ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

Rainer Manthey and Viacheslav Wolfengagen (Eds)

Advances in Databases and Information Systems 1997

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, (ADBIS'97), St Petersburg, 2-5 September 1997

A Modeling Tool for Workload Analysis and Performance Tuning of Parallel Database Applications

Silvio Salza and Massimiliano Renzetti

Published in collaboration with the
British Computer Society

Springer

ISBN: 3-540-76227-2

A Modeling Tool for Workload Analysis and Performance Tuning of Parallel Database Applications

Silvio Salza and Massimiliano Renzetti
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Roma, Italy
E-mail: salza@dis.uniroma1.it

Abstract

After about two decades of evolution parallel database architectures have finally reached a maturity level that makes them a serious competitor to traditional mainframe based architectures and distributed database systems. Dealing with these innovative systems requires to change considerably the application design process, mostly because of the sophisticated physical data organization (e.g. relation declustering), and, in general because of the more complex execution model. In this paper we present a methodology, and a connected modeling tool for the performance analysis of parallel database applications. The methodology is based on the results of previous work we have performed on this subject for traditional sequential DBMS, and is devised for a strict integration into the design procedure. The modeling approach allows both to provide the designer with an early feedback on the performance of the application, and a to produce a detailed account of the execution cost, that helps focusing the problem and gives a guide for the design improvement.

1 Introduction

Parallel database systems were originally introduced in the late '70s but have only recently reached their maturity as a commercial product. The last generation is mostly based on standard hardware and operating systems, and is an actual alternative to traditional mainframe based database architectures and to distributed systems based on networks of workstations, especially in some specific applications, like massive transaction systems, and, more recently data warehousing, which requires the evaluation of complex queries on very large databases.

Application designers and database administrators have therefore to face new problems connected with configuration and tuning of new applications. Actually this is partially a new edition of an old problem, since the original idea of Codd to make the design of a relational application largely independent from the details of the physical organization did not actually work out completely, even in the sequential environment. Additional problems arise in a parallel environment because of the more sophisticated physical data organization (e.g. relation declustering), and, in general, because of the more complex execution model.

The main problem, both in parallel and in sequential database systems, is that query execution plans, that actually affect the performance are generated by the optimizer and therefore are completely out of the designer's control. Therefore one has to get involved anyway with the physical level, both to detect where the performance problem is, and to set up a solution. Performance tuning may become an extremely painful process, especially if one has to wait for the final stages of the implementation before being able to trace the problem.

In this paper we present a methodology, and a connected modeling tool for the performance analysis of parallel database applications. The methodology is based on the results of previous work we have performed on this subject for traditional sequential DBMS, and is devised for a strict integration into the design procedure of parallel database applications. This is meant to provide both an early feedback on the performance of the application,and a detailed account of the workload and the execution cost, that help focus the problem and give

a guide for the design improvement. The approach we propose is quite general, but, of course, the tool was developed with reference to a specific parallel RDBMS, namely IBM DB2.

The paper is organized as follows. In the next section we briefly discuss the parallel execution model, i.e. the physical data organization and the way relational operators are executed. Then in Section 3 we give a sketch of the modeling methodology and we discusses how it integrates in the application design process. Section 4 introduces the workload model, i.e. the set of parameters that we use to give a quantitative characterization of the database (*static workload*), of the transactional workload (*dynamic workload*), and to represent the system configuration and the physical allocation of the database. Sections 5 and 6 discusses the evaluation of the transaction execution cost, that is far more complex than for sequential database, since relational operations may include preliminary phases to rearrange data and to ship intermediate results. In Section 7 we present a mathematical model of the bufferpool, which allows to take into account the effect of database buffering in reducing the disk I/O cost. Next in Section 8 we show how the designer may get from the cost analysis an important feedback to locate the system bottlenecks, and use this information as an input to the load balancing strategy. Section 9 discusses the evaluation of response times, another important performance index computed by the model. Finally conclusions are given in Section 10

2 The Parallel Execution Model

In designing our tool we have referred to DB2 *Parallel Edition* (DB2 PE), [2] an IBM product that runs on IBM SP2 parallel architecture [1]. This system has a typical MIMD architecture, i.e. is composed by the interconnection, via a fast interconnection network, of several largely independent systems, each with its own private memory and disks managed by a Unix operating system. Accordingly the parallel DBMS has a *shared nothing* architecture [22] where data are partitioned among the processors and stored on local disks, and queries are executed according to a *function shipping* philosophy, i.e. to perform database operations where the data reside.

More precisely each relation is *declustered* (i.e. horizontally partitioned) on a subset of processors called *node-group*. Declustering is performed by hashing the relation on a given attribute, which is called the *partition key*.

Relational operations performed on base relations are executed in parallel by all the nodes of a node-group, typically the one on which one (and possibly both) operands are declustered. Operations performed on intermediate results get their data from other operators by communicating in several different ways:

- *temporary file*: the intermediate result produced by a previous operator is materialized, declustered and stored on local disk;
- *pipeline*: two or more operators can be pipelined; the intermediate results are not written to disk, and the destination operator may start as soon as the first block of data is available;
- *table-queue*: two operators communicate through a FIFO file.

In the first two cases the source and the destination operators must be on the same node-group, but operators executed by different node-groups have always to communicate through table-queues.

There are several options for the execution of each relation operator, mostly depending from data placement. For instance a join can be performed in four different ways:

- *Collocated join*: is performed when the two relations are declustered on the same node-group, and the partition key is the join attribute. Since all the couples of joinable tuples must be on the same node, no data exchange is needed and all the nodes in the node-group may work in parallel.
- *directed join*: if only one relation is partitioned on the join attribute, then a preliminary step is performed to meet the conditions of the collocated join. The other relation is hashed on the join attribute and partitioned among the nodes of the node group of the first relation.

- *broadcast join*: in some cases, even if the conditions for a distributed join hold, it may be convenient, instead of partitioning the relation which is not collocated, to ship the whole relation to all the nodes of the nodegroup of the first relation. Then in each node the local partition of the first relation is joined with the other relation.
- *repartition join*: is the general case, when no special condition is met. Neither relation is partitioned on the join attribute, and, possibly, the operands are partitioned on different nodegroups. In this case a preliminary step of hash repartitioning is performed on both operands, and the results are shipped to a possibly different nodegroup. Then a collocated join is performed.

It is clear from the above that the execution cost and the degree of parallelism of a join strongly depend on the way the operation is performed. The difference between a collocated join and a repartition join may be dramatic. The former needs no data shipping and is performed completely in parallel on all the nodes of the node group, while the latter is composed of many steps and requires the shipping of the intermediate data and an higher degree of synchronization between the nodes (see the discussion in Section 6 for details).

