A Multi-Environment Cost Evaluator for Parallel Database Systems

Frédéric Andrès¹, Michel Couprie², Yann Viémont³

¹Centre de Recherche BULL, 64 Route de Versailles
78288 Les Ulis, France
Email: Andres@org.bull.fr

²ESIEE, 2 Boulevard Illiade Pascal, B.P.99
93462 Noisy-Le-Grand, France
Email: Michel.Couprie@org.bull.fr

³Laboratoire MASI, Université Paris 6
45, Avenue des États-Unis, 78600 Versailles, France
Email: Viemont@masi.uv.fr

Abstract: In this paper, we investigate issues involved in designing and using a cost model for query optimization in parallel database environments. The large range of possible multiprocessor computers, the different requirements of data-intensive applications to be supported, and the high number of parallel algorithms and information access methods make a multi-environment oriented approach necessary for the design of a cost model. Our proposal follows this approach and introduces a separation between the optimizer and the cost model. A separate tool called the adaptive cost evaluator (ACE) implements the cost model. The cost evaluator is seen as an intelligent black box by the optimizer, and can also be used independently to investigate other problems such as targeted architectures and data clustering. This design leads to the parameterization of the evaluator with libraries describing the environment (architecture, database profile, system, operators and access methods). In addition, the status report of the implementation of the cost evaluator, along with some query evaluation results, is given.

1. Introduction

New applications such as office automation, stock trading databases, CAD and expert systems highly influence the next generation of information systems. These applications require database management systems (DBMS) more and more efficient in terms of user response time. This goal can be achieved by the improvement of the query compiler, the parallel execution of queries and the use of efficient parallel relational algorithms [BIT83a, VAL84].

The query optimizer is fundamental to obtain high performance. One way to increase its optimization capability is to improve its ability to accurately evaluate query execution time. The role of the query optimizer is to derive an efficient execution plan to get the information requested by the user [VAL90]. This plan specifies all the information (e.g. access methods, operation order) to compute a query. The optimizer should be able to optimize simple queries as well as very complex ones. Some applications such as logic programming [KBZ86] introduce expressions with several hundreds of joins.

To choose an execution plan, the query optimizer needs to evaluate a number of possibilities with the help of some cost evaluation programs. The cost evaluation is based upon a cost model which enables the estimation of query costs while taking into account specific detail of a target computer and target database.

In this paper, we describe a cost evaluator tool appropriate to the design and prototyping of a complete DBMS as part of the EDS' ESPRIT II project [AND90a]. This DBMS is to run on a parallel machine with distributed memory [LOP89]. The major goal of the project is the production of a high performance, highly parallel database server. The query compilation fully exploits the parallelism available in the EDS architecture. In addition to supporting query optimization, the cost evaluator is used to solve other problems such as data placement and performance prediction.

Work on cost modeling in query optimization started with the System R evaluation strategy [SEL79] and with the design of database machine prototypes [SHA79, DEW79] where special-purpose hardware was used to accelerate relational operators. In a number of related works (e.g. [SWA89, WHA87]), the cost model is embedded into the optimizer. In such cases, the cost models are designed for a specific (centralized) environment and could not easily be generalized.

One pioneering effort in flexibility is the EXODUS optimizer generator [GRA87]. Nevertheless, the cost model is still linked to the optimizer and this approach falls short in several ways: first, the costs for complex functions (e.g. when non-linear cost are involved or when data are dependent of each other) are difficult to express; secondly, in the case where several methods are used for the same query, it is difficult easily to separate their costs.

To overcome these problems, we propose a multi-environment approach for the design of the cost model. The associated tool (called Adaptive Cost Evaluator) is implemented as a distinct tool, separated from the optimizer. The knowledge about the environment is decomposed into specific libraries (architecture, database profile, system, operator and access method). This knowledge is expressed by...
the DBMS designer in a declarative form using an Environment Specification Language (ESL) and is then compiled into an internal representation to preserve efficiency.

The main objectives of this design are easy addition of future extensions (extensibility) and the use of the cost evaluator in several software and hardware environments (adaptability). The tool that we develop to achieve these goals may be seen as the first step toward a Cost Model Generator.

