sorting, physical algebra, transactions and concurrency management, and query optimization techniques to generate physical plans. A *physical plan* is an algorithm to execute 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 to the query. To make a physical plan more readable, it is usually given as a tree of physical operators.

There are several teaching and learning tools for relational query languages. The focus below is on those that support logical and physical relational algebra.

- ACME is an e-learning framework that supports the automatic correction of problems related to the design of ER diagrams, relational database schemas, normalization, relational algebra and SQL queries. The user defines relational algebra queries using a textual notation [4].
- RAT, Leap RDBMS and Relational are examples of tools for defining and executing textual relational algebra queries [3].
- iDFQL [1] and RALT [2] are examples of interactive systems that enable a logical plan tree to be defined using an interactive graphical interface with a data flow approach. RALT is the only learning tool for relational algebra with the interesting feature of *data lineage* to allow users to track how particular data are derived.

All the above proposals are interesting tools for teaching and learning relational algebra but not physical algebra. Tools such as TOAD (Tool for Oracle Application Developers), go some way in this direction, in fact TOAD shows the Oracle physical plan of an SQL query, how the plan changes by rewriting the query, and a comparison of different query execution alternatives for the same problem. Nevertheless even TOAD-like tools are not really intended specifically for physical algebra.

The proposed interactive JRS environment fills this gap by providing a unique environment with the possibility of practicing with the following different methods in order to formulate relational database queries, and to compare the query results just by graphical interaction using a relational DBMS implemented in Java and designed for educational use which supports a large subset of SQL-92:

- 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 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 general;
 - changing the optimization level from greedy to limited uniform cost or uniform cost;
 - 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.

The paper is structured as follows. Section 2, describes the features of two graphical editors for logical and physical plans. In Section 3, we present examples using the graphical editors, and in Section 4, conclusions and future work are presented.

2 Editor Window Areas

Once a database has been selected, the two graphical editors for logical and physical plans are activated with the **Logical Plan** and **Physical Plan** buttons from the main JRS window. 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.

When activated, a graphical editor displays a window divided into three main areas (Figure 1):

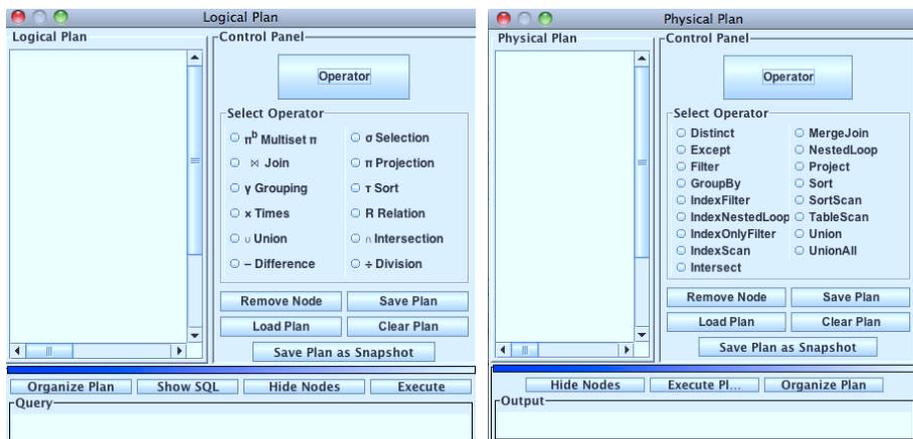


Fig. 1. The Logical and Physical Plan Editors

- **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.

The nodes of a logical plan are the basic operators of the relational algebra (π , σ , γ , \times , \bowtie , \cup , \cap , $-$, and \div) and, in order to have a resulting a set of records, they require that the relations of the leaves of the tree are defined with at least one key. Instead, the result of the operators π^b and τ is a multiset of records, which are provided to define a logical tree of common SQL queries. The graphical editor allows the use of the operators π^b or τ as

the root of an expression tree only. When both are used, the root must be τ .

The nodes of a physical plan are the JRS operators used by the query optimizer to generate access plans.

- **Query/Output:** An area that contains the result of a plan execution, which depends on the type of tree. The area is called **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 selected node*. This feature enables to analyze the result of a plan step-by-step 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 redesign 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.

Since the graphical editors are integrated with a relational system, the definition and execution of a plan is simplified by the fact that the graphical interface performs the following task:

1. *Syntax Checking:* The definition of a node is checked for syntax errors.
2. *Semantics Checking:* The definition of a node is checked for semantic errors including type mismatches. It is not possible to use invalid attribute and relation names with the menu provided.
3. *Query Evaluation:* A logical plan is translated into SQL and evaluated using the JRS Relational Engine, while a physical plan is evaluated using the JRS Storage Engine.