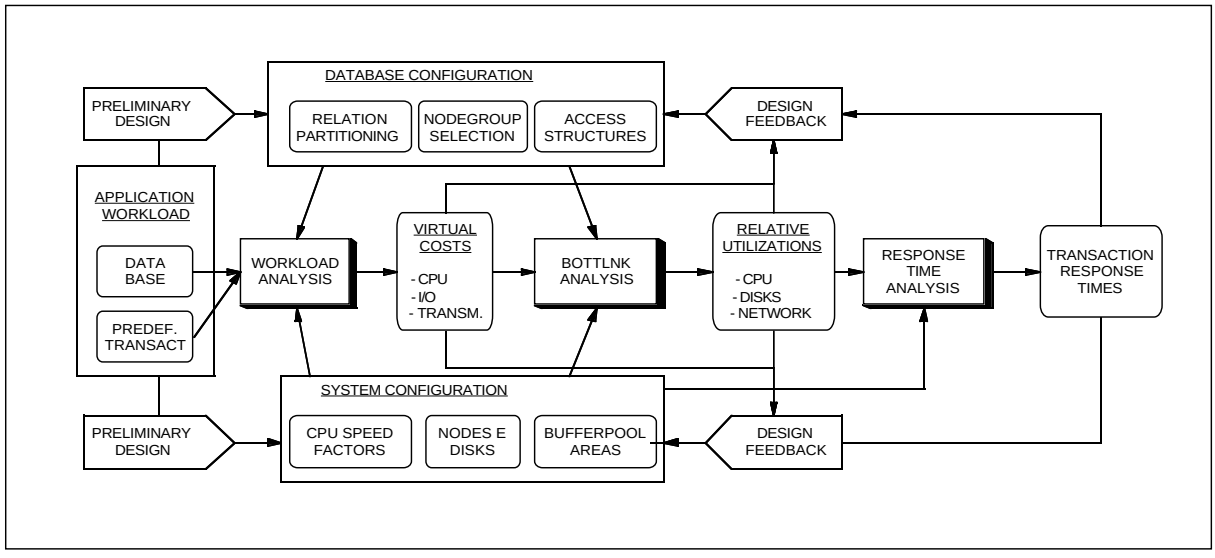

Figure 1: Application design and performance tuning

Therefore the query optimization problem in such an environment becomes both more crucial and considerably harder than in a sequential case. Much effort has been devoted in the literature to this topic [8, 11, 15, 16], though actually only specific aspects have been thoroughly analyzed, e.g. [4, 12].

The main point is that in the parallel model the solution space and the execution cost strongly depend on the data placement. Therefore one has to actually face two intimately connected optimization problems: query and data placement optimization. The application designer is in fact faced with the problem of selecting the best data allocation for a given workload of predefined queries.

The purpose of our modeling tool is indeed to provide the designer with an easy way to compare different design alternatives, and to have an analytical account of the execution cost that may guide him in the tuning and in the capacity planning.

3 Application Design and Performance Modeling

The aim of this section is to give an overall idea of our modeling methodology, and more specifically to explain how the methodology integrates with the application design procedure. This is meant to be just a preliminary

overview of the matter, to introduce the reader to the problem, and to clearly state the goals since the beginning. For a detailed discussion of the various topics one should instead refer to the following sections.

The main phases of the methodology are sketched in Figure 1. We actually do not include in our discussion the conceptual design phase, since we are only interested in the phases strictly related with the performance analysis. The starting point of our diagram (and of the design procedure) is therefore a description of the *application workload*. This represents the characterization of both the logical and quantitative aspects of the application: the logical database schema, the extensional characteristics of the tables (cardinality, attribute extension etc.), and a description of database accesses performed by the application.

As for this last component of the workload, that we shall call *dynamic workload*, we make the assumption of having *predefined* workload, i.e. that all the relevant accesses to the database belong to a set of predefined functions, and that the designer may take advantage of this information. This is indeed a reasonable assumption for some very important classes of applications. A classical case are transaction processing systems (e.g. in banking environments), where the final user may access the system only through a set of predefined transaction types, and the fraction of the load corresponding to each transaction type can be determined from system specifications. Another interesting example are large data warehouses where most of the load is given by the applications feeding the warehouse, and by the applications that extract data from the warehouse for predefined OLAP processing or to feed to second level warehouses. For sake of simplicity, we shall refer in the rest of the paper to the transaction processing case, and hence we shall call in the following sections *transaction types* the components of the dynamic workload.

On the other hand, the target of the design process are the database and system configuration, i.e. to determine the system configuration, the relation partitioning, the access structures, and, in general, how to allocate the functions and the data to the nodes of the parallel architecture.

The first phase of the design procedure, that we shall call *preliminary design* , produces, directly from the workload description a preliminary solution for the database and system configuration. This will be used as an input to the modeling methodology to get first level performance estimates, that, in turn, are used to refine the configuration.

The main phases of the methodology are represented in the diagram by the shaded boxes:

- *Workload analysis*: this phase produces from the workload description a detailed account of the execution cost of each component of the dynamic workload on each node of the parallel architecture, in terms of CPU, disk and transmission cost. These represent the unitary *net* execution costs (*virtual costs*) of each dynamic workload component, and do not take into account the device characteristics, the system congestion and the interference with other workload components. For instance, in a transactional system, virtual costs measure the impact on each single system resource of the execution of an instance of a given transaction. This may already provide the designer a first level of feedback to improve the database and system configuration.

- *Bottleneck analysis*: in this phase we compute from the virtual costs and the system configuration an estimate of the *relative resource utilizations*. This is done by taking into account the resource characteristics, e.g. CPU speed factors and disk access times, and the effect of I/O buffering. Relative resource utilizations does not depend on the workload *intensity* but only on the workload *profile*, e.g. on the transaction mix in a transaction processing system. This phase allows the designer to locate the system bottlenecks, and hence produces a very valuable design feedback for a load balancing strategy.

- *Response time analysis*: this phase produces response time estimates, typically a very important performance index, and a crucial reference for the designer. The modeling tool may give an analytical account of transaction response times, i.e. split them down in components corresponding to the different steps of the execution plans and resource involved. Again this provides the designer with valuable information for a further refinement of the design, i.e. of the database and system configuration.