The following section illustrates the importance of the cost evaluator with an example. Section 3 describes the design of a multi-environment cost evaluator. The detailed presentation of the multi-environment libraries is given in sections 4, 5, 6 and 7. Section 8 reports the status of our experiments and section 9 draws some conclusions.

2. Cost Evaluator Requirements

By means of an overview of the compiler and an example of query optimization, we plan the requirements of the cost evaluator.

2.1. Overview of the compiler

The compilation process follows a number of phases until the final parallel query program is produced. These phases communicate through an intermediate language called LERA (Language for Extended Relational Algebra) [KEL89a].

The parser performs syntactic and semantic analysis of queries given as input expressed in a language called ESQL (for Extended SQL) and generates an equivalent LERA program. Semantic verifications are done with the help of the cost evaluator. Consider the following relational database with three relations where key attributes are given in bold:

\[
\begin{align*}
\text{CAR} & (\text{PLATE}, \text{MODEL}, \text{MAKE}, \text{COLOR}) \\
\text{CHECK} & (\text{PLATE}, \text{DATE}) \\
\text{OWNERSHIP} & (\text{UCENCE}, \text{PLATE}, \text{DATE})
\end{align*}
\]

We assume that these base relations are materialized\(^2\) and split horizontally on several nodes of the computer. We define the \textit{home} of a relation as the subset of the nodes where the relation is stored.

To summarize, the Optimizer-Parallelizer transforms a sequential LERA program into a parallel LERA program [KEL89b] by applying the best distributed execution strategy determined with the help of the cost evaluator. In this presentation, the optimizer concept refers to a physical optimization processing. The optimizer performs an evaluation of the query tree to produce an optimal (or at least heuristically "good") execution plan\(^1\). It should analyze a set of plans large enough to contain the optimal plan but small enough to keep the optimization time acceptable.

The selection of the plan is based on calls to the cost evaluator. Each call from the optimizer to the cost evaluator (see (3) in Figure 1) can be expressed in a Database Programming Language called physically optimized LERA (or LERA-PHY). This language is mainly characterized by operation annotations (e.g. home of the operation execution, global and local algorithms, data distribution), and by operand annotations (e.g. dataflow mode of the input operands). Both types of annotations are necessary for the cost estimation. The cost evaluator estimates the execution time for each plan with calls to the Cost Evaluator about the database schema (see (4) in Figure 1). This cost estimate is used by the optimizer for an exhaustive or heuristic search among the alternative execution plans.

The Optimizer-Parallelizer has to perform optimization at two levels:

- \textit{local optimization} is limited to one computer node at a time, and only explores the different possible local algorithms. The choice of the cheapest local algorithm depends mainly on the execution time estimate.
- \textit{global optimization} takes into account communication and intermediate and final result construction. It determines which global algorithm will be used to achieve intra-operation parallelism among the computer nodes. The global algorithm specifies which nodes are concerned (i.e. the home of an operation) and how intermediate data are shared among them.

At the end of its work, the optimizer decides the order of the operations in the computation. It also decides how intermediate relations are produced and consumed (materialization or pipelining) for the entire operation tree.

To these two optimization levels, correspond to similar levels in the cost evaluator.

2.2. Query example

Let us take an example to illustrate the interaction between the optimizer and the cost evaluator. Consider the following relational data base with three relations where key attributes are given in bold:

\[
\begin{align*}
\text{CAR} & (\text{PLATE}, \text{MODEL}, \text{MAKE}, \text{COLOR}) \\
\text{CHECK} & (\text{PLATE}, \text{DATE}) \\
\text{OWNERSHIP} & (\text{UCENCE}, \text{PLATE}, \text{DATE})
\end{align*}
\]

We assume that these base relations are materialized\(^2\) and split horizontally on several nodes of the computer. We define the \textit{home} of a relation as the subset of the nodes where the relation is stored.

\(^1\) An execution plan describes sequences of operations, communications and synchronizations leading from the existing relations to the result of the query.

