Abstract: We introduce a recursive query evaluation method whose main goal is to obtain recursive query cost estimate without performing the query. In particular, we show that the execution estimate of the parallel processing of recursive queries can be done by exploiting the characteristics of the database profile during the evaluation process for request optimization. As many recursive queries involve computation of a transitive closure, the method estimates both transitive closure sizes and transitive closure execution costs.

1 Introduction

To support such applications as integrated office systems, design and engineering, or expert systems, database systems of the new generation must support complex objects and/or deduction as well as offer extensibility. In addition, these systems must deal with a wide range of computer architectures, and provide for parallelism and adaptability.

Coupling the support of recursive queries on complex objects, the use of parallelism, extensibility, and adaptability requires a new approach to database system design. A possible way to achieve this goal is to embed a large part of the coupling in a query-cost evaluation tool. This paper describes such a tool and focuses on cost evaluation techniques for complex recursive queries.

ACE, an adaptive query-cost evaluator tool designed at the Bull Corporation Research Center [AND90a], provides extensibility and adaptability through its library-oriented design. Four libraries provide parameters and formulas describing architectures, relational operations and access methods, systems, and databases. With this tool, it is possible to study the effect of chosen environment parameters on the cost of given queries, while keeping the values of the other environment parameters fixed. Thus, questions such as: "what is the best algorithm for a given parallel architecture?", or "what is the best architecture for a given application (database + queries)?", can be answered. ACE provides a means of experimentally evaluating new designs without having to actually build a DBMS to do so. For ACE to accept query trees involving a transitive closure operation it was necessary to define a methodology for the estimation of recursive query costs in the context of various parallel environments.

Several efficient parallel transitive closure algorithms have been proposed to take advantage of parallel architectures. They can be classified in two classes: those algorithms that assume a shared memory architecture and those that assume a distributed memory architecture.

In the first class, we mainly find parallel transitive closure algorithms as Warren and adaptations of Warren algorithms (for example [LU87]). The two main parallel transitive closure algorithms of the second class are TCPO (Transitive Closure with Parallel Operations) and TCPP (Transitive Closure with Parallel Programs) presented in [VAL88]. TCPO is based on the execution of the individual operations of the transitive closure in parallel, while TCPP is based on the execution of the transitive closure program in parallel. A new efficient algorithm named PTCH (Parallel Transitive Closure based on Hashing) [HAM90] has been also proposed and fits into the second class.

* This work was supported by the EDS Esprit II project under grant 2025.
Transitive closure algorithm performance is traditionally studied independently of specific queries. In particular the size of the result is an assumption within the context of the experiment [BAN88, VAL88]. For the cost evaluation of more general query trees, especially when other operations are performed at each step of the recursion, and when the final result of the closure operation is further input to other operations, a new recursive query evaluation method is necessary including the estimate of both the compute time and the result sizes.

In this paper, we present the recursive query evaluation method used by our multi-environment cost evaluator. A method of estimating intermediate transitive closure sizes and transitive closure parallel execution costs is proposed. The paper is organized as follows. In the next section, we present a review on cost evaluation. Section 3 presents a brief overview of ACE. In section 4, we outline through an example how ACE evaluates the cost of fixpoint parallel execution. Section 5 describes size estimates for \( U \) (Union) and \( \Delta \) (The increment) for semi-naive algorithms and extends the method to aggregates. In Section 6, we report results obtained from evaluation experiments and finally section 7 draws some conclusions.

2 Cost Evaluation

The evaluation of the cost of a query involves two main aspects: size and time. Sizes must be estimated for both permanent relations and operation results. Permanent relation sizes are stored in database profiles while result sizes must be estimated and can be stored in similar structures.

A number of studies have evaluated the performance of transitive closure algorithms. Two main methodologies are used. First, an analytical model [VAL88, HAM90] has been proposed to evaluate such algorithms. This model especially concentrates on the effects of parallelism. Thus it only evaluates roughly the cpu cost of the execution of the algorithms. This simplification of the cost analysis cannot be done when the number of tuples generated by the transitive closure is small or varies a lot with each iteration. Secondly, H. Lu [LU87] proposes a finer analytical model which is adaptable and extensible for other relational operators. His model has a better cost decomposition which can be compared to that of [SWA89]. A first attempt at combining the two approaches can be found in [AND90b].