We like to point out that, thanks to our *performance modeling* approach, the whole design cycle represented in the diagram may be completed long before the implementation of the system. This has evident advantages

```
CUSTOMERS  Card=75000          ITEMS      Card=3000000        SUPPLIERS  Card=5000
  Cust#    Char(4)    75000     Price    Num(8)        500     Supp#    Char(4)   5000
  Name     Char(25)   74950     Qty      Num(8)        100     Name     Char(25)  4995
  Phone    Char(15)   75000     C_date   Date         1825     Addr     Char(40)  5000
  Addr     Char(40)   75000     Status   Char(1)         4     Phone    Char(15)  5000
  Comm     Char(117)     80     Ord#     Char(4)    750000     Comm     Char(101)   30
  Nat#     Char(4)       20     Part#    Char(4)     99800     Nat#     Char(4)     15
- Index   CUST_I1(Name)          Supp#    Char(4)      4950   -Index    SUPP_I1(Name)
                               -Index    ITEMS_I1(Ord#)


ORDERS     Card=750000           PARTSUPP  Card=400000         PARTS      Card=100000
 Ord#      Char(4)    750000      A_qty    Num(4)       3500     Part#    Char(4)   100000
 Clerk     Char(15)     1000      S_cost   Num(8)      12000     Name     Char(55)    1200
 Price     Num(8)      75000      Price    Num(8)      25000     Part#    Char(4)   100000
 O_date    Date         1825      Supp#    Char(4)      5000     Descr    Char(23)    5000
 Status    Char(1)         4     -Index    PS_I1(Part#)        -Index     PART_I1(Name)
 Comm  T   Char(79)      120     -Index    PS_I1(Supp#)
 Cust#     Char(4)     75000
-Index    ORD_I1(Ord#)


REGIONS    Card=5                NATIONS  Card=25
 Reg#      Char(4)         5       Nat#     Char(4)     25
 Name      Char(25)        5       Name     Char(25)    25
 Comm      Char(152)       3       Comm     Char(152)   25
                                   Reg#      Char(4)    25
```

Figure 2: Sample static workload

```
Transaction T1         Rate = .20          Transaction T2       Rate = .15

SELECT PARTS.Name,ITEMS.Qty,ITEMS.Price    SELECT SUPPLIERS.Name,PARTSUPPS.S_cost,A_qty
FROM CUSTOMERS,ITEMS,ORDERS,PARTS          FROM PARTS, PARTSUPPS, SUPPLIERS
WHERE CUSTOMERS.Cust# = ORDERS.Cust# AND   WHERE PARTSUPPS.Supp# = SUPPLIERS.Supp# AND
      ITEMS.Ord# = ORDERS.Ord# AND                PARTS.Part# = PARTSUPPS.Part# AND
      ITEMS.PART# = PARTS.Part# AND               PARTS.Name = 'CPU'
      CUSTOMERS.Name = 'Smith' AND          ORDER BY PARTSUPPS.S_cost, SUPPLIERS.Name
      ITEMS.Price > 1000000 [.2]
ORDER BY PARTS.Name


Transaction T3         Rate = .35          Transaction T4      Rate = .30

SELECT SUPPLIERS.Name,CUSTOMERS.Name,
      NATIONS.Name                         SELECT SUPPLIERS.Name
FROM CUSTOMERS,NATIONS,SUPPLIERS           FROM NATIONS,REGIONS,SUPPLIERS
WHERE CUSTOMERS.Nat#=SUPPLIERS.Nat# AND    WHERE NATIONS.Nat#=SUPPLIERS.Nat# AND
      NATIONS.Nat#=SUPPLIERS.Nat#                NATIONS.Nat#=REGIONS.Nat# AND
                                                 REGIONS.Name='Eur' OR Name='Afr'
```

Figure 3: Sample dynamic workload

on the *performance monitoring* approach that requires to implement the system and run a benchmark before getting any kind of performance estimate.

Nevertheless the two approaches can be integrated by using performance measures to validate the model and refine the parameter estimates, and then again the model to compare configuration changes, and to perform a capacity planning analysis based on workload and system configuration evolution scenarios

4 The Workload Model

As already stated in Section 3, in our model we assume that all transactions arriving to the parallel DBMS belong to a set of *predefined transaction types*. To give a quantitative description of the application the user must therefore specify, and supply them as an input to the tool, the two main components of the workload:

- the *static workload*, composed of the logical schema of the database, of a set of parameters that summarize the physical extension of the relations and the statistical characteristics of the attributes, the specification of the relation declustering, and of the definition of the access structures;
- the *dynamic workload*, composed of the specification of a set of predefined transaction types and of their arrival rates;

More formally we define a *database* as a set of *relations*: $D = \{\mathcal{R}_i, i = 1, \ldots, N\}$. Each relation is a set of *tuples* $\mathcal{R}_i = \{r_{ij}, j = 1, \ldots c_i\}$, where c_i indicates the *cardinality* of the relation.

Each tuple r_{ij} is an ordered set of k_i values, where k_i is said the a-rity of the relation:

$$r_{ij} = \langle r_{ij}[1], \ldots, r_{ij}[k_i] \rangle \tag{1}$$

$$r_{ij}[h] \in V_i[h] \qquad h = 1, \ldots, k_i; \qquad j = 1, \ldots, c_i \tag{2}$$

$$\mathcal{R}_i[h] = \{r_{ij}[h]\} \qquad j = 1, \ldots c_i \qquad h = 1, \ldots, k_i \tag{3}$$

The multisets $\mathcal{R}_i[h]$, containing all the values assumed by a given field in the relation tuples, are called *attributes*, and the corresponding base sets $V_i[h]$ are called *value sets* and contain only distinct values.

For our purposes, a quantitative description of the database can be given by specifying the cardinalities $c_i, i = 1, \ldots, n$ of the relations, and the following parameters for each attribute $\mathcal{R}_i[h]$ in the database:

- $o_i[h] = card(V_i[h])$, called the attribute *originality*, which represents the number of distinct values in $\mathcal{R}_i[h]$;
- $\lambda_i[h]$, called the attribute *extension*, which represents the number of bytes needed to store each element of $\mathcal{R}_i[h]$.
- $\tau_i[h] \in \{char, number, date\}$, called the attribute *type*, which represents its data type.
- $\nu_i[h]$, which represents the expected fraction of *null values* in $\mathcal{R}_i[h]$.

Moreover to represent the coupling between attributes we must specify for every couple of *union-compatible* attributes the *overlapping factor* :

$$w_{i,h}^{j,k} = \frac{card(V_i[h] \cap V_j[k])}{card(V_i[h])} \tag{4}$$

which gives the percentage of values that are common to both attributes.

