
Dynamic Memory Allocation for Large Query Execution

**Luc Bouganim^{*,**} — Olga Kapitskaia^{**}
Patrick Valduriez^{**}**

^{*}PRiSM Laboratory, Versailles, France
Luc.Bouganim@prism.uvsq.fr

^{**}INRIA Rocquencourt, Le Chesnay, France
Olga.Kapitskaia@inria.fr
Patrick.Valduriez@inria.fr

ABSTRACT. *The execution time of a large query depends mainly on the memory utilization which should avoid disk accesses for intermediate results. Poor memory management can hurt performance and even lead to system thrashing because of paging. However, memory management optimization is hard to incorporate in a query optimizer because of cost estimate errors. In this paper, we address the problem of efficient memory management for large query execution. We propose a static memory allocation scheme applied at start-up time, and a more efficient dynamic execution model which performs memory-adaptive scheduling of the query. Our execution model handles gracefully memory overflow by choosing dynamically the best scheduling among several possible using a simple cost model. The model is robust to cost estimate errors. We describe a performance evaluation using a prototype implementation. The experiments with many queries show significant gain over static strategies.*

KEY WORDS: *Memory Management, Query Execution, System Paging*

RÉSUMÉ *Le temps d'exécution de requêtes complexes dépend principalement de la gestion de la mémoire qui peut éviter des accès disques, notamment pour les résultats intermédiaires. Une mauvaise gestion de la mémoire peut entraîner un effondrement des performances dû au swap système. Cependant, il est difficile d'incorporer l'optimisation de l'utilisation de la mémoire dans l'optimiseur de requêtes. Dans cet article, nous proposons des solutions pour une gestion efficace de la mémoire lors de l'exécution de requêtes complexes. Plusieurs méthodes statiques d'allocation de la mémoire sont d'abord proposées, puis un mécanisme dynamique, plus efficace est présenté. Ce dernier change l'ordre d'exécution des requêtes de manière à optimiser la gestion de la mémoire. Il permet ainsi de s'adapter à d'éventuels débordement de capacité ou à des erreurs du modèle de coût. Nous réalisons une évaluation de performances sur un prototype montrant des gains importants par rapport aux stratégies statiques.*

MOTS CLÉS: *gestion de la mémoire, exécution de requête, système, swap.*

1. Introduction

Large queries, *i.e.* queries with many complex operators (like join and aggregate) on large databases, have become very frequent in modern decision-support applications. Because they are often of a strategic nature, good response times are required. With either single-processor or parallel systems, this can only be achieved using large amounts of main memory, so as to avoid disk accesses for intermediate results. But main memory is not free and infinite. The “buy more memory” solution to increase performance has obvious limitations, especially with increasing multi-query workloads. Therefore, efficient memory management is crucial.

Poor memory management can hurt performance [COR 96] and even lead to system thrashing because of *paging*, *i.e.* repeated disk accesses to free memory pages. To avoid paging, commercial database systems typically rely on database tuning. For instance, Oracle [Ora97] requires the DBA to specify the maximum amount of memory for each join (Hash-Area-Size or Sort-Area-Size parameters). Although it works, this solution is difficult for the DBA and can hardly result in optimal memory utilization.

Memory management optimization could be automatically supported by the query optimizer (at compile-time) and/or the query engine (at run-time). When producing the “optimal” query plan, the optimizer may also take into account the amount of memory needed for intermediate results. However, this is a hard optimization problem [GAR 97] which can make traditional optimization strategies, like Dynamic Programming [HAA 89], impractical. Furthermore, such prediction is based on a cost model with important parameters (database statistics, available memory size, etc.). These parameters may no longer be accurate at run-time. For instance, an intermediate result may be much larger than expected. If the query engine executes the query plan “as is”, performance can be bad. To improve the quality of query plans, some optimization decisions may be delayed until run-time [GRA 89]. During optimization, several execution alternatives are “encoded” in the plan by means of *choose-nodes*. At run-time, the alternatives are chosen based on actual parameter values. Such work does not consider optimal memory utilization. Furthermore, it would be very hard to include all alternatives at compile-time given the number of parameters influencing memory consumption.

In this paper, we consider memory management optimization during query execution. Related research has been done in the areas of buffer management, real-time queries and multi-queries [DAV 95, SAC 86, CHO 85]. Buffer management algorithms [HAA 90, O’N 93] concentrate on intelligent caching but we are not aware of any approach that tries to optimize resource utilization by considering the query execution plan.

In the context of real-time queries, [PAN 94, DAV 94] propose to modify the algorithms of the relational operators in order to adapt to the available memory. This solution can be applied for one operator. However, memory allocation cannot be done locally at the operator level based on the available remaining memory because this might block execution. For instance, the operator that is executed first claims all the available memory, thereby blocking all the following operators.

Finally, in a multi-query environment, [YU 93] defines the concept of *return on*

consumption to study the overall reduction in response times due to additional memory allocation. The objective is to find a near-optimal way to distribute the available memory over the queries in order to obtain the best overall reduction in response time. The authors propose a heuristic based on the observation that an operator, say a hybrid-hash join, may obtain better return on consumption if it is executed in its maximum memory allocation point (therefore consuming much memory for a short time) or near its minimum memory allocation point (consuming less memory for more time). The heuristic devised is then to allocate more memory to queries which need less memory. However, the authors restrict themselves to single-join queries.

In this paper, we address the problem of efficient memory management for large query execution. We propose two solutions. The first one applies a static memory allocation scheme at start-up time (just prior to execution) which distributes the memory among the operators of a query plan. We propose several algorithms to do this. The second solution is a dynamic execution model which performs memory-adaptive scheduling of the query. It resolves at run-time the memory allocation problems as they occur, by dynamically changing the scheduling of the query plan. We provide an experimental validation and comparison of these strategies.

The paper is organized as follows. Section 2 states the problem more precisely. Section 3 proposes algorithms for static memory allocation. Section 4 motivates and proposes our dynamic execution model, with related algorithms. Section 5 presents a set of experiments using a prototype implementation which shows significant gains over static strategies. Section 6 concludes.