Regarding result size estimation, many methods have been proposed for classic relational operators. These methods fall within two main categories according to how much is known about the shape of the distribution: parametric and non-parametric methods. With parametric methods, the distribution has a well-known form (uniform, normal, Pearson family and Zipf, ...). The mean and the standard deviation are estimated from the distribution. Works as [FED87, CHR83] have proposed such methods. D. Gardy [GAR90] has demonstrated that the size of derived relations can be computed by multivariate generating functions. She proposes a modelization of generating function parameters with suitable urn models. With non-parametric methods, nothing is assumed about the form of the distribution. Several statistical methods have been used to keep a summary of the data: different types of histograms depending on the criterion chosen to set interval boundaries (equal-width histograms [KOO82], equal-height histograms [PLA84], multi-dimensional histograms [MUR82]), sampling [OLK86], moments and density functions [GRA87] described later.

Recent proposals introduce new statistical methods for modeling attribute value distributions. In System MULTI-STAR, research efforts were directed to develop the "Piecewise Uniform Method" [BEL89]. Instead of keeping single average values to represent the number of occurrences of each attribute value, they introduce \( m \) parameters which each represents the number of occurrences of attribute values in a partition, corresponding to a subrange of \( 1/m \) of the original value range. This model supports uniform and non-uniform distributions, and intra-relation attribute value correlations.

Graefe [GRA87] derived reliable formulas to describe data distributions from moments and density functions to estimate the size of selection and join results. The ex-
3 Overview of ACE

ACE [AND90a] is a tool designed for evaluating the cost of parallel execution of queries. ACE avoids the "lack of extensibility and adaptability" of the traditional approach that embeds a cost model into a query optimizer [GRA87].

The multi-environment-oriented nature of ACE allows the evaluation process for several kinds of architectures, of operators and access methods, of applications, of systems, and of queries. This feature of ACE is supported by a library-oriented design. Libraries are the way to store the knowledge about the specific environment, providing for extensibility and for adaptability without compromising the efficiency of the cost evaluator.

Four libraries are supported by ACE: (1) the architecture library modeling the hardware architecture where the DBMS is installed, (2) the database profile library grouping the knowledge about the database schema along with statistics about user data, (3) the operator and access method library modeling the algorithms used by the data manager, and (4) the system library modeling the operating & transactional system supported by the hardware architecture and supporting the DBMS (see Figure 1).

4 Recursive Query Cost Evaluation in ACE

In this section, we outline some key functionalities of ACE for recursive query cost evaluation by means of an example.

4.1 Operand Relations and Views

We consider an inventory database, where the following information is a relational data model with two relations where key attributes are given in bold:

- **BASIC-PART** (Part#, Supplier, Country, Price)
- **ASSEMBLY** (Part#, Subpart#, Quantity)

and the recursive view **PART-PRICE** as in the ESQL language [GARD90]:

CREATE VIEW PART_PRICE AS

SELECT Part#, Price
FROM BASIC-PART
UNION
SELECT A.Part#, SUM(A.Quantity * C.Price)
FROM ASSEMBLY A, PART-PRICE C
WHERE A.Subpart# = C.Part#
GROUP BY A.Part#;

This view gives for each composite part the sum of the cost of its subparts taking into account the number of their occurrences. We assume that the base relations are stored and split horizontally on several nodes of the computer. We define the *home* of a relation as the subset of the nodes where the relation is stored.

4.2 Query Evaluation

The following query mixing recursion and aggregate illustrates the execution-time estimate done by the cost evaluator:

SELECT PART#
FROM PART_PRICE
WHERE PRICE > 100000
This ESQL request retrieves all the parts whose price is more than 100,000 francs.

Let us introduce the following syntax of the fixpoint constructor as defined in [KEL89]:

\[
\text{<fixed-rel-point> } = \text{FIXPOINT (<home>, <algorithm>)} \quad (<\text{initial-relation}>, <\text{iterative-relation}>)
\]

The fixpoint is defined by reference with the naive closure algorithm, independently of the actual algorithm specified in the <algorithm> annotation. The <home> annotation is as defined before.

Here <initial-relation> specifies the initial set of tuples and may be either empty, a relation identifier or a relational expression which is computed at initialization. The second parameter <iterative-relation> specifies the relational expression to be applied at each iteration. To compute new tuples (of the same schema), the filter_map operator has a similar syntax and corresponds to a simple selection (projection restriction) in our example. Its syntax is defined as follows:

\[
<\text{Output relation name}> = \text{FILTER-MAP} \\
[<\text{home}>, <\text{global algorithm}>, <\text{local algorithm}>] \\
(<\text{input relation information}>, <\text{predicate expression}>, <\text{map expression}>)
\]