\(^2\) A materialized relation is a relation which is stored in the memory or on disks in contrast with dataflow transfers between operators.
Let us consider the query: "PLATE#, MODEL, MAKE of blue cars that were checked on December, 12th 1989 and that have at least one owner". The corresponding query tree is shown Figure 2. Note that a more complex query would not change the complexity of the cost evaluator task.

![Query tree of the example](image)

In our example, let us look at the subtree Join(selection(CAR, Color = 'blue'), Plate# = Plate#, selection(CHECK, Date = '121289')).

The optimizer has to choose between two plans depending on the way the parallelism is exploited. It can choose either a plan where the two selections are processed in parallel with results materialized before the join operation is done, or a plan where the two selections are computed in parallel but where one result is pipe-lined to the join operation and the other one is materialized. Each plan is translated in LERA-PHY by the optimizer and sent to the cost evaluator for execution time estimation. The first corresponding LERA-PHY program is given Figure 3.

(1) T1 = FILTER_MAP (HOME(CAR), ALL_SELECT, SCAN) (CAR [STORED], CAR.COLOR = "blue", *)
(2) T2 = FILTER_MAP (HOME(CHECK), ALL_SELECT, SCAN) (CHECK [STORED], CHECK.DATE = '121289', *)
(3) P1 = SEARCH (HOME(CAR), ASSOCIATE(T1.PLATE#, NESTEDJOIN) (T1 [STORED], T2 [STORED], T1.PLATE# = T2.PLATE#, *)

![LERA-PHY execution plan](image)

Related work [MAN88] on Profile Estimation Tree is useful for each local cost estimation (e.g. join between P1 and CHECK). Some of the local-cost estimations require statistical information about the database contents (e.g. conditional probabilities).

![Cost SubTree](image)

![Profile Estimation Tree of the example](image)
The Profile Estimation Tree for our example is shown Figure 5. To estimate the size of Pl, the result of the join between T1 and T2, the cost evaluator estimates the size of the result of leaf operations (selection). This estimation modifies the statistical attributes of T1 and T2. In consequence, it has to calculate the conditional marginal distribution (called \( D_E \)) of the attribute \( PZ\# \) in the relations T1 and T2 given respectively the predicate expressions \( \text{COLOR} = 'Ku' \) and \( \text{DATE} = '121289' \). Similar estimations are done for sizes and distributions of the successive operation results.

Through this example, we show that the cost evaluator role is to combine result size estimations and time estimations to guide the optimizer choices.

3.ACE : An Adaptative Cost Evaluator

In this section, we give an overview of the cost evaluator implementation. The architecture of the cost evaluator is intended to solve shortcomings in query evaluation. First, each DBMS contains a module to estimate query execution times which is specific to that DBMS environment and is not compatible with that of another DBMS. Second, specific details of hardware and software environments are usually not fully taken into account by the cost model of the optimizers. Our multi-environment approach leads to a library-oriented design which is described in 3.1. Then we introduce the cost evaluator kernel and the expert library in the subsections 3.2 and 3.3. In 3.4, the metric definitions for the costs we want to estimate are given.

3.1. The Libraries

In a multi-environment approach, libraries are a way to store the knowledge about a specific environment, providing for extensibility and for adaptability without compromising the efficiency of the cost evaluator. Environments are described in a language called Environment Specification Language (ESL). Clauses in ESL grammar are grouped in four distinct units corresponding to four libraries.

The four libraries are (as shown Figure 6):

1. The Architecture Library modeling the hardware architecture where the DBMS is installed.
2. The System Library modeling the operating & transactional system supported by the hardware architecture and supporting the DBMS.
3. The Operator and Access Method Library modeling the algorithms used by the data manager.
4. The Database Profile Library grouping the knowledge about the database schema along with statistics about user data.

Users enter environment information through ESL one unit at a time. Each unit is compiled to generate a library structured in an object-oriented fashion (the current implementation is in object Pascal). We had two objectives in mind when defining the ESL grammar. First, it provides for reusability of environment declarations: a set of declarations, corresponding to either an architecture component or a cost subformula, can be given a name and used several times in other declarations. Secondly, it makes implementation easier, since the parser outputs directly the desired objects for the libraries.