This characterization is simple enough to allow reasonably the designer to estimate the parameter values even during early phases of the design, and on the other hand it is sufficient, as we shall discuss later, to compute the execution cost of any relational operation performed directly on the base relations. However, dealing with

```
LINEITEMS     on   NG0   key  Ord#     skew=.30, node 7
NATIONS       on   NG2
ORDERS        on   NG0   key  Ord#
PARTS         on   NGB   key  Part#
PARTSUPPS     on   NGB   key  Part#    skew=.40, node 8
REGIONS       on   NG2
SUPPLIERS     on   NGA   key  Supp#    skew=.70, node 3
```

Figure 4: The allocation map

```
CPU-MEMORY:                          DISK  A:
   Speed Factor   1.00                 Size                1 GB
   Overhead        .05                 Service time       16 ms
   Bufferpool     1000 blocks          Block size       4096 byte
   Sortheap       1000 blocks          Overhead           .1
   Segments         16                 Segments            6

NET                                   DISK  B:
   Transm. time       10  ms            Size                2 GB
   Block size       4096 byte           Service time       24 ms
   Overhead          .05                Block size       4096 byte
                                        Overhead           .15
                                        Segments            10
```

Figure 5: Processing node configuration

complex queries requires to get estimates of the parameters for the intermediate relations as well. This can be performed using a methodology that we have proposed in previous papers [6, 19].

Figure 3 shows a listing produced by the tool from the parameters supplied by the user for the static workload of a sample application, a typical customer-supplier-part database, we actually used to test the tool, and to which we will refer all along the paper. Relation schemata are shown together with the cardinality, attribute originalities and the specification of indices. The allocation map is shown in Figure 4, referring to a system of 8 nodes with node-groups $\mathrm{NG}_0 = \{1 \ldots 8\}$, $\mathrm{NG}_A = \{3, 4\}$, $\mathrm{NG}_B = \{5, 6, 7, 8\}$ and $\mathrm{NG}_i = \{i\}$. The configuration of a node is shown in Figure 5.

For the dynamic workload, as mentioned before, we assume that all transactions arriving to the system belong to a fixed set of predefined *transaction types* $\mathcal{Q} = \{\mathcal{T}_i, i = 1, \ldots, m\}$, for which all the details except some constants are specified. The user is therefore required to supply for every transaction type $\mathcal{T}_i$ the following information:

- the *relative arrival rate* a_i;
- a SQL definition of the transaction type;
- the expected selectivity of each atomic predicate;
- the CPU and I/O overhead of the *application part* of the transaction;

Again this information can reasonably be derived by the designer, from the user specification of the application. Perhaps the most delicate part is estimating the selectivities, since these may have a considerable impact on the execution cost.

The transaction set of the sample application is reported in Figure 3. The user estimates of the selectivities for the atomic predicates are printed in square brackets. No selectivities are specified for equality predicates, since, assuming a uniform distribution, they are directly computed by the model from the database parameters.

5 Virtual Execution Costs

We consider two components of the execution cost: the *storage cost*, i.e. the secondary memory layout and the storage requirements of the database (tables and indices), and the transaction execution cost (CPU, I/O and transmission cost), for every transaction type in the set $\mathcal{Q}$.

To provide an accurate specification of the I/O cost, in terms of disk access rates, we consider the tables and the indices to be partitioned in *homogeneous data segments*. These are collections of blocks belonging to the same table or index such that all the blocks in the same segment have the same probability to be accessed during the execution of the application, for example the blocks at the same level of a B-tree index. The partition in segments allows to compute the actual I/O cost, taking into account the uneven effect of DBMS buffering on segments with different access rates.

At a first level of analysis we want simply to characterize the transaction execution cost in terms of *resource demands*, expressed independently from the system configuration and the workload profile (arrival rate and transaction mix). We call these *virtual costs*. These include the CPU service time estimated on a reference processor, the number of *logical* disk accesses, i.e. computed without taking into account the buffering.

Virtual costs are computed with reference to the *parallel execution plan*, which represents the sequence of concurrent tasks performed by the parallel systems during the evaluation of the query. The execution tree is actually generated by the *query optimizer* (traditionally the most *mysterious* parts of a RDBMS), according to a complex strategy, mostly driven by the data placement and the availability of indices. To *mimic* that strategy in the modeling tool has in fact given us a few technical problems, and has required intensive checking against the DBMS.

A sample parallel execution plan is shown in Figure 6 for transaction T1. The leaves represent base relations and the nodes parallel operators, each labeled with the node-group on which it is executed. The arcs represent the flow of data between operators, and their label specifies the kind of connection.

6 Cost Evaluation

To understand how execution costs may be evaluated let us consider a rather general example, i.e. a *repartition join* (see Section 2) $T = R \bowtie S$ between two base relations R and S partitioned on different nodegroups NG_R and NG_S, with the operation performed on third nodegroup NG_J and with the result delivered to a fourth nodegroup NG_T. The execution of the repartition join consists of the following steps:

1. Sequential scan of each partition R_i of R performed in each node of NG_R, to evaluate a selection predicate on the attributes of R.

2. Local merge-sort on the join attributes of each partition R_i of R performed in each node of NG_R.

3. Hash repartitioning of R on the join attributes to distribute the tuples of R on the nodegroup NG_J where the join has to be executed. This step involves costs for both nodegroups NG_R where the hashing has to be performed and NG_J where the blocks are shipped and then stored. Therefore we shall consider two substeps 3a and 3b.

4. Sequential scan of each partition S_i of S performed in each node of NG_S, to evaluate a selection predicate on the attributes of S.