2. Problem Formulation

In this section, we state our assumptions regarding query processing. This will help defining the problem. Query processing is classically done in two steps. The query optimizer generates an “optimal” *query execution plan* (denoted by *QEP*) for a query. The *QEP* is then executed by the query engine which implements an *execution model* and uses a library of relational operators [GRA 93a].

2.1. Query Execution Plans

The optimizer can consider different shapes of *QEP*: left-deep, right-deep, segmented right-deep or bushy. Bushy plans are the most general, and the most appealing, because they offer the best opportunities to minimize the size of intermediate results [SHE 93]. Thus, we consider bushy trees in this paper.

A *QEP* is represented as an *operator tree* and results from the “macro-expansion” of the join tree [HAS 94]. Nodes represent atomic physical operators and edges represent dataflow. Two kinds of edges are distinguished: blocking and pipelined. A blocking edge indicates that the data is entirely produced before it can be consumed. Thus, an operator with a blocking input must wait for the entire operand to be materialized before it can start. A pipelined edge indicates that data can be consumed “one-tuple-at-a-time”. So the consumer can start as soon as one input tuple has been produced.

Let us consider an operator tree that uses hash join. Three operators are needed: scan to read each base relation, build and probe. The build operator produces a blocking output, *i.e.* the hash table, while probe produces a pipelined output, *i.e.* the result tuples. Thus, there is always a blocking edge between build and probe, *i.e.* a blocking constraint between operators. An operator tree can be decomposed as a set of maximum pipeline chains (*pc* for short), *i.e.* chains with highest numbers of pipelined operators. Each *pc* is identified by a number assigned in a left-deep recursive manner, noted *pid*. Figure 1 shows a bushy QEP with 6 relations and the corresponding operator tree. PC_i denotes a pipeline chain, $Scan_i$, $Build_i$, $Probe_i$ are the operators and HT_i is a hash-table, result of the execution of PC_i . Throughout this paper we use the execution plan of Figure 1 as a running example.

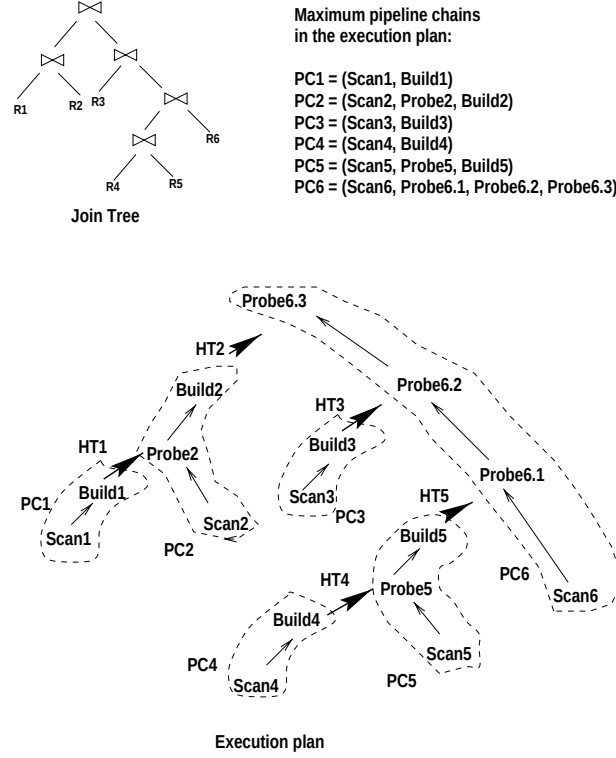


Figure 1. A join tree and an execution plan

2.2. Memory Allocation

Most relational operators, *eg.* join, aggregate, union, can be implemented using hashing (*eg.* hash join), sorting (*eg.* sort-merge join) or nested loops. To ease presentation, we consider multi-join queries using hash-based algorithms. However, our discussion is independent of relational operators and of their implementation.

Relational operators can operate in a range of memory allocations between their minimum and maximum requirements with dissimilar performance results. If the memory allocated to an operator is smaller than its minimum memory requirement (denoted by M_{min}^{op}), the operator cannot be executed [DEW 84]. The operator executes with the optimal performance at its M_{max}^{op} . The performance degrades if the operator is executed in between M_{max}^{op} and M_{min}^{op} . In the latter case we say that the operator is *degraded*. We define the *memory allocation point* of an operator, denoted M^{op} , as the amount of memory which is available to this operator to execute.

For example, a hybrid-hash join [SCH 89] of relations R and S has its minimum memory allocation point at $\sqrt{\delta \mid R \mid}$, where R is the smaller of the two relations and δ is the expansion factor for the hash table (overhead of the hash structure) [YU 93], while the maximum allocation point is $\delta \mid R \mid$, when the hybrid hash can be “up-graded” to a simple hash join.

2.3. System Paging

The query engine typically trusts the compile-time decisions of the optimizer and executes the QEP “as is”. However, the optimizer uses a cost model with many parameters, such as database statistics and available resources (CPU power, disk bandwidth, memory, etc.) to compute the cost of QEP’s. The values of these parameters can change between compile-time and run-time [COL 94, HAA 97, IOA 91] and make the optimizer decisions inappropriate. The incapacity of a query engine to cope with inappropriate plans can severely hurt performance. If the available memory is less than that assumed by the optimizer, the database system can even thrash because of paging.

To illustrate paging, let us consider pc_6 of Figure 1. Assume that $HT_5 + HT_3 + HT_2$ (hash tables produced respectively by pc_5 , pc_3 , and pc_2) do not fit in memory. To execute pc_6 , these hash tables are loaded in virtual memory, *i.e.* part of the tables is in physical memory while some other part is on a swap partition on disk. Each tuple read by Scan6 is probed with HT_5 and can potentially access any page of HT_5 . If the tuple matches, it is probed with HT_3 . Again, it can access any page of HT_3 . The same procedure is repeated for HT_2 . Thus, during the execution of pc_6 , pages of HT_5 , HT_3 and HT_2 are “randomly” accessed, with one page-fault for each page not in physical memory.