3.2. The Cost Evaluator Kernel

The cost evaluator kernel consists of two layers performing non-local cost evaluation (estimation of communication and materialization costs) and local cost evaluation (estimation of local algorithms) as shown in Figure 6. The first layer takes into account temporary relations and

![Figure 6: The Cost Evaluator Architecture](image-url)
data transfers by pipes. The estimation processing is based on the assumption that the way of consuming an input relation indicates how to produce these relations created by other previous operators. Such a policy can be either a relation store, or a relation transmit (pipe line). That means that this layer studies the shared relations between LERA PHY instructions. The second layer calculates local cpu cost and result sizes using the information stored in the libraries and input operand size estimations. The cost evaluator kernel can also call experts (defined in subsection 3.3) to solve specific problems (e.g. The Data Placement Estimation).

3.3 The Expert Library

An expert library corresponds to a unit where some knowledge and a method are linked to solve a specific problem. A typical example is data placement. The cost evaluator is to be used to help solve the data placement problem for a given data schema and a given class of queries. Such an expert library is yet considered as an open problem in the project.

3.4 Metrics Definition

We divide the cost of a query into three components: response time, total time (also often called execution cost) and inter-arrival time. The response time (named RT) of a query is defined as the estimated time elapsed from the initiation of the query until the first result tuple is produced. The total time (named TT) of a query is defined as the estimated time elapsed from the initiation of the query until the last result tuple is produced. The inter-arrival time (named IAT) represents the average interval between the production of two tuples.

The total time is given by the following formula:

\[ TT = RT + (\text{number of tuples produced} - 1) \times IAT. \]

This proposal is an extension of [SU86] to more general operators and query trees. Other performance metrics such as throughput and burden defined in [AGR85] will be calculated for query estimation in multi-user environments.

4. Modeling of the Architecture

Knowledge about the architecture is expressed in ESL through the architecture unit. This unit contains clauses describing the overall architecture and clauses describing components. Component declarations are named, so that the same component can be used several times in the same architecture or in different architectures. In this section, we introduce the motivations behind the definition of architectures.

To allow EDS researchers the exploration of several variants, the tool must be usable on a large range of computer architectures, especially multiprocessor architectures as the current EDS computer. Although this latter architecture, and hence the core of this paper, is distributed-memory oriented, future studies will also be done for shared memory architectures. Adaptability of the cost evaluator should be achieved without having to rewrite software. For that purpose, a simple mapping between the architecture-dependent input variables of the cost computations and the corresponding architecture characteristics is critical.

4.1 Components

The component part of the architecture library contains the lowest level knowledge about the target architectures: it describes the memory, cpu, and interconnection components. Each component is described in terms of specific properties. Figure 7 describes the properties for the first version of the tool.

Two metrics are used to express information: the number of instructions and execution time in seconds of an instruction. In addition, several simplifying assumptions have been made about CPU costs:

- several compound operations are considered as cpu attributes when there is no (or little) variant in software.
- the cost of an elementary sort operation \( C_{sort} \) includes both comparison and swapping costs.

At the user level, components are declared in ESL through the MEMORY, CPU, and INTERCONNECTION clauses of the grammar given in annex. Each value is given a name used as a terminal symbol in overall architecture definitions.

4.2 Overall Architecture

The overall architecture part of the architecture library contains a high level description of the architectures used in the studies done with the cost evaluator. At this level, a computer architecture is viewed as a collection of interacting components and the focus is on such issues as distributed versus shared memories, number of levels in memory hierarchies, interconnection types and width, etc...