3 Examples Using the Graphical Plan Editor

Examples are given using the database schema in Figure 2 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;
```

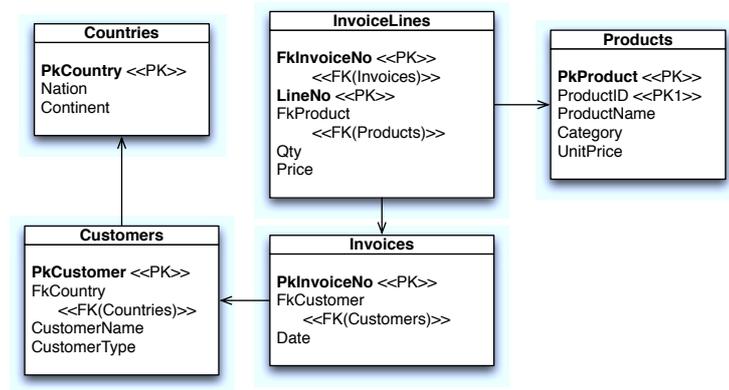


Fig. 2. The Relational Database Schema

Let us first select the option **Show Access 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 an *iterator tree* in a separate window (Figure 3a). 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 number of different values for an attribute, and the existence of certain indexes.

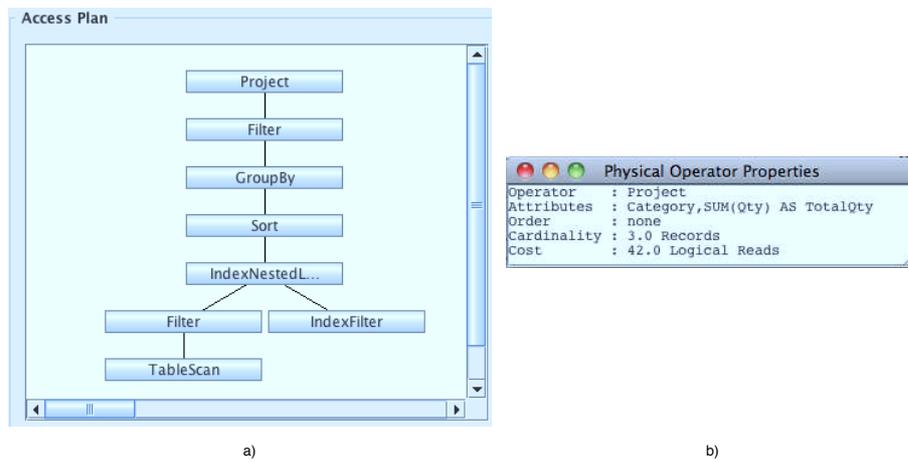


Fig. 3. The Optimized Physical Query Plan

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 3b).

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

A Logical Plan

Let us define the query using the relational algebra, as shown in Figure 4.

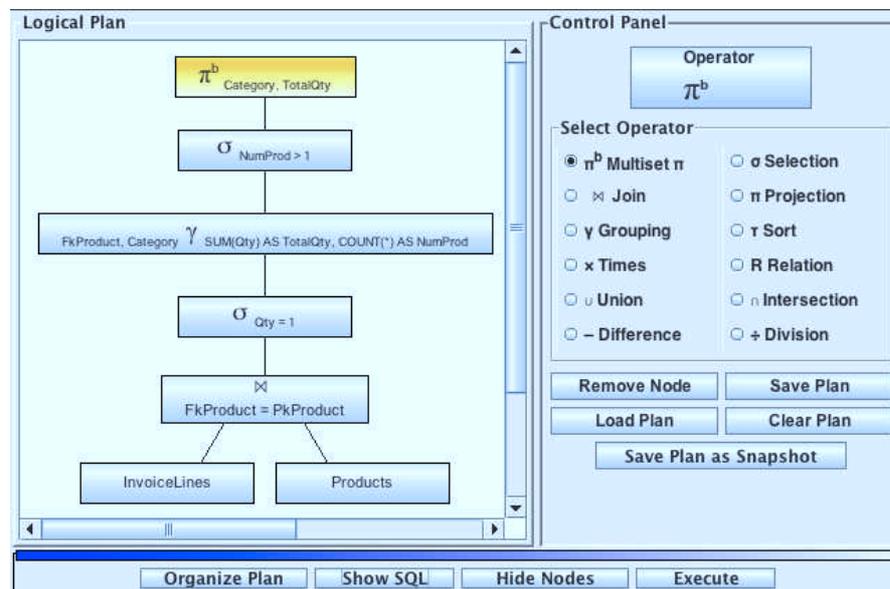


Fig. 4. A Logical Query Plan

Each node of a logical or physical plan has a different contextual menu, which is dynamically created taking into account the attributes of the operands and the schema of the database. Each choice of parameters entails 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, with a right click on the node, but only when the node has all the required operands specified.

By clicking on **Hide Nodes** the logical plan is shown in the traditional form of an algebraic *expression tree* (Figure 5).

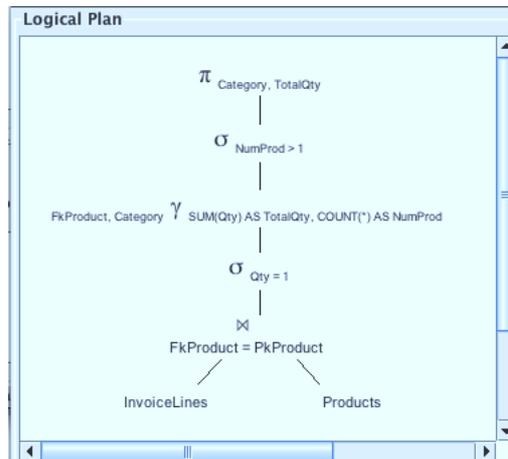


Fig. 5. A Logical Query Plan

Double clicking on a node opens a **Logical Plan Node Information** window with the following information:

- **Operator:** The node operation.
- **Relation:** The relation name of a leaf node.
- **Condition:** The condition for selection and join operators.
- **Result Type:** The operator result type is a set denoted $\{(A_1 T_1, \dots, A_n T_n)\}$, or a multiset, or a sorted set denoted $\{\{(A_1 T_1, \dots, A_n T_n)\}\}$.
- **Order:** The order of records in the query result.

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

- Clicking on **Execute** produces the query result in the query result window.
- Clicking on **Show SQL** opens a query window that displays the SQL query generated by the Logical Editor, for the subtree of the selected plan node. Figure 6 shows the query generated when the selected node is the plan root, which in this case is the same query used to get the physical plan in Figure 3. 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.

Another Logical Plan

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