To evaluate the negative effects of paging, we have simulated the execution of multi-join queries using a *naive* execution model which simply executes the query plan without considering memory, and varied the available memory. The methodology used for this experiment is described in Section 5.. Experimental results, described in the table below, show a dramatic performance degradation when paging occurs even with a small ratio of missing memory.

<i>% of available memory</i>	<i>Degradation (%)</i>
100	0
90	50
80	160
70	350
60	625
50	1200

A simple solution to avoid paging is to fix a maximum memory size for a *QEP* and never bypass this limit. However, memory allocation cannot be done *locally* at the operator level based on the available remaining memory because this might block execution. Therefore, this distribution must be done *globally* by considering the whole *QEP*.

2.4. Problem Statement

We can now simply state the problem. Given a bushy *QEP* and its budget of available memory, our goal is: (i) to produce an execution which does not generate paging, (ii) to distribute the available memory among the operators of a *QEP* in a near-optimal way in order to minimize response time.

3. Static Memory Allocation

In this section, we propose several strategies and algorithms to statically distribute the available memory among operators. Each algorithm may be applied at start-up time when the system is aware of the amount of available memory. We present these algorithms in the context of the commonly accepted *iterator model* [GRA 93a, GRA 96, ANT 96].

3.1. The Iterator Model

The iterator model considers each operator as an *iterator* that supports three procedure calls: **open** to prepare an operator for producing an item, **next** to produce an item, and **close** to perform final clean-up. A *QEP* is activated starting at the operator tree root and progressing towards the leaves. The dataflow in the model is demand-driven: a child operator passes a tuple to its parent node in response to a **next** call from the parent. Thus the iterator model allows for pipeline operator execution and the order of execution of a *pc* is fixed by the shape of the tree.

For simplicity, let us consider a hash join operator. The **open** call on the join performs the following steps: (i) allocate the hash table, (ii) open the left input, (iii) issue **next** to the left input while there are tuples, and insert them in the hash table, (iv) issue **close** to the left input. The **next** call on the join issues **next** calls to the right input until a match is found with the previously built hash table. Finally, the **close** call on the join issues a **close** to the right input and releases the hash table. Thus, the operators are executed in a left-deep recursive way, *pc's* are executed one-at-a-time. For example, the iterator model executes the tree of Figure 1 in the order $pc_1, pc_2, pc_3, pc_4, pc_5, pc_6$.

3.2. Strategies for Static Memory Allocation

To develop an efficient memory distribution strategy, we use the heuristic of *ignoring the memory needed for the result of a pc*¹. This heuristic does not yield paging. In the worst case, the result of a *pc* can be written on disk and read back once. In addition, more memory can be granted to the operators and thus increase the number of operators executing at their M_{max}^{op} . Since pipeline chains are executed sequentially, at any given time, a *pc* being executed can obtain the whole memory allocated to the *QEP*. Thus the problem of distributing memory between the operators of a *QEP* can be reduced to distributing memory between the operators of a *pc*. We now present several strategies to do this.

Even distribution (Even) A naive strategy is *even* distribution between operators of a *pc*. It can hardly be optimal but is independent of the size of the relations and thus is robust wrt estimation errors. The algorithm is simple: (i) give its M_{min}^{op} to each operator to ensure its execution, (ii) distribute the remaining memory evenly between the operators.

Maximum memory for smaller demand (Max2Min) The second strategy follows the heuristic devised in [YU 93]. The idea is to allocate the maximum amount of memory to the operators which have the smallest M_{max}^{op} in order to obtain a better return on consumption. The algorithm is as follows: (i) give M_{min}^{op} to each operator, (ii) sort operators in reverse order wrt M_{max}^{op} , (iii) distribute the remaining memory following the sort order, trying to complete memory allocation of first operators up to M_{max}^{op} .

For example, let us consider *pc*₆ of Figure 1. Suppose that *HT*₅, *HT*₃ and *HT*₂ are respectively of size 9, 25, and 16 MB and the available memory is 32 MB. Thus, first Probe6.1, Probe 6.2 and Probe6.3 get M_{min}^{op} , i.e. respectively 3, 5 and 4 MB (ignoring δ for simplicity). The remaining memory is 20 MB. Then, we try to complete memory allocation of the smallest operator, i.e. Probe6.1, and add 6 MB to it. The second smallest operator, i.e. Probe6.3 gets an additional 12 MB of memory and Probe 6.2 the remaining 2 MB. Thus, Probe6.1 and Probe6.3 are executed at their M_{max}^{op} and Probe6.2 is executed near its M_{min}^{op} .

Other strategies Following the same scheme, there are several other strategies. We have tested the **Ratio** strategy which gives to each operator an amount of memory proportional to its M_{max}^{op} and **Max2Max** strategy which give the maximum amount of memory to the operators having the largest M_{max}^{op} .

3.3. Discussion

The advantages of static allocation are simplicity and the fact that they can be applied without changing the execution model. In fact, these algorithms only assume sequential execution of *pcs* [SHE 93]. However, static allocation has problems: (i) it is based

¹We have experimentally validated this heuristic.

on estimated values² which may be inaccurate or costly to evaluate at start-up time (needs to access the meta-base); (ii) since there is no control of memory consumption during execution, physical operators must be able to dynamically adapt to their operand sizes, *eg.* augment the number of buckets in a hybrid-hash join; and (iii) the memory allocation may not give the best performance because it tries to execute at any cost maximum pipeline chains.

Let us explain this last point. Consider again the example of Figure 1. The above strategies would try to execute pc_6 in pipeline by distributing the available memory between the three probes. An alternative is to “cut” pc_6 , *i.e.* change a pipelined edge into a blocking edge. This pushes to the extreme the heuristic proposed in [YU 93] in order to give all the available memory to a given operator to try to execute it at its maximum allocation point. This could be done either statically, at start-up time, or dynamically when finding that the memory is full. To avoid estimation errors, we prefer a dynamic approach.