A possible execution plan given as input to ACE is:

```
part_price = FIXPOINT
(home(assembly), semi-naive/set oriented)
PROJ
(home(basic_part), all-select, scan)
(basic_part(b) (stored), (b.part#, b.price)),
JOIN
(assoc-join(assembly.part#, nested-loop)
(assembly(a) (stored), part_price(p) (stored),
a.subpart# - p.part#, (a.part#, sum(a.qty * p.price))))
RES = FILTER_MAP
(home(assembly), all-select, scan)
(part_price(p) (stored), p.price > 100000, (p.part#)).
```

On this plan, <initial_relation> becomes a Proj instruction which will be executed on the home of the relation BASIC_PART with the global algorithm1 all-select and the local algorithm2 scan. The last two arguments provide information on the way of getting BASIC_PART tuples (in this case, the relation BASIC_PART is stored) and the map expression (basic_part.part#, basic_part.price) to indicate the attributes to keep. <iterative_relation> becomes a Join instruction. The global algorithm is an assoc-join which indicates that the relation ASSEMBLY is reorganized over the nodes by a hash function on the attribute part#. The local algorithm is a nested-loop join. Both input relations are specified: ASSEMBLY (a) and PART_PRICE (p) which are stored. The two last arguments are the selection predicate and the map expression which specifies the attributes of the result relation.

The annotation "semi-naive / set oriented" refers to a specific algorithm whose description gives information about the way the table part_price must be transferred and the way the results must be reinserted at each iteration. This algorithm was chosen as an example for its simplicity and because the method described herein is by a large independent of the details of the algorithm.

Let us now describe the Filter_map instruction. <Output relation name> is RES. <home> represents the home of the relation ASSEMBLY where this instruction will be executed. <global algorithm> and <local algorithm> are respectively all-select and scan. <Input relation information> provides the name of the input relation (PART_PRICE) and where the tuples of this relation are stored. <predicate expression> and <map expression> are respectively p.price > 100000 and p.part#.

Given this plan as input, ACE constructs the Cost Analysis Graph of the request. To construct the graph, ACE applies a transformation on the request plan and produces an intermediate Cost Analysis Program (Figure 3) to help

---

1 an algorithm is called global if it concerns all nodes of the architecture.
2 an algorithm is called local if it concerns each node locally.
the user understand the cost evaluation. This Cost Analysis Program makes implicit all the execution mechanisms whose cost is estimated by ACE (for example data transfers, how the result has to be reinserted at each iteration) and develops the low level cost operators: ALGO_EVAL, STORE_EVAL, TRANSMIT_EVAL.

These functions produce a relation profile as result and update costs. The costs are initialized as global variables and are explicit in the Cost Analysis Program. Figure 3 introduces the new data operation ADD [BOR90] whose behaviour is quite similar to the INSERT one, except that it produces as an output a stream of data corresponding to the actual new inserted tuples. $10.\text{profile}$ is the profile of the relation $10$ which describes the stream of the new inserted tuples. $10.\text{profile}$ represents the relation $10$ in which the tuples of the relation $R2$ (e.g. $R2.\text{profile}$) are inserted.

```plaintext
R1.\text{profile} = \text{ALGO}_\text{EVAL}
       = \text{PROJ, scan, (basic_part.\text{profile}(a),}
          (a.part#, a.price));
R2.\text{profile} = \text{TRANSMIT}_\text{EVAL}
       = \text{R1.\text{profile, home(assembly)-hashed on part#};}
10.\text{profile} = \text{ALGO}_\text{EVAL}
       = \text{ADD, (R2.\text{profile, part_price.\text{profile})}};
A.\text{profile} = \text{STORE}_\text{EVAL}(10.\text{profile});

\text{WHILE } A.\text{profile.size} < 0
\begin{align*}
1.\text{profile} &= \text{ALGO}_\text{EVAL} \\
       & = \text{JOIN, nestedloop, (assembly.\text{profile},} \\
       & \text{A.\text{profile, assembly.subpart#} = A.part#}, \\
       & \text{assembly.part#, sum(assembly.qty * A.price));}
11.\text{profile} &= \text{TRANSMIT}_\text{EVAL} \\
       & = \text{R1.\text{profile, home(assembly), hashed on part#};}
12.\text{profile} &= \text{ALGO}_\text{EVAL} \\
       & = \text{ADD, (11.\text{profile, part_price.\text{profile})}};
A.\text{profile} &= \text{STORE}_\text{EVAL}(12.\text{profile});
\end{align*}

\text{END WHILE}

R5s.\text{profile} = \text{ALGO}_\text{EVAL}
       = \text{FILTER, MAP, scan, (part_price.\text{profile,}} \\
          \text{part_price.price > 100000, (part_price.price#));}

\text{Fig 5: Cost Analysis Program}
```