5. Local merge-sort on the join attributes of each partition of S performed in each node of NG_S.

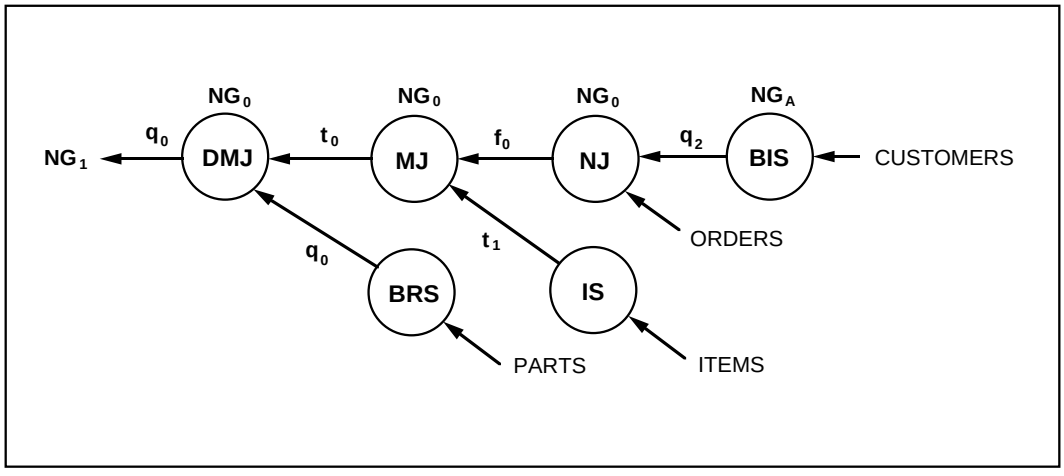

Figure 6: Parallel query execution plan

step#	*step*	*n-groups*	*I/O*	*CPU*	*NET*
1	SCAN(R)	NG_R	$C_i^{\mathrm{I/O}}(\mathrm{SCAN}(R),\mathrm{NG}_R)$	$C_i^{\mathrm{CPU}}(\mathrm{SCAN}(R),\mathrm{NG}_R)$	—
2	SORT(R)	NG_R	$C_i^{\mathrm{I/O}}(\mathrm{SORT}(R),\mathrm{NG}_R)$	$C_i^{\mathrm{CPU}}(\mathrm{SORT}(R),\mathrm{NG}_R)$	—
3a	HASH(R)	NG_R	—	$C_i^{\mathrm{CPU}}(\mathrm{HASH}(R),\mathrm{NG}_R)$	$C_i^{\mathrm{NET}}(\mathrm{HASH}(R),\mathrm{NG}_R)$
3b	HASH(R)	NG_J	$C_i^{\mathrm{I/O}}(\mathrm{HASH}(R),\mathrm{NG}_J)$	$C_i^{\mathrm{CPU}}(\mathrm{HASH}(R),\mathrm{NG}_J)$	—
4	SCAN(S)	NG_S	$C_i^{\mathrm{I/O}}(\mathrm{SCAN}(S),\mathrm{NG}_S)$	$C_i^{\mathrm{CPU}}(\mathrm{SCAN}(S),\mathrm{NG}_S)$	—
5	SORT(S)	NG_S	$C_i^{\mathrm{I/O}}(\mathrm{SORT}(S),\mathrm{NG}_S)$	$C_i^{\mathrm{CPU}}(\mathrm{SORT}(S),\mathrm{NG}_S)$	—
6a	HASH(S)	NG_S	—	$C_i^{\mathrm{CPU}}(\mathrm{HASH}(S),\mathrm{NG}_S)$	$C_i^{\mathrm{NET}}(\mathrm{HASH}(S),\mathrm{NG}_S)$
6b	HASH(S)	NG_J	$C_i^{\mathrm{I/O}}(\mathrm{HASH}(S),\mathrm{NG}_J)$	$C_i^{\mathrm{CPU}}(\mathrm{HASH}(S),\mathrm{NG}_J)$	—
7	JOIN(R,S)	NG_J	$C_i^{\mathrm{I/O}}(\mathrm{JOIN}(R,S),\mathrm{NG}_J)$	$C_i^{CPU}(\mathrm{JOIN}(R,S),\mathrm{NG}_J)$	—
8a	HASH(T)	NG_J	—	$C_i^{\mathrm{CPU}}(\mathrm{HASH}(T),\mathrm{NG}_J)$	$C_i^{\mathrm{NET}}(\mathrm{HASH}(S),\mathrm{NG}_S)$
8b	HASH(T)	NG_T	$C_i^{\mathrm{I/O}}(\mathrm{HASH}(T),\mathrm{NG}_T)$	$C_i^{\mathrm{CPU}}(\mathrm{HASH}(T),\mathrm{NG}_T)$	—

Figure 7: Elementary execution costs in a repartition join

6. Hash repartitioning of S on the join attributes to distributes the tuples of S on the nodegroup NG_J where the join has to be executed. This step involves costs for both nodegroups NG_S where the hashing has to be performed and NG_J where the blocks are shipped and then stored. Therefore we shall consider two substeps 6a and 6b.
7. A local merge-join on each node of NG_J where the two operands have been repartitioned.
8. Hash repartitioning of the result relation T on the destination nodegroup NG_T. This step involves costs for both nodegroups NG_J and NG_T, therefore in this case too we shall consider two substeps 8a and 8b.

This is indeed a quite general case since a join is performed in this way usually only when no special condition occurs. Therefore some of the above steps may vanish if, for instance, the relations are already sorted, there are no selection predicates, or some of the nodegroups involved are the same.

A schema of the costs connected to the various steps is presented in Figure 7.

6.1 Disk I/O Costs

During the scan of step 1 in each node $i \in \mathrm{NG}_R$ the corresponding partition R_i of relation R is read from disk. Therefore the I/O cost (virtual disk accesses) is:

$$C_i^{\mathrm{I/O}}(\mathrm{SCAN}(R), \mathrm{NG}_R) = B_{R_i} \tag{5}$$

where B_{R_i} denote the number of blocks in partition R_i, and can easily be computed from relation cardinality c_R, the nodegroup cardinality, the attribute extensions and the disk block size b.

The result of the selection performed in step 1 on node i is a partition R_i^{SCAN} with a number of blocks:

$$B_{R_i^{\mathrm{SCAN}}} = \left\lceil \frac{\sigma_{\pi_R} \;\; c_{R_i} \;\; t_{R^{\mathrm{SCAN}}}}{b} \right\rceil \tag{6}$$

where σ_{π_R} is the selectivity of the predicate on R, and $t_{R^{\mathrm{SCAN}}}$ is the tuple extension of R^{SCAN} and b the disk block size.

Step 2 consists on the local merge-sort of partition R_i^{SCAN} on each node of NG_R. The number of 'sorted runs' depends on the number of pages h in the sort-heap area (a system configuration parameter):

$$\mathrm{SR} = \left\lceil \frac{B_{R_i^{\mathrm{SCAN}}}}{h} \right\rceil \tag{7}$$

and therefore the total I/O cost of this step for node i is given by:

$$C_i^{\mathrm{I/O}}(\mathrm{SORT}(R), \mathrm{NG}_R) = 2\ \mathrm{SR}\ \lceil \log_h \mathrm{SR} - 1 \rceil \tag{8}$$

During the hash repartition phase of step 3b the result of step 2 R_i^{SORT} is hashed and shipped from each node of NG_R to the nodes of NG_J, and there stored. Therefore, assuming an uniform distribution of the tuples on the nodes of NG_J, the I/O cost for each of them is equal to the number of blocks in the corresponding partition $B_{R_i^{\mathrm{HASH}}}$:

$$C_i^{\mathrm{I/O}}(\mathrm{HASH}(R), \mathrm{NG}_J) = \left\lceil \frac{\sum_{j \in \mathrm{NG}_R} \mathrm{card}(R_j^{\mathrm{SORT}})}{b \;\; \mathrm{card}(\mathrm{NG}_J)} \right\rceil \tag{9}$$

The I/O cost of steps 4, 5, and 6b is given by formulas identical to the (5)-(9) but with R changed to S.