4. Dynamic Execution Model

The *QEP* produced by the optimizer does not specify fully its execution. Some decisions on how to execute a plan are left to the query engine. Thus, without changing the *QEP* and respecting data dependencies, the query engine can: (i) decide on the order of the execution of *pcs*, (ii) decide whether operators in *pcs* are always executed in a pipeline fashion, or whether the result of the operators is materialized, (iii) choose the memory allocation point of each operator, and (iv) decide on the memory management policy.

Our execution model uses these “degrees of freedom” to adapt dynamically to the available resources. The input of our execution model is a *QEP*, a scheduling S for this *QEP*, and an amount of available memory M . By scheduling, we mean a total ordering of the execution of *pcs* in the *QEP*. Our execution model takes as input any scheduling. However, some schedulings yield better resource utilization than others. We propose the **SchedOpt** algorithm, detailed in Appendix A, which generates a scheduling optimizing memory utilization.

In this section, we present the design rules of our model, give some useful terms and definitions, describe the algorithms and present a complete example. We conclude by discussing the pros and cons of our model.

4.1. Design Rules

To avoid paging and to optimize resource utilization, our dynamic execution model follows several rules:

Rule 1: Start the execution of a *pc* only if all the data needed for its execution holds in M .

Rule 2: If a *pc* cannot be fully executed in M , cut the *pc*.

Rule 2 applies when auxiliary structures, such as hash tables, do not fit in M . The *pc* is

²except the Even strategy which is far from optimal.

split into smaller pipeline chains which are executed one after the other. Assuming that each operator can be executed in M at its M_{max}^{op} , this decision guarantees that there is no paging. The temporary result can be written on disk and read again to execute. For example, let us assume that $HT_2 + HT_3 + HT_5 > M$ and that $HT_3 + HT_5 < M$. Two possible schedulings appear: (i) execute pc_6 until Probe6.1, materialize the result and continue execution of pc_6 (Probe6.2 and Probe6.3); (ii) execute pc_6 until Probe6.2, materialize its result and then execute the rest of pc_6 (Probe6.3). Obviously, the relative benefit of each alternative depends on the size of the intermediate result.

Rule 3: If an operator can not be executed in M , degrade it.

A degraded operator is executed at $M_{op} < M_{max}^{op}$. As such an execution generally incurs high overhead, our model tries to avoid the application of the Rule 3 by applying Rule 2.

Rule 4: When M is full, transfer to disk the data that is needed the latest for execution.

Unlike traditional buffer management algorithms such as LRU or MRU, our execution model exploits the knowledge regarding execution, *i.e.* which base relations or temporary results are currently in memory, and when each relation in memory is to be used. Thus it can reduce significantly the number of disk accesses.

Rule 5: Never flush data to disk if it is used by another pc .

Violation of this rule can have the same effects as paging, *i.e.* repeated disk accesses.

Rule 6: Construct the auxiliary structures only when needed.

The execution model avoids useless work by constructing structures, *eg.* hash tables just before they are actually used. Rules 5 and 6 guarantee that such work is performed only once during execution.

4.2. Preliminaries

Several basic functions to manipulate the pipeline chains of a QEP will be useful to present the execution model algorithms. Let $pc = (op_1, \dots, op_i, \dots, op_n)$ be a pipeline chain in QEP where op_i is an operator, S be a scheduling of pipeline chains in a QEP and M be the amount of memory available to a QEP , we define the following terms:

blocks: Let pc_1 and pc_2 be pipeline chains in QEP , op_1 be an operator of pc_1 , op_2 be an operator of pc_2 . If op_1 blocks op_2 , then pc_1 blocks pc_2 . In our example, pc_3 blocks pc_6 .

successor: If pc_1 blocks pc_2 , pc_2 is successor of pc_1 , noted $Succ(pc_1)$. In our example, $pc_6 = Succ(pc_3)$.

blocking set: The *Blocking Set* of a pipeline chain pc , noted $BSet(pc)$ is a set of pipeline chains that block pc . In our example $BSet(pc_6) = (pc_5, pc_3, pc_2)$.

executable: A pc is executable if all $pc' \in \text{BSet}(pc)$ have been already executed.

Now, we define some functions with the format **FunctionName(Input):Output**:

Cut($pc, op_i \in pc, S$): $pc', \overline{pc'}, S$. The pipeline chain pc is cut in $pc'=(op_1, \dots, op_i)$ and $\overline{pc'}=(op_{i+1}, \dots, op_n)$, and replaced in S by $\{pc', \overline{pc'}\}$. The BSet of pc' and $\overline{pc'}$ are recomputed. In our example, **Cut**($pc_6, \text{Probe6.2}, S$), where $S=(pc_1, pc_2, pc_4, pc_5, pc_3, pc_6)$ returns $pc'=(\text{Scan6}, \text{Probe6.1}, \text{Probe6.2})$, $\overline{pc'}=(\text{Probe6.3})$ and $S'=(pc_1, pc_2, pc_4, pc_5, pc_3, pc_6.1, pc_6.2)$.

FindBestOp(pc, M): $op_i \in pc$. This function finds op_i such that $pc' = (op_1, \dots, op_i)$ is the pipeline chain that can be executed in M and materializing the result of op_i incurs minimal I/O overhead among $(op_1, \dots, op_1) \dots (op_1, \dots, op_n)$.

Next(pc, S): pc' . pc' is such that it follows pc in S and has not yet been executed. For example, **Next**(4)=5.

4.3. Model

We model the activity of the query engine as an event-driven system using the notions of *pc_tasks*, *scheduler* and *executor*. *Pc_tasks* represent pipeline chains being executed. They are created by the scheduler and passed on to the executor with specific commands. The scheduler knows the scheduling S , the execution plan QEP , the state of the memory, and the current state of all *pc_tasks*. The scheduler monitors the execution of a *pc_task* and sends commands to the executor. The executor executes one *pc_task* at a time. It communicates with the *memory manager* to obtain new pages, send messages to inform the scheduler when a particular situation during execution occurs, and waits for the scheduler decision. The *MemoryOverflow* message indicates that the memory manager failed to allocate a new page and thus, memory is full. The *EndOfTask* message reports that a given *pc_task* has been successfully completed. The execution ends when all *pc_tasks* are completed and the final result is produced³.