The Cost Analysis Graph represented in Figure 4 is the internal result structure of the cost evaluation.

```
\begin{tikzpicture}
  \node (R1) {R1 pf};
  \node (R2) [below of=R1] {R2 pf};
  \node (store) [below of=R2] {Store Cost};
  \node (add) [below of=store] {Add Cost};
  \node (transmit) [right of=add] {Transmit Cost};
  \node (part_price) [below of=transmit] {PART_PRICE pf};
  \node (assembly) [below of=part_price] {ASSEMBLY pf};
  \node (part) [below of=assembly] {PART pf};
  \node (basic_part) [above of=R1] {BASIC PART pf};

  \draw [->] (R1) -- (R2);
  \draw [->] (R2) -- (store);
  \draw [->] (store) -- (add);
  \draw [->] (add) -- (transmit);
  \draw [->] (transmit) -- (part_price);
  \draw [->] (part_price) -- (assembly);
  \draw [->] (assembly) -- (part);
  \draw [->] (part) -| (basic_part);
  \draw [->, dotted] (R2) -- (part_price);
  \draw [->, dotted] (part_price) -- (assembly);
  \draw [->, dotted] (assembly) -- (part);

\end{tikzpicture}
```

\text{Figure 4: The Cost Analysis Graph}
```

The dotted arc indicates that the cost evaluation goes on with the Filter Cost operator at the end of the iterative join. The arrows which are cut specify materialization. The notation pf means profile.

This Cost Analysis Graph (CAG) is subjected to four phases of processing. Depending on its content, these phases may be separated by transparent time interval. The four phases of an instruction evaluation are CPU Cost evaluation, Output Profile Creation, Data Transfer Cost Evaluation, and Data Materialization Cost Evaluation as shown in Fig 5. CPU Cost Evaluation and the Output Profile Creation are performed in the Local Cost Evaluator. The Data Transfer Cost Evaluation and the Data Materialization Cost Evaluation is performed in the Global Cost Evaluator.

The CPU Cost Evaluation is characterised by the evaluation of the cost of the local algorithm of each relational operator specified in the Cost Analysis Graph (e.g. Proj Cost, Join Cost, and Filter Cost). When this processing is over, the Output Profile Creation is started. This phase performs a size estimate then a distribution estimate of the instruction. Finally, it creates the profile of the instruction result (e.g. R1 pf, R2 pf, 10 pf, ASSEMBLY pf). The final phase is the data transfer cost evaluation and
the data materialization cost evaluation. It first evaluates the data transfer cost in terms of the cost of data streams. The data stream provides data transfer between two processors (Evaluation of Transmit Cost). Then it evaluates the data materialization cost as the cost of storing the result relation (Evaluation of Store Cost). The values of the metrics introduced in section 2 are iteratively calculated for each operator of the Cost Analysis Graph. The costs, the output result of the evaluation, are stored in each associated node of the graph.

The 

\( \Delta \) method: If ACE has no previous knowledge on the transitive closure (for example this relation has been sent in pipe), the number of iterations is implicitly given by the breakpoint check on the condition \( \Delta . \text{profile}. \text{size} \neq 0 \) and \( (90\% < \varepsilon) \) where \( \varepsilon \) is the confidence rate.

After each iteration of the while loop in the iterative transitive closure algorithm (Figure 4), \( \Delta . \text{profile}. \text{size} \) is given by the selectivity factor of the join.

The without-\( \Delta \) method: ACE calculates the number of iterations of the transitive closure directly from its profile. When only a small subset of the entire graph is to be closed, the number of iterations is given by the average height of the graph defined by the recursive relation.

In the iterative Transitive Closure Algorithm presented Figure 3, after \( i \) iterations of the while loop:

\[
\text{PART_PRICE}. \text{profile} = \text{U} \text{ in } [0,i]
\]

(ASSEMBLY.\text{profile} \ join BASIC_PART.\text{profile})

The algorithm terminates when

ASSEMBLY.\text{profile} \ join BASIC_PART.\text{profile} is equal or included in

\[
\text{U} \text{ in } [0,i-1]
\]

ASSEMBLY.\text{profile} \ join BASIC_PART.\text{profile} (1)

The number of iterations is given by the graph defined by the recursive relation. Let \( x \) and \( y \) be any two nodes in the graph, let \( \text{minpath}(x,y) \) denote the shortest path from \( x \) to \( y \) in the graph, the number of iterations \( p \) is given by:

\[
\text{p} = \max_{x,y} (\text{minpath}(x,y))
\]

(1) holds when \( p \leq i \), so that the execution plan presented in Figure 4 requires \( p \) iterations to be computed. In this case, the size of the result is given by the following formula:

\[
\text{U}. \text{profile}. \text{size} = \sum_{i=0}^{\infty} H \text{ fan-out i}
\]

where \( H \) is the average height of the graph structure.

The Statistical Profile:

The estimation method previously described needs both a minimal set of statistical data for each important attribute in each base relation and statistics on derived relations. An attribute is called important according the user appli-