```

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

```

Fig. 6. The SQL Query to Execute the Logical Plan in Figure 4

Proposition 1. Let $\alpha(X)$ be the set of columns in X and $R \bowtie_{C_j} S$ 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

$$A\gamma_F(R \bowtie_{C_j} S) \equiv \pi_{A \cup F}^b((A \cup \alpha(C_j) - \alpha(S))\gamma_F(R)) \bowtie_{C_j} S$$

if the following conditions are true:

1. $A \rightarrow f_k$, with A the grouping columns in $R \bowtie_{C_j} S$.
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).

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

```

CREATE VIEW 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.

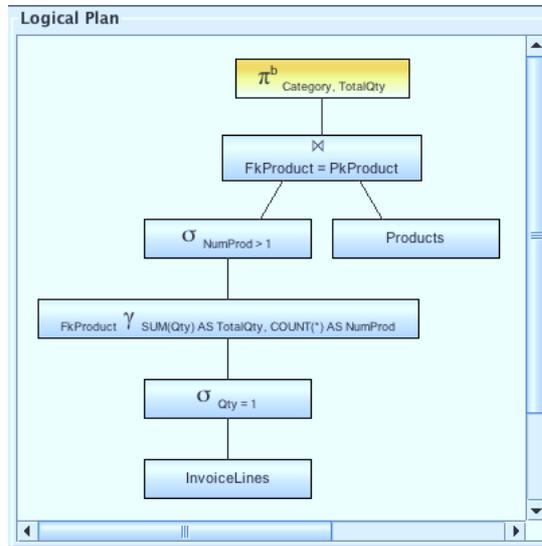


Fig. 7. Another Logical Query Plan

A Physical Plan

Let us now design a physical plan that exploits carrying out the group-by before a join, 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* (Figure 9).

Double left clicking on a node displays a box with 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 $\{(A_1 T_1, \dots, A_n T_n)\}$
- **Order:** The order of records in the query result.
- **Cardinality:** The estimated number of records of the plan result.
- **Cost:** The estimated number of pages read from or written to disk to produce the result.

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.