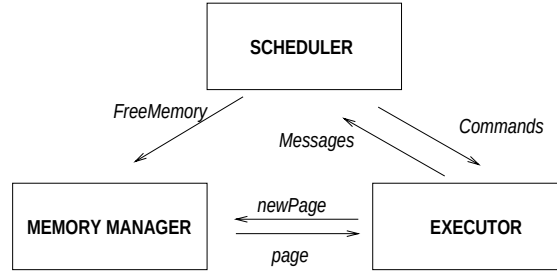


Figure 2. Query Engine Architecture

³Messages are implemented as function calls and thus very cheap.

4.3.1. *Pc_tasks*

A *pc_task* p is defined as a tuple $\langle pc, state, mem, disk \rangle$. pc is the corresponding pipeline chain, $state$ is one of the states given in Figure 3, mem is the current amount of memory consumed by p , and $disk$ is the current amount of disk space consumed by p . The commands of the scheduler change the state of a *pc_task*. During execution, the values of the mem and $disk$ variables are also changed. For ease of presentation, we apply the definitions of Section 4.2. to *pc_tasks*. For instance $BSet(p)$ means $BSet(pc)$, where pc is a pipeline chain of p . In the following section, we use interchangeably the terms pc and *pc_task*.

The state diagram is given in Figure 3. The states are detailed below.

- p is in **Active** state if it is created by the scheduler and given to the executor.
- p is in **Idle** state if its execution is completed. Its result can be stored in memory, on disk, or partially in memory and partially on disk.
- p is in **Used** state if p is completed, p is in $BSet$ of p' and p' starts its execution. Thus the result of p is necessary for the execution of p' and should be loaded in memory. Furthermore, an auxiliary structure based on the result of p is constructed, if needed, and stays in memory until the end of p' (Rule 5 and 6). For example, the beginning of the execution of the *pc_task* containing pc_2 puts the *pc_task* containing pc_1 in **Used** state, *i.e.* loads the result of pc_1 and constructs the hash table needed by pc_2 .
- p is in **Done** state if $Succ(p)$ is in **Idle** state. Thus the results of p are not needed anymore and the executor can free the memory allocated to p .

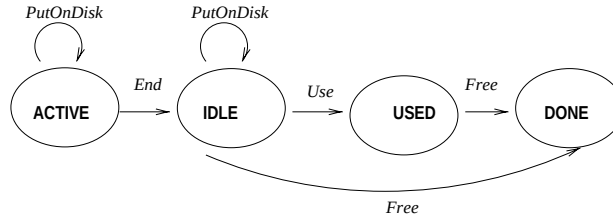


Figure 3. *Pc_task* execution state diagram

4.3.2. *The Scheduler Algorithm*

The algorithm works as follows: while there are pipeline chains to execute in S ,

- ask for the next pipeline chain pc ,
- create a new *pc_task* p by calling $Create_task(pc)$,
- read the data needed for execution of pc in memory and construct auxiliary structures if needed (Rule 6),
- ask the executor to execute p ,

(v) process the messages from the executor by calling EndOfTask() or MemoryOverflow().

The algorithms for the Create_task(), MemoryOverflow() and EndOfTask() functions are given in Figures 4, 5 and 6 respectively. We give below more explanations.

```

CreateTask(pc)
/* pc is the currently scheduled pipeline chain */
if(pc can be completely executed in memory M)
    p=< pc, active, 0, 0> /* Rule 1 */
else
    if (opi = FindBestOp (pc, M) is not null) /* Rule 2 */
        (pc',  $\overline{pc'}$ , S) = Cut(pc, opi, S);
        p=< pc', active, 0, 0>
    else /* by Rule 3 operator opi is to be degraded */
        Mopi = M; /* by default Mopi = Mopmax */
        /* Note, that opi is the first operator of pc */
        (pc',  $\overline{pc'}$ , S) = Cut(pc, opi, S);
        p=< pc', active, 0, 0>
return p;

```

Figure 4. CreateTask algorithm

CreateTask: In our calculation of the amount of memory needed for a pc , we do not consider the size of the result that can be stored on disk. One of the following is possible:

1. pc **can be completely executed in** M . Following Rule 1, the scheduler creates a pc_task $p=< pc, active, 0, 0>$.
2. pc **cannot be completely executed in** M . Following Rule 2, the scheduler "cuts" pc into pc' and $\overline{pc'}$ and creates a pc_task $p=< pc', active, 0, 0>$. As stated previously, the scheduler finds the "best" points to "cut" the pipeline chain. To compute this (these) point(s) the scheduler has to generate all possible "cuttings" of the pipeline chain that fit in the memory.⁴ Since at this moment all the left operands of the pipeline chain have been materialized, the scheduler knows exactly the sizes of these operands which facilitates the task of generating different possible schedulings. Then, the scheduler evaluates the relative overhead of each scheduling, which is directly proportional to the size of intermediate results (that will be materialized as a result of execution of this scheduling). To estimate the intermediate result size, the scheduler has to estimate the size of

⁴Note, that the number of possibilities remains small even for long pipeline chains. For example, for a pc with 6 pipelined joins only 30 "cutting" possibilities are generated.

the pipelined relation. Note, that the accuracy of these estimates is not crucial because of the following: (i) the scheduler is only interested in *relative* costs in order to choose the best alternative; (ii) even if the scheduler doesn't "cut" the pipeline chain in the best point, the chosen scheduling will never provoke paging (since only possibilities that fit into memory are generated).