cation if it is frequently used in requests. The statistics are provided from the database structure which is relational-oriented (relation, attribute, index...). It is structured as sets of statistics: central tendency (mode, mean, and median), dispersion (minimum range, maximum range, variance and standard deviation), sizes (number of instances, of distincts values), frequency distribution (normality, uniformity, and value intervals), and graph statistics (max, min, mean for path length, fan-out factor, etc...).

Histograms, moments and density functions are the tools used by the cost evaluator to store statistic information when the distributions are not uniform. It provides the capability to be adaptable the type of information: histograms for alphanumeric type, moments and density functions for numeric type.

The database profile summarizes the data of the DBMS application and is built from these sets of information. A profile can be seen as a snapshot of the DBMS application. Each profile has an evaluation number (named EVNU). EVNU enables the user to undo query evaluation and to redo it after parameter modification. The higher EVNU corresponds to the newest profile. This profile can either be the real image of the database application, or the result of a query evaluation. Each query evaluation creates a new version of profile. This profile will be the real image resulting from the execution of the query.

An ESQL query is generally translated in several execution plan instructions. So a query corresponds to an instruction tree where intermediate profiles will be built. Each derived profile (intermediate or final one) reflects the evaluation of query executions and so carries a degree of uncertainty in the estimation accuracy. Thus, we introduce the confidence rate to provide the accuracy of the estimated and propagated information.

6 Experiment Report

In this section, we report first results obtained from the evaluation of the execution plan given Figure 4.

The execution plan was given to the cost evaluator for two types of architectures. The first one is a distributed memory multiprocessor architecture which comprises nodes (local memory + processor) grouped by a bus. The second is a shared memory multi-processor architecture. We made the number of processors for each architecture configuration vary.

Basic relations size are defined as follows:

Basic_Part size : 200 tuples,
Assembly size : 800 tuples.

The value distribution of the basic relations is uniform.

The actual number of tuples produced is equal to 524.

The estimate number of tuples given by the cost evaluator is equal to 650. To compute the result, it has required 6 iterations. The loop-breakpoint check is done after each iteration. Figure 6 compares Δ size estimate and actual size computation.

<table>
<thead>
<tr>
<th>number of iterations</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δ size estimate</td>
<td>282</td>
<td>184</td>
<td>62</td>
<td>55</td>
<td>42</td>
<td>25</td>
</tr>
<tr>
<td>actual Δ size</td>
<td>212</td>
<td>175</td>
<td>50</td>
<td>46</td>
<td>38</td>
<td>13</td>
</tr>
</tbody>
</table>

**Figure 6: Δ size after each iteration**

Figure 7 introduces various estimates of the execution in 4 different environments.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Architecture 1</th>
<th>Architecture 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>3 Nodes 6 Nodes</td>
<td>3 Nodes 6 Nodes</td>
</tr>
<tr>
<td>Total Time</td>
<td>90 s 60 s 115 s 83 s</td>
<td></td>
</tr>
<tr>
<td>Total Cost</td>
<td>250 s 200 s 332 s 305 s</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7: Result Table**

7 Conclusions and Open Issues

In this paper, we have presented the methodology for parallel recursive query evaluation used by a multi-environment cost evaluator. We have pointed out some lacks of power in query tree evaluation of existing systems where recursion is involved. We have proposed a solution for the specific problem that is original in several ways:
(1) the solution suits a multi-processor architecture, (2) the methodology is based on a multi-environment approach through libraries, (3) it offers the possibility to add easily new cost formulas and to combine several estimation methods to evaluate recursive queries, and (4) recursive relational operator trees can be evaluated.

Our priority for future work is to gain experience with the cost evaluator in the EDS DBMS prototype and other environments. Examples of open research issues are the adaptation of the methodology to new fixpoint algorithms and the study of the evaluation of the recursion in object-oriented environments.

ACKNOWLEDGEMENTS

We thank Bjorn Bergsten, Pascale Borla-Salamet and Jeremy Dick for their comments and useful suggestions on earlier drafts of this article.

REFERENCES


[KEL89] S. Kellett "LERA1 - Language for Extended Relational Algebra (The EDS DBMS Intermediate Interface) EDS Definition Report EDS.DD.111.0001-