Step 7 is simply a local merge-join on the sorted relation partitions, and hence has an linear I/O cost for each node of NG_J:

$$C_i^{\mathrm{I/O}}(\mathrm{JOIN}(R, S), \mathrm{NG}_J) = B_{R_i^{\mathrm{HASH}}} + B_{S_i^{\mathrm{HASH}}} \tag{10}$$

Finally, similarly to step 3b, the hash repartition on the destination nodegroup has for each node $i \in \mathrm{NG}_T$ an I/O cost equal to the number of blocks $B_{T_i^{\mathrm{HASH}}}$ in the corresponding partition:

$$C_i^{\mathrm{I/O}}(\mathrm{HASH}(T), \mathrm{NG}_T) = \left\lceil \frac{\sum_{j \in \mathrm{NG}_J} \mathrm{card}(T_j)}{b \ \mathrm{card}(\mathrm{NG}_T)} \right\rceil \tag{11}$$

where $\mathrm{card}(T_j)$ is cardinality of the result of the local join in node $j \in \mathrm{NG}_J$, and can be computed from the cardinalities of the operands and the join selectivity (refer to [6] and [19] for details).

6.2 CPU Costs

CPU cost estimates can in general be computed directly from the I/O cost, by assuming that the processing cost is proportional to the number of blocks read and written during the operation, and to the number of tuples compared through equality or selection predicates.

For instance in the selection scan in step 1 the CPU cost is proportional to the number of blocks of R_i scanned and to the number of tuples of R_i, since for each tuple the selection predicate must be evaluated:

$$C_i^{\mathrm{CPU}}(\mathrm{SCAN}(R), \mathrm{NG}_R) = t_{\mathrm{I/O}} \ B_{R_i} + t_{\mathrm{pred}}(\pi_R) \ c_{R_i} \tag{12}$$

where $t_{\mathrm{I/O}}$ is the CPU overhead of a disk block read or write, and $t_{\mathrm{pred}}(\pi_R)$ is the CPU time needed to evaluate the selection predicate on a single tuple.

Similarly for the merge-sort in step 2 the CPU cost is proportional to the number of block reads and writes and to the number tuples compared during the sort and merge phases. Therefore according to the (6) and (7):

$$C_i^{\mathrm{CPU}}(\mathrm{SORT}(R), \mathrm{NG}_R) = t_{\mathrm{I/O}} \ C_i^{\mathrm{I/O}}(\mathrm{SORT}(R), \mathrm{NG}_R) + t_{\mathrm{conf}}(\pi_J) \ (\lceil \log_h \mathrm{SR} \rceil \ \mathrm{card}(R_i^{\mathrm{SCAN}}) + B_{R_i^{\mathrm{SCAN}}} \ n^2_{R^{\mathrm{SCAN}}}) \tag{13}$$

where $t_{\mathrm{conf}}(\pi_J)$ is the CPU time needed to compare two tuples of R^{SCAN}, and $n_{R^{\mathrm{SCAN}}}$ is the number of tuples of R^{SCAN} per block.

For the repartition join in substep 3a, for each node in NG_R the CPU cost of writing the result into the table queue is:

$$C_i^{\mathrm{CPU}}(\mathrm{HASH}(R), \mathrm{NG}_R) = t_q \ B_{R_i^{\mathrm{SORT}}} \tag{14}$$

where t_q is the CPU cost of writing a block in a table queue. Moreover for substep 3b for each node in NG_R the CPU cost of writing its partition to disk is:

$$C_i^{\mathrm{CPU}}(\mathrm{HASH}(R), \mathrm{NG}_J) = t_{\mathrm{I/O}} \ B_{R_i^{\mathrm{HASH}}} \tag{15}$$

As for the I/O cost, the CPU cost of steps 4, 5, 6a and 6b is given by formulas identical to the (12)-(15) but with R changed to S.

For the merge-join of step 7 we have two components: the CPU overhead for reading the two hashed relations from disk, and the processing time needed to compare the tuples :

$$C_i^{\mathrm{CPU}}(\mathrm{JOIN}(R, S), \mathrm{NG}_J) = t_{\mathrm{I/O}} \ (B_{R_i^{\mathrm{HASH}}} + B_{S_i^{\mathrm{HASH}}}) + t_{\mathrm{conf}}(\pi_J) \ (\mathrm{card}(R_i^{\mathrm{HASH}}) + \mathrm{card}(S_i^{\mathrm{HASH}})) \tag{16}$$

Finally, similarly to step 3, in step 8 (the hash repartition on the destination nodegroup) we have for each node in NG_J, the CPU cost of writing the result into the table queue, and for each node in NG_T the CPU cost of writing its partition to disk:

$$C_i^{\mathrm{CPU}}(\mathrm{HASH}(T), \mathrm{NG}_J) = t_q \ B_{T_i} \tag{17}$$

$$C_i^{\mathrm{CPU}}(\mathrm{HASH}(T), \mathrm{NG}_T) = t_{\mathrm{I/O}} \ B_{T_i^{\mathrm{HASH}}} \tag{18}$$

6.3 Transmission Costs

According to the table in Figure 7 there are transmission costs only for steps 3a, 6a and 8a, i.e. for the hash repartition steps. This is indeed in full agreement with the basic philosophy of the shared-nothing architecture, which is substantially based on function shipping and tries to minimize data shipping. Therefore, unless for very large configurations and very inappropriate choices in database design, the communication network seldom becomes a system bottleneck.

Nevertheless transmission costs, that are measured in blocks sent through the interconnection network, can easily be evaluated since they are strictly proportional to the size of the results of the corresponding steps.

For instance, for step 3a the transmission cost for each node in NG_R is:

$$C_i^{\mathrm{NET}}(\mathrm{HASH}(R), \mathrm{NG}_R) = B_{R_i^{\mathrm{SORT}}} \tag{19}$$

Similar expressions, *mutatis mutandis*, can be written for steps 6a and 8a.