3. **even op_1 of pc cannot be executed in M .** The scheduler failed to create a smallest possible pc_task , i.e. a pc_task containing the *first* operator of pc , thus this operator can not be executed in M . The only possible solution is to degrade op_1 , and proceed by creating $pc_task\ p \mid op_1 \in pc, pc \in p$.

```

MemoryOverflow()
/* current is a current pc_task, pc ∈ current */
/* check if some data can be transferred to the disk */
if ( ∃p | p is in Idle state,
    pid(p) < pid(current), mem(p) > 0)
    putOnDisk(p);
    continue(current);
else
    /* continue to execute current */
    /* writes its result to disk */
    PutOnDisk (current)

```

Figure 5. *MemoryOverflow algorithm*

MemoryOverflow: The scheduler can take one of the following decisions:

1. **Transfer some data to disk.** This is possible if there is some data D in memory which is to be used after the data D' being currently produced (Rule 4). In effect, if D is to be used before D' , writing D' directly to disk produces less disk accesses. Furthermore, the scheduler should not select data produced by a pc in a Used state (Rule 5).
2. **Continue execution by writing on disk.** If none of the above is possible, the scheduler continues execution by writing the results directly on disk.

EndOfTask: In response to this message, the scheduler frees the memory given to the operands of the current task.

4.4. Complete Example

We now illustrate our model with a complete example of the execution of the QEP of Figure 1. The initial scheduling S given by the algorithm SchedOpt is $(pc_1, pc_2, pc_4, pc_5, pc_3, pc_6)$. The scheduler finds the first executable pipeline chain, pc_1 and creates

```

EndOfTask(current)
/* current is a current pc_task, pc ∈ current */
state of current = Idle;
End(current);
Let PI denote the pc_tasks in Used state.
For ∀pi ∈ PI
    state = Done;
    Free(pi);

```

Figure 6. *EndOfTask* algorithm

pc_task $p1 = \langle pc_1, active, 0, 0 \rangle$, issues the command *Execute*($p1$) and waits for the executor's messages. The executor processes $p1$ and stores the result of the execution in memory. When $p1$ is done, it sends an *EndOfTask*($p1$) message to the scheduler. The scheduler puts $p1$ in *Idle* state, looks for the next pipeline chain to execute, finds pc_2 , creates a new task $p2$. As *BSet*($p2$) contains pc_1 , the scheduler must construct HT_1 . It does this by issuing *Use*($p1$) to the executor. When the table is constructed, the scheduler issues *Execute*($p2$). The execution of $p2$ produces a *MemoryOverflow* message from the executor that cannot obtain new pages from the memory manager. The scheduler continues the execution of $p2$ by storing the result directly on disk. When $p2$ is done, the executor sends the *EndOfTask*($p2$) message to the scheduler. The execution of $p4$, $p5$ and $p3$ follow the same scheme. Now, pc_6 has to be executed. The scheduler calls **FindBestOp**(pc_6, M). **FindBestOp**() finds out that $HT_5 + HT_2 + HT_3$ does not fit in memory, but $HT_3 + HT_2 < M$ and $HT_3 + HT_5 < M$. Thus, two "cutting" points are possible: after Probe6.1 or after Probe6.2. **FindBestOp**() evaluates the costs of these schedulings and chooses the cheapest one, namely, cutting the pipeline chain after Probe6.1. Thus, the scheduler cuts pc_6 by calling **Cut**($pc_6, \text{FindBestOp}(pc_6, M)$). The **Cut** function returns two pipeline chains, $pc_{6.1} = (\text{Scan6}, \text{Probe6.1})$ and $pc_{6.2} = (\text{Probe6.2}, \text{Probe6.3})$ and replaces pc_6 by $\{pc_{6.1}, pc_{6.2}\}$ in scheduling S . Thus, $S = (pc_1, pc_2, pc_4, pc_5, pc_3, pc_{6.1}, pc_{6.2})$. The scheduler creates a new task $p6.1$, puts a task $p5$ in *Used* state and issues *Execute*($p6.1$) to the executor. At the end of $p6.1$ the scheduler executes $p6.2$ and since **Next**($pc_{6.2}$) is null, it stops.

The step-by-step execution is detailed below. The *Message* column shows the orders of the scheduler (in this font) and the messages from the execution engine (in *this* font). For shortness, the function **CreateTask**() is referred to as **Create**(). The column *Pc_task* shows the state of the current task. Variable *m* denotes the amount of memory used by the current task, variable *d* denotes the amount of disk space used by the current task. If a pc_task is not currently using any memory and/or disk space, the corresponding variable is shown equal to 0. The column *Mem* shows which relations (or part of which relations) are currently in memory, and column *Disk* shows which relations (or part of which relations) are saved on disk.