For example consider the distributed memory multiprocessor architecture of Figure 8. On this simplified example, we have only kept details relevant to the following discussion. The architecture comprises three nodes (local
memory + processor) grouped around some interconnection network. It can easily be described in ESL using three basic components as follows (provided "deltacpn", "memcpn", and "cpucpn" components have already been described):

$$EDS1 = ARCH-BEGIN$$
$$DELTA-NETWORK deltacpn;$$
$$DM (3 * (MEMORY memcpn, CPU cpucpn));$$
$$ARCH-END$$

**Figure 8: Sample Architecture 1**

Now consider the (simplified) architecture of Figure 9. This multiprocessor architecture has a shared memory and five processors with local caches. It can similarly be described in ESL as follows:

$$EX2 = ARCH-BEGIN$$
$$BUS buscpn;$$
$$SM GLOBAL MEMORY memcpn;$$
$$5 * (CACHE cachecpn, CPU cpucpn);$$
$$ARCH-END$$

**Figure 9: Sample architecture 2**

Properties of the overall architectures are derived from the properties of the components. The same component can appear several times in a given architecture and can be shared among different architectures.

5. The Database Profile

In this section, we present the role of the database profile (e.g. execution time estimation, result size estimation). Then we introduce the motivations and basic principles that guide our design of the database profile library and the methods used by the cost evaluator to estimate result size.

5.1. Definition and related work

The database profile is a simplified but representative knowledge of the database schema. It also contains statistics about the database. This profile is a "melting pot" where abstract and physical database levels are summarized [SEV81]. The Database Profile is mainly useful in the cost model for the estimate of query result sizes. These sizes are used to update the memory occupation rate when result tuples are materialized and are also necessary to estimate the query execution time.

Related work has been done in several directions. Several approaches analyze the needs of cost estimation. Mackert and Lohman [MAC86a], [MAC86b] show that size estimation is a main factor in both I/O and CPU components in the cost model. Others based on nonparametric and parametric methods [CHR84], identify several components which influence the cost model. First, the distribution of attribute values as well as the independence or non-independence of those values affect the number of tuples accessed for a given request. Second, the distribution of queries in relation to attribute values affect the estimate of size of the result: the queries may or may not be uniform when they reference attribute values. Others present methods based on adaptive sampling [LIP90] and introduce a new way to estimate query results. This last approach requires no distribution probability for the data and no detailed statistics about the database.

Our definition of the database profile adds upon work by Christodoulakis [CHR84] and others [MUR87, PIA85]. Our approach combines parametric methods, nonparametric methods and sampling methods. This provides adaptability to the cost evaluation. It also allows the cost evaluator to choose the best method for each operator in the same query.

5.2. The Database Profile Library

This library contains information about the database structure which is relational-oriented (relations, attributes, indexes, ...). The components of a database profile are shown in Figure 10. Several database profiles may be introduced in a library, each profile being identified by a database name. The Database Profile Unit of the ESL language is used to specify the contents of the Database Profile Library (see in Annex for the ESL grammar).

**Figure 10: The Database Profile**
When used within the context of the EDS optimizer, the database profile library is managed by the catalog manager.

6. Modeling of relational Operators and Access Methods

In this section, we describe how the knowledge about relational operator algorithms and access method algorithms is organised. Access methods based on less-conventional index structures, such as AVL [DEW84] and/or Quadtree [SAM90] may be described to the cost evaluator.

For each algorithm (either relational operator or access method), we associate a set of costs formulas. A cost formula is an expression of the form:

\[ C = f(P_1..P_n) \]

where \( C \) is a component of the cost (CPU cost, init cost ...), \( P_1..P_n \) are parameters like the relation size, selectivity or CPU speed and \( f \) is a (generally simple) function.

To evaluate a given the cost formula, the cost evaluator must first find the value of each formal parameter. These values can be found:
- in the query (relation & attribute names, ...)
- in the architecture library
- in the database profile library (relation size, distribution of values...)
- in the system library.

For example, the ESL sequence of Figure 11 declares the cost formulas corresponding to the Filter_Map operator. This relational operator is implemented with a scan algorithm applied to the tuples of the relation given as input arguments in LERA-PHY queries.

7. System Library

We showed in the previous sections how specific information about the hardware architecture, the database and the algorithms are expressed and stored into libraries. In the same way, the system library will concentrate information about both the operating system (process creation, memory management, ...) and the transactional system (lock management, logging, commitment management, ...). We are currently working on the cost estimation of concurrency control and recovery mechanisms (for example, see [AGR85]). The system library is not yet fully specified.