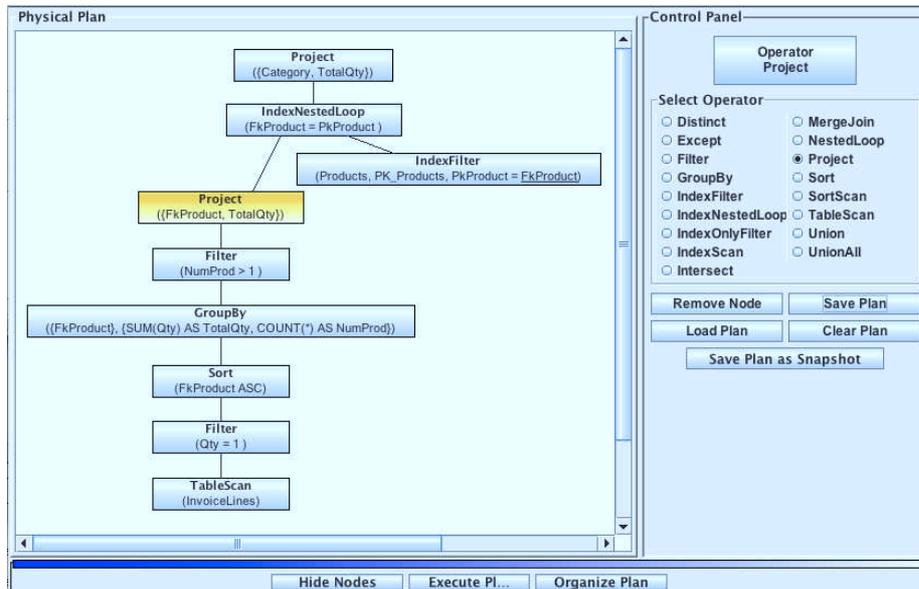


Fig. 8. A Physical Query Plan

4 Conclusions and Future Work

We have presented the JRS relational database system with graphical editors of executable logical and physical plans to support the teaching and learning of query languages and query processing. The main feature of this system is that, unlike other e-learning environments, it has been designed to support both the teacher and student to experiment not only with the SQL language to query a data base and to analyze the query plans generated by the query optimizer, but also to experiment (a) with the execution of a logical plan defined with relational algebra, and (b) with the execution of physical plans defined with the physical operators of the database system. The features of JRS attract student interest but avoid them having to spend their time writing syntactically and semantically correct solutions. Instead the focus is on understanding the concepts of query languages and query processing, and being motivated to experiment with different solutions to solve a problem. In addition, the node information given by the graphical interface is useful for getting the students used to the properties of each operator of a query plan. Thus they learn to give the correct answer to the solution in a written examination that tests the student's ability concerning query processing.

Several system improvements are under consideration. First of all the graphical interface in order to make the interactions simpler. In addition we aim to improve the translation of a logical plan into SQL, to add other join operators, to generalize the relational algebra division operator to express general queries

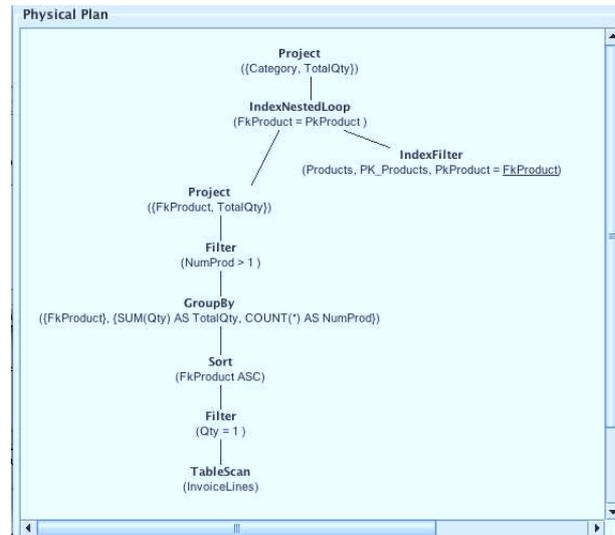


Fig. 9. The Iterators Tree

involving universal quantification, and to provide enhanced functionality and new features on the basis of the most-requested improvements by system users.

Acknowledgments. Thanks to the anonymous referees for their constructive comments.

References

1. Appel A., E. Q. Silva, C. Traina Jr., A. J. M. Traina. iDFQL - A Query-based Tool to Help the Teaching Process of the Relational Algebra. In *World Congress on Engineering and Technology Education, WCETE*, Guararujá, SP, 2004.
2. Mitra P. Relational Algebra Learning Tool. Imperial College, London, 2009.
3. *Relational Algebra*, <en.wikipedia.org/wiki/Relational_algebra>, March 2011.
4. Soler J., I. Boada, F. Prados, J. Poch, R. Fabregat. An Automatic Correction Tool for Relational Algebra Queries. In M., Gervasi and M. Gavrilova (Eds.) *ICCSA 2007*, LNCS, vol. 4706, pp. 861–872, Springer, Berlin Heidelberg, 2007.