<i>Message</i>	<i>Pc_task</i>	<i>Mem</i>	<i>Disk</i>
Create(p_{c1})	$p_1:<pc_1,Active,0,0>$		
Execute(p_1)	$p_1:<pc_1,Active,m,0>$	1	
EndOfTask(p_1)			
End(p_1)	$p_1:<pc_1,Idle,m,0>$	1	
Create(p_{c2})	$p_2:<pc_2,Active,0,0>$	1	
Use(p_1)	$p_1:<pc_1,Used,m,0>$	1	
Execute(p_2)	$p_2:<pc_2,Active,m,0>$	1,2	
MemoryOverflow			
PutOnDisk(p_2)	$p_2:<pc_2,Active,m,d>$	1,2	2
EndOfTask(p_2)			
Free(p_1)	$p_1:<pc_1,Done,0,0>$	2	2
End(p_2)	$p_2:<pc_2,Idle,m,d>$	2	2
Create(p_{c4})	$p_4:<pc_4,Active,0,0>$	2	2
Execute(p_4)	$p_4:<pc_4,Active,m,0>$	2,4	2
EndOfTask(p_4)			
End(p_4)	$p_4:<pc_4,Idle,m,0>$	2,4	2
Create(p_{c5})	$p_5:<pc_5,Active,0,0>$	2,4	2
Use(p_4)	$p_4:<pc_4,Used,m,0>$	2,4	2
Execute(p_5)	$p_5:<pc_5,Active,m,0>$	2,4,5	2
EndOfTask(p_5)			
Free(p_4)	$p_4:<pc_4,Done,0,0>$	2,5	2
End(p_5)	$p_5:<pc_5,Idle,m,0>$	2,5	2
Create(p_{c3})	$p_3:<pc_3,Active,0,0>$	2,5	2
Execute(p_3)	$p_3:<pc_3,Active,m,0>$	2,5,3	2
MemoryOverflow			
PutOnDisk(p_2)	$p_2:<pc_2,Idle,0,d>$	5,3	2
EndOfTask(p_3)			
End(p_3)	$p_3:<pc_3,Idle,m,0>$	5,3	2
Create(p_{c6})			
Cut(p_{c6})			
Create($p_{6.1}$)	$p_{6.1}:<pc_{6.1},Active,0,0>$	5,3	2
Use(p_5)	$p_5:<pc_5,Used,m,0>$	5,3	2
Execute($p_{6.1}$)	$p_{6.1}:<pc_{6.1},Active,m,0>$	5,3	2
MemoryOverflow			
PutOnDisk($p_{6.1}$)	$p_{6.1}:<pc_{6.1},Active,m,d>$	5,3	2,6.1
EndOfTask($p_{6.1}$)			
Free(p_5)	$p_5:<pc_5,Done,0,0>$	3	2,6.1
End($p_{6.1}$)	$p_{6.1}:<pc_{6.1},Idle,m,d>$	3	2,6.1
Use(p_3)	$p_3:<pc_3,Used,m,0>$	3	2,6.1
Use(p_2)	$p_2:<pc_2,Used,m,0>$	3,2	6.1
Execute($p_{6.2}$)	$p_{6.2}:<pc_{6.2},Active,m,0>$	3,2,6.2	
EndOfTask($p_{6.2}$)			
Free(p_3)	$p_3:<pc_3,Done,0,0>$		
Free(p_2)	$p_2:<pc_2,Done,0,0>$		

4.5. Discussion

Our model has several important features. First, it adapts dynamically during the execution and thus does not depend on the accuracy of estimates. Second, it takes as input any scheduling and this scheduling can be modified during execution. This feature opens the possibility of optimizing the scheduling, for example using our *SchedOpt* algorithm. It can also be applied in distributed or parallel environments where dynamically changing the scheduling can be very useful to address problems such as data availability [AMS 96] and load balancing [BOU 96]. Third, the overhead of our execution model, *i.e.* the cost of the scheduler algorithms, is negligible as executor/scheduler communications can be implemented as function calls. In addition, the scheduler is involved during execution only at very specific times, *eg.* *MemoryOverflow*, *EndOfTask*. Fourth, it is based on the heuristic of “cutting” a pipeline chain and materializing intermediate results in order to provide more memory to the operators and thus increase the number of operators executing at their M_{max}^{op} . In the worst case, this heuristic implies writing on the disk and reading back the intermediate result. Static strategies may execute more operators at a point different from M_{max}^{op} . The execution of an operator at a point different from M_{max}^{op} generally induces a number of disk accesses equal to two to three times the size of their operands [YU 93]. Since in a pipeline chain the operand of an operator is the result of a previous operator, static strategies may also materialize intermediate results. The validity of this heuristic was verified in our experimentations (see Section 5.).

5. Performance Evaluation

The performance evaluation of several execution strategies is made difficult by the need to experiment with many different queries and large relations. The typical solution is to use simulation which eases the generation of queries and data, and allows testing with various configurations. On the other hand, using implementation and benchmarking would restrict the number of queries and make data generation very hard.

We used a performance evaluation methodology similar to [BOU 96]. We fully implemented our dynamic execution model and the iterator model with the different static algorithms, and simulated the execution of operators. With this approach, query execution does not depend on relation content and can be simply studied by generating queries and setting relation parameters (cardinality, selectivity).

In the rest of this section, we describe our prototype and report on performance results varying the available memory and the estimation accuracy. We present results with a single *QEP* in order to explain the behavior of the different strategies and also averaged results to evaluate the overall gain of our dynamic strategy. The first experiment reports on performance comparison with perfect estimates while the second experiment shows the performance degradation of the static strategies when errors are introduced in estimates.

5.1. Experimentation Platform

We first detail the way we have generated the different *QEP*'s, then describe our prototype and the parameters used for the operator simulation. Finally, we present the methodology that was applied in all experiments.

5.1.1. Generating Query Execution Plans

The input to our execution model is a *QEP* obtained after compilation and optimization of a user query. To generate queries, we use the algorithm given in [SHE 93] with three kinds of relations: small (10K-20K tuples), medium (100K-200K tuples) and large (1M-2M tuples). First, the predicate connection graph of the query is randomly generated. As in [SHE 93], we consider only acyclic connected graphs. Second, for each relation involved in the query, a cardinality is randomly chosen in one of the small, medium or large ranges. Third, the join selectivity factor of each edge (R,S) in the predicate connection graph is randomly chosen in the range $[0.5 * \min(|R|, |S|) / |R \times S|, 1.5 * \max(|R|, |S|) / |R \times S|]$.

To generate *QEP*'s, we have implemented a Volcano-style dynamic programming query optimizer [GRA 93b]. For each query, the best *QEP* is retained, *i.e.* the one which has the smallest estimated cost. As we master the query generation, optimization and execution, we can easily compute the *QEP* cost with no errors or introduce some errors on purpose to simulate real situations.

Without any restriction, the produced *QEP*'s have a high variation in memory consumption. After several tries, we used the following rule to generate *QEP*'s with roughly the same memory needs: consider only *QEP*'s with their *minimum optimal memory* between 100 MB and 128 MB. We define the minimum optimal memory as the amount of memory necessary to execute the memory greediest pipeline chain of the *QEP* (without considering its result), each operator executing at its M_{max}^{op} .

In order to verify the stability of the experimental results, we have produced thousands of *QEP*'s, varying the number of relations (from 6 to 12), the minimum optimal memory and the distribution of the relations over the small, medium and large ranges. We then restricted ourselves to 50 *QEP*'s with 8 relations for the complete experiments.