8. Status Report

In this section, we report results obtained from the evaluation of the execution plan given in Figure 4. Then we present first results of a study on DeWitt's benchmark.

8.1. Example estimation

The execution plan was given to the cost evaluator for the two sample architectures presented in Figures 8 and 9. The estimate of the size of the query is 10 tuples. The operator and access method library formulas are based on formulas given in [AND90b] and [AND90c]. We made the number of processors for each architecture configuration vary. The results are summarized Figure 12.

Filter_Map = ALGO_BEGIN
ARCH_VAR (* imported variables from architecture library *)
C_i, C_p;
(* cost of a single instruction and of evaluating a simple predicate *)
SYS_VAR; (* imported variables from system library *)
Pcreat; (* Process creation cost (in instructions) *)
DB_VAR (* imported variables from database profile library *)
R.CARD, pred_est();
(* function to estimate the number of evaluated predicates per scanned tuple *)
OAM_VAR (* imported variables from operator and access method library *)
C_init;
(* cost of initializing a relation scan and of getting the first tuple *)
C_next; (* cost of getting the next tuple *)
QUERY_VAR (* information given by the input query *)
R,qualif; (* qualif : predicate expression *)
DEF
Cost fm_cpu = Pcreat * C_i
+ C_initscan (* Initialization cost to get the first tuple *)
+ C_scan; (* cost to get the next tuples *)
C_initscan = C_init * C_i + pred_est(qualif) * C_p;
C_scan = (R.CARD - 1) * (C_next * C_i + pred_est(qualif) * C_p);
ALGO_END

Figure 11. Filter_Map Illustration
8.2. Benchmark estimation

The Wisconsin Benchmark [BIT83b] is a well known and widely used benchmark for relational DBMSs, based on a precisely defined database and a set of standard queries. We are currently working on modeling the Wisconsin database with our tool and on using the benchmark to validate the environment descriptions. As an example, Table 13 shows the comparison of measured time for a Join (1000 * 10000 tuples) performed on the 6-processor DDC prototype [COUSS], [BER88] with our estimates.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measure (in seconds)</td>
<td>3.26</td>
<td>2.86</td>
<td>2.13</td>
<td>1.45</td>
</tr>
<tr>
<td>Estimation (in seconds)</td>
<td>3.40</td>
<td>3.01</td>
<td>2.30</td>
<td>1.53</td>
</tr>
</tbody>
</table>

Table 13: Result Table

9. Conclusions and Open Issues

In this paper, we have presented the design of a cost evaluator meeting the requirements of the parallel database management system developed in the EDS Esprit II project. We have pointed out a lack of generality and reusability in existing systems and we have proposed a complete solution that is original in several ways:

- The cost evaluator suits a multi-processor architecture;
- It is based on a multi-environment approach. The knowledge about the environment is declarative and divided into four libraries: architecture, database profile, operators and access methods, and system;
- This knowledge is expressed in an Environment Specification Language;
- It is a distinct tool not embedded within the optimizer. It can thus be used for other tasks (modeling, aid to data placement, ...);
- It offers the possibility of easy addition of new cost formulas, and the combination of several estimation methods;
- It follows the needs and methodology of the EDS DBMS development.

Our priority for future work is to gain experience with the cost evaluator in the EDS DBMS prototype and extensions of the LERA-PHY functionalities. Open research issues are:
- The adaptation of the tool to object-oriented database environments;
- The study of specific problems such as Fixpoint operator estimation [BOR90] and data placement;
- The study of a cost evaluator generator.

ACKNOWLEDGEMENTS

We thank Pascale Borla-Salamet, Björn Bergsten and Brian Hart for helpful discussions and useful suggestions.

REFERENCES


[GRA87] G. Graefe, D. DeWitt "The EXODUS Optimizer Generator", Computer Sciences Department, University Of Wisconsin, February 1987, Tech Report #687

[KEL89a] S.M. Kellett "LERA 1 - Language for Extended Relational Algebra" EDS.DD.111.0001, 1989

[KEL89b] S.M. Kellett "LERA Par - Parallel Lera Definition " EDS.DD.111.0002, October 1989


[MAC86b] L. Mackert and G. Lohman "R* Optimizer Validation and Performance Evaluation for Distributed Queries" 12th VLDB conference (Kyoto Japan Aug.).