7 The Buffer Model

On each node the buffer is managed with a global LRU policy. Therefore the data blocks with high global access rate tend to reside in it. Actually, all the blocks of the same homogeneous data segment have the same access rate, and then the same probability to be in the buffer. Therefore for a given workload, and hence for a given access pattern of the data blocks, to compute the number of physical accesses, we need to evaluate for each segment the *buffer hit ratio*, that is the equilibrium probability that a block of the segment is found in the buffer.

More formally, let us refer to a node with buffer of n_b blocks, and to a set of n_s homogeneous data segments, with global access rates f_i, and sizes B_i (in blocks). Let now R_i be the *resident set size* of segment i, i.e. the average number of blocks in the buffer, and let α_i and ω_i be respectively the rates at which the blocks of segment i enter and leave the buffer. In an equilibrium condition the number of resident pages of every segment is constant, and then the two rates α_i and ω_i must equate:

$$\alpha_i = f_i \frac{B_i - R_i}{B_i} \tag{20}$$

Similarly ω_i depends on the resident set size of the segment, and on the global replacement rate:

$$\omega_i = \frac{R_i}{N_b} \sum_{j=1}^{n_s} \omega_j \tag{21}$$

Then substituting the (20) in the (21) and considering that $\alpha_i = \omega_i$ we get:

$$f_i \frac{B_i - R_i}{B_i} = \frac{R_i}{N_b} \sum_{j=1}^{n_s} f_j \frac{B_j - R_j}{B_j} \tag{22}$$

The (22) are a set of nonlinear equations in the unknowns R_i. An approximate solution to the system can then easily be computed with the iterative formulas:

$$\hat{R}_i[0] = N_b \frac{B_i}{\sum_{j=1}^{N_s} B_j} \tag{23}$$

$$\hat{R}_i[k] = \frac{f_i B_i N_b}{f_i N_b + Bi \sum_{j=1}^{N_s} f_j \frac{B_j - \hat{R}_j[k-1]}{B_j}} \tag{24}$$

From a practical point of view, the experience shows that the iteration converges quickly to the solution of the system, and hence we may compute the hit ratios, that can be expressed as the ratio between R_i and B_i.

8 Bottleneck Analysis

The execution costs computed by the tools for the sample application are shown in Figure 8. CPU costs are given in seconds and I/O and transmission costs in block accesses. Logical costs are shown in parenthesis. Cost are split by node and by transaction type, and represent the service demand of a single transaction to each resource in the system. Adding up the demands to each resource, weighted by transaction arrival rate, gives the relative utilizations printed in the last column. Relative utilizations are normalized so that the *bottleneck* resource has utilization 1.

NODE	RES.	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$	UTIL.
1	CPU	.998	.019	.043	.003	.041
1	D1.A	4302 (4676)	90 (90)	204 (204)	12 (12)	.246
1	D1.B	7171 (7793)	151 (151)	341 (341)	20 (20)	.614
2	CPU	.952	.007	.001	.001	.031
2	D2.A	3060 (3569)	33 (33)	1 (1)	1 (1)	.148
2	D2.B	5101 (5948)	56 (56)	1 (1)	1 (1)	.370
3	CPU	.762	.022	.306	.013	.041
3	D3.A	3329 (3571)	78 (93)	1128 (1260)	46 (61)	.259
3	D3.B	5548 (5951)	1 (156)	1880 (2100)	76 (101)	.651
4	CPU	.761	.014	.163	.005	.032
4	D4.A	3277 (3570)	51 (59)	588 (671)	19 (27)	.210
4	D4.B	5464 (5950)	87 (99)	982 (1118)	33 (44)	.525
5	CPU	.843	.022	.000	.000	.035
5	D5.A	3405 (3777)	54 (62)	0 (0)	0 (0)	.150
5	D5.A	5676 (6296)	89 (102)	0 (0)	0 (0)	.660
6	CPU	.843	.022	.000	.000	.035
6	D6.A	3405 (3777)	54 (62)	0 (0)	0 (0)	.150
6	D6.B	5676 (6296)	89 (102)	0 (0)	0 (0)	.660
7	CPU	1.869	.022	.000	.000	.056
7	D7.A	8214 (8588)	57 (62)	0 (0)	0 (0)	.555
7	D7.B	13693 (14314)	94 (102)	0 (0)	0 (0)	1.000
8	CPU	.843	.140	.000	.000	.031
8	D8.A	3406 (3777)	54 (62)	0 (0)	0 (0)	.165
8	D8.B	5676 (6296)	89 (102)	0 (0)	0 (0)	.412
NET		1474	41	547	33	.004

Figure 8: Execution costs

The cost analysis gives very valuable feedback to the designer. First it allows to locate the bottleneck (disk D7.B in the example), and to compute the maximum overall transaction arrival rate (0.015 trans/sec in the example). More in general it shows which resources have problems and which transactions are responsible for them. A more detailed account is also produced by the tool that gives the resource demands of each operator in a query execution plan.

All this information can be utilized by the designer to take the appropriate actions in reconfiguring the system. For instance in the case of Figure 8, that shows a very unbalanced I/O bound situation, appropriate actions could be adding disks, extending buffer sizes, changing the file allocation and modifying the node-group configuration. As the configuration is changed the tool can immediately compute the new costs.

9 Transaction Response Times

From the execution costs the tool computes an estimate of the transaction response time, for a given workload intensity, i.e. for a given overall transaction arrival rate. The system is modeled as a multiclass open queueing network, which is solved analytically. We have used a hierarchical modeling approach which takes into account the cross interaction between transactions executed concurrently through slow-down factors computed from

resource utilizations.

The transaction response time is computed from the parallel execution tree (e.g. the one in Figure 6), from the average elapsed times of the elementary concurrent tasks performed by the processing nodes, that we shall call *atomic tasks*. The tree is visited from the leaf to the root, to compute the relevant times connected to the execution of each parallel operator:

- the time when the execution of the operator begins;
- the time when the first elementary block of data of the result is produced;
- the time when the execution of the operator terminates.

To do this one must take into account two different levels of parallelism:

- *inter-operator parallelism*: the overlapping between the execution of different operators, e.g. because of pipeline or table-queue communication;
- *intra-operator parallelism*: the parallel execution of a given operator by all the nodes of the node-group.

For both levels of parallelism we used a stochastic modeling approach, because we considered a crucial point to model the time spread in the duration of the atomic tasks.

Λ	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$
.005	644.0	4.4	77.7	3.2
.010	1148.9	7.9	101.9	4.1
.014	28260.9	251.4	154.2	6.2

Figure 9: Transaction response times

For the inter-operator parallelism we explicitly model the connection between a *source operator* that stochastically delivers elementary blocks of data to a *destination operator*, and therefore we compute an *average overlapping factor* between the two.