5.1.2. Experimental Prototype

We have implemented the iterator model along with the static memory distribution algorithms and our dynamic execution model. For the sake of comparison, we also implemented a naive execution strategy which executes the *QEP* "as is", relying on a classical buffer management strategy (LRU). The performance of this strategy is shown in Section 2.. The different execution strategies share the simulated operator library, simulated buffer management system and I/O system. Since the different strategies use the same lower-level code, the performance difference can only stem from the execution strategies. We used classical parameters [YU 93] for the simulation. They are presented below. The prototype is written in C and runs on a Sun Ultra1.

<i>Parameter</i>	<i>Value</i>
CPU Speed	100 Mips
Disk Latency	17 ms
Disk Seek time	5 ms
Disk Transfer Rate	6 MB/s
I/O Cache Size	8 pages
Perform an I/O	3000 Instr.
Tuple Size	200 bytes
Page Size	8 Kb
Move a tuple	500 Inst.
Search for match in hash table	500 Inst.
Produce a result tuple	100 Inst.
Available memory	16 MB - 256 MB

5.1.3. Experimentation Methodology

In the following experiments, each point in a graph is obtained from a computation based on the response times of 50 *QEP*'s. Since the different *QEP*'s correspond to different queries, we have two problems. (i) Each *QEP* has different memory requirements (even if their minimum optimal memory is in the same range, *eg.* 100 MB-128 MB). Therefore, we cannot use the available memory as value for the X-axis. The solution is to use a *memory ratio* (noted *MemRat*) defined as the ratio of available memory wrt the minimum optimal allocation memory. (ii) Computing the average response time does not make sense. Therefore, the results with several *QEP*'s will always be in terms of comparable execution times. To compare the performance results of strategy A over strategy B, the *performance ratio* is defined as the ratio of the response time with strategy B over the response time with strategy A. Strategy A is then called the *reference*. Each point is computed as the average of this ratio for all *QEP*'s. For instance, an average ratio of 1.2 means that strategy B is, on average, 20% slower than strategy A. Therefore, each point of a graph is obtained with 50 measurements, each on a different *QEP*, using the formula:

$$\frac{1}{50} \sum_{i=1}^{50} \text{resp. time of experiment} / \text{reference resp. time}$$

where the reference response time will be indicated for each experiment. Each response time is computed as the average of five successive measurements. Using this methodology, we obtained several MB of raw results which we treated with a database in order to compute averaged ratios and analyze the results.

5.2. Performance Comparisons with Perfect Estimates

For this first experiment, the 50 *QEP*'s were generated with perfect estimates (both during query optimization and query execution). We first show the results with one *QEP* and then present the averaged results.

Figure 7 presents the results for one of the 50 *QEP*'s⁵, so we can use the response time as Y-axis. It shows the performance of strategies **Even**, **Ratio**, **Max2Min** and of our dynamic execution model called **Dyn**. To better understand the results, we present in Figure 8, the number of operators which are executing under their M_{max}^{op} point, i.e. the number of degraded operators for the same *QEP*.

Even is the worst strategy when $MemRate > 100\%$, i.e. when there is sufficient memory to execute all operators at their M_{max}^{op} point. As **Even** evenly distributes the memory between the operators, it leads to the degradation of several operators. Each jump observed in the **Even** curve is a new degraded operator.

This is avoided by the **Ratio** strategy as it distributes memory between operators proportionally to their M_{max}^{op} . When $MemRate$ is 90%, the **Ratio** strategy assigns a little less than M_{max}^{op} to each operator of the memory greediest pipeline chain of the *QEP*. Thus, its response time jumps from 512s to 851s. The jump is higher than for the **Even** strategy as all the operators of one pipeline chain degrade at the same time. This behavior is repeated successively for each pipeline chain of the *QEP*.

Max2Min is clearly the best static strategy as it only degrades one operator when there is no way to execute one pipeline chain entirely without degrading any operator. It begins by degrading the memory greediest operator of the memory greediest pipeline chain. This is repeated successively, degrading one operator at each jump.

Max2Max⁶ is one of the worst strategies. **Max2Max** gives the maximum memory to the greediest operator in a pipeline chain, thus leading to more degraded operators.

Dyn appears to be the best strategy as it degrades an operator only when its M_{max}^{op} is larger than the memory available for the whole plan. For this *QEP*, it occurs when $MemRate = 60\%$ (see Figure 8). When $MemRate$ is between 100% and 60 %, the response time increases smoothly as **Dyn** materializes some of the intermediate results while “cutting” the pipeline chains. This exhibits, for this specific *QEP*, the validity of the heuristic used in **Dyn**. Note that **Dyn** also obtains better results when $MemRate > 100\%$. This stems from a better initial scheduling, computed by the algorithm given in Appendix A.

Figure 9 presents averaged results of **Max2Min**, the best static strategy, and **Dyn**. We compute the performance ratio using **Dyn** as reference. We also show in this figure the minimum and maximum performance ratio over the 50 *QEP*'s. Figure 10 presents the averaged number of degraded operators for the 50 *QEP*'s. We have also included the **Ratio** and **Even** curves to verify the behavior of the different strategies.

Figure 10 is very similar to Figure 8. Obviously, each curve is smoother as it results from averaging 50 values. The first jump happens, when $MemRate = 100\%$, because we take as X-axis, the ratio of the available memory and the minimum optimal memory, i.e. all the *QEP*'s degrade at the same point. Note also that **Dyn** has an average of degraded operators less than 1 until $MemRate = 50\%$, i.e. on average, at most one operator is degraded. This happens with $MemRate = 100\%$ for **Max2Min** and **Ratio**.

Thus, Figure 9 gives an idea of the gain expected by our approach, over the best static strategy, **Max2Min**. On average, **Dyn** is always better than **Max2Min** by a

⁵We choose it randomly since all individual experiments (i.e. for one *QEP*) present the same picture.

⁶For clarity, we do not show the **Max2Max** strategy on the graphs.