ANNEX

Environment Specification Language

<environment> ::= <declaration> | <component specification> ;
<declaration> ::= <ARCH declaration> | <BD declaration> | <ALG declaration> ;
<component specification> ::= <ARCH component specification> | <BD component specification> | <SM component specification> | <DM component specification> ;
]<ARCH declaration> ::= <name> '=' 'ARCH-BEGIN'
| Arch body |
<Arch body> ::= <Interconnection Network>| '<Component List>' |
<Interconnection Network> ::= 'BUS <name>' |
| 'DELTA_NETWORK <name>' |
| 'ARCH component specification' ::= '<SM component specification> | <DM component specification> ;
| '<SM component specification> ::= '<mem-comp>' | '<cpu-comp>' | '<cache-comp>' |
| '<DM component specification> ::= '<mem-comp>' | '<cpu-comp>' | '<inter_net-comp>' |
| '<mem-comp>' ::= 'MEMORY <name>':'(MEMSIZE' '=' '<int>','MAT' '=' '<int>);'
| '<cpu-comp>' ::= 'CPU <name>':'(Cm' '=' '<int>','Ccomp' '=' '<int>','Scomp' '=' '<int>','Ck' '=' '<int>','Ce' '=' '<int>','C' '=' '<int>','Cj' '=' '<int>);'
| '<inter_net-comp>' ::= 'INTERCONNECTION <name>':'(PacSj' '=' '<int>','Pmt' '=' '<int>);'
| '<cache-comp>' ::= 'CACHE <name>':'(CACHEI' '=' '<int>','CAT' '=' '<int>);'
| <name> ::= (<character>)*;
| <int> ::= (<digit>)*;
Component List ::= SM <SMA Components> |
  DM <int> '*' (' MEMORY <name> ';' CPU <name> ')' |
<SMA Components> ::= GLOBAL MEMORY <name> ;
  [ <int> '*' (' CACHE <name> ';' CPU <name> ')' ] ;
<DB declaration> ::= DB_BEGIN |
  <relation> |<definition_part> |
  DB_END ;
<relation> ::= <name> <relation component> '(' <attribute> ')' ;
<attribute> ::= <name> <attribute component> <structure> ;
<structure> ::= INDX <index component> |
  HASHED <hash component> | NONE ;
<BD component specification> ::= <relation component> |
  <attribute component> |
  <index component> |
  <hash component> ;
<relation component> ::= |`
  NAME '=' <name> ','
  CARD '=' <int> ','
  SIZE '=' <int> ','
  ATT_NB '=' <int> ','
  MAX_BC_TU_NB '=' <int> ','
  BC_TU_NB '=' <int> ')' ;
<attribute component> ::= |`
  NAME '=' <name> ','
  DIFF VALUE CARD '=' <int> ','
  SIZE '=' <int> ','
  DISTRIBUTION '=' <distribution type> ','
  RELATION_NAME '=' <name> ','
  TYPE '=' <attribute type> ','
  <min & max values> ')' ;
<distribution type> ::= U | N ;
<attribute type> ::= I | R | S ;
<min & max values> ::= MAX VAL '=' <int> ','
  MIN VAL '=' <int> ;
<index component> ::= |`
  NAME '=' <name> ','
  SIZE '=' <int> ','
  ITEMS_NB '=' <int> ','
  VALUE_NB '=' <int> ')' ;
<hash component> ::= |`
  NAME '=' <name> ','
  <signature> ;
<signature> ::= <digit> ^ ;
<real> ::= <digit> ^ <digit> | <digit> | E | e | [+-] <digit> ;
<ALG declaration> ::= <name> = ALG_BEGIN |
  <ALGO body> |
  ALG_END ;
<ALGO body> ::= ARCH_VAR <name_list> |
  SYS_VAR <name_list> |
  DB_VAR <name_list> |
  OAM_VAR <name_list> |
  QUERY_VAR <name_list> |
  DEF <definition_part> * ;