For the intra-operator parallelism instead we model the concurrent execution of the atomic tasks by the processors in the node-group (which may have different utilizations) as a composed Poisson process, since the more nodes are in the node-group the larger will be the spread in the total execution time of the operator, and in the delivery time of the first block of the result.

Transaction response times for the sample application are shown in Figure 9 for three different overall arrival rates. As one would expect from the discussion on the execution costs in the previous section, response times for the arrival rate of .014 are very high since this value is quite near to the maximum rate, and in all cases transaction T1 has a response time considerably worse than the other transactions.

The response time estimates computed by the model are indeed of great help for the designer since they give, even in the early phases of the design, an important feedback on a very crucial performance index. Even if affected by some degree of approximation, such information is indeed very valuable in selecting the proper system configuration and carrying on a capacity planning study.

10 Conclusions

In this paper we have presented a systematic methodology for the performance analysis of parallel database applications. There are at least two good reasons for incorporating the modeling approach in the design procedure. First, performance evaluation is a far more crucial problem in parallel RDBMS than it is for sequential RDBMS, since high performance is the most likely reason why a parallel architecture was selected.

Then a parallel database application *has* to be performing. Second with these innovative architectures designers and DB administrators have to deal with a far more complex physical organization and execution model, and this makes in some cases very difficult to predict the quantitative effect of their design decisions. Therefore it is very important to be able to get some feedback in the early stages of the design, even before the application is implemented.

Three kinds of performance measures are produced by the model:

- execution costs: i.e. the resource demands by every transaction to system resources; this may suggest changes in the physical organization (i.e. a different relation partitioning or node-group definition);
- resource utilizations: these allow to locate the bottlenecks and are of crucial help in improving the system configuration (bufferpool area size, disk allocation etc.), and in implementing load balancing strategies;
- transaction response times: these are end-user performance indices, and may be checked against the original design specifications or used in a capacity planning study.

The modeling task has proved to be considerably more complex than in the classical case of sequential RDBMS, and several new specific problems had to be solved. The most interesting parts are the bottleneck analysis and the evaluation of transaction response times that are thoroughly discussed in the paper.

To validate our methodology and the modeling tool, we have run a series of experiments to compare, for several system configurations, the performance estimate computed by the tool, with the measures taken on the running application. For each experiment a stream of a few hundred transactions with Poisson interarrival times was first generated off line and recorded. This was done according to the transaction set and the relative arrival rates of Figure 3, and randomly selecting the constants in the queries, using trapezoidal distributions that complied with the elementary selectivities. The experiment consisted in running a set of processes, that connected to the DBMS to simulate user sessions, and passed to it the transactions of the stream. An additional process took a record of the response times and the other performance measures. The experimental overhead turned out to account for less than 5% of the CPU, and for a small number of I/O accesses. This was taken into account in the model as system overhead. Preliminary results show a good agreement between the measures and the performance estimates produced by the tool.

References

[1] T. Agerwala et alii. SP2 system architecture. *IBM System Journal*, 34(2):152–184, 1995.

[2] C.K. Baru et alii. DB2 Parallel Edition. *IBM System Journal*, 34(2):292–321, 1995.

[3] D. Bitton, H. Boral, D.J. DeWitt. Parallel algorithms for the execution of relational database operations. *ACM Trans. on Database Syst.*, 8(3): 324–353, 1983.

[4] P. Borla-Salamet, M. Zait, M. Ziane. Parallel query processing in DBS3. In *Proc. of Parallel and Distributed Inforamtion System Conf.*, San Diego, Jan. 1993, pp.93–102.

[5] F. Carino,P. Kostamaa. Exegesis of DBC/1012 and P-90 - industrial supercomputer database machines. In *Proc. of Intl. Conf. on Parallel Architectures and Languages Europe*, pp. 877–892, 1992.

[6] F. Cesarini, S. Salza eds. *Database Machine Performance: Modeling Methodologies and Evaluation Strategies.* Lecture Notes in Computer Science n. 257, Springer-Verlag, Berlin 1987.

[7] M.S. Chen, M.L Lo, C.V. Ravishankar, P.S. Yu. On optimal processor allocation to support pipelined hash joins. In *Proc. of ACM SIGMOD Conf.* 1993, pp. 185–196.

[8] M.S. Chen, H. Hsiao, P.S. Yu. On parallel execution of multiple pipelined hash joins. In *Proc. of ACM SIGMOD Conf.,*, Portland, 1989, pp. 185–196.

[9] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. of Eleventh International Conference on Very Large Data Bases*, Stockolm, 1985, pp. 127–141.

[10] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of ACM SIGMOD Conf.*, 1990, pp. 102–111.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[12] W. Hong, M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

[13] R. Katz,J. Ousterout,D. Patterson,M. Stonebraker. The design of XPRS. In *Proc. of Int. Conf. on Very Large Databases*, 1993, pp. 55–68.

[14] H. Lakshmi and P.S. Yu. Effectiveness of parallel joins. *IEEE Transactions on Knoledge and Data Engineering*, 2(4): 410–424, 1990.

[15] R.S. Lanzelotte, P. Valduriez, M. Zait. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proc. of Int. Conf. on Very Large Databases*, Dublin 1993, pp. 493–504.

[16] H. Lu, K.L. Tan. On resource scheduling on multi-join queries in parallel database systems. *Information Processing Letters*, 48(4): 189–195, 1993.

[17] L. Mackert and G. Lohman. Index scan using a finite lru buffer: A validated I/O model. *ACM Trans. on Database Syst.*, 14: 401–424, 1989.

[18] G. Sacco and Schkolnick. Buffer management in relational database systems. *ACM Trans. on Database Syst.*, 11: 473–498, 1986.

[19] S. Salza, M. Terranova. Evaluating the size of queries on relational databases with non uniform distribution and stochastic dependence. In *Proc. of ACM SIGMOD Conf.,*, Portland, 1989, pp. 8–14.

[20] S. Salza, R. Tomasso. A Modeling Tool for the Performance Analysis of Relational Database Applications. In *Computer Performance Evaluation: Modeling Techniques and Tools*, R. Pooley and J. Hillston eds., Edimburgh University Press, pp. 247–259, 1992.

[21] M. Stonebraker. The Case for Shared-Nothing. *Database Engineering*, (9)1, 1986.

[22] P. Valduriez. The case of shared something. In *Proc. of IEEE Conf. on Data Engineering*, 1993, pp. 460–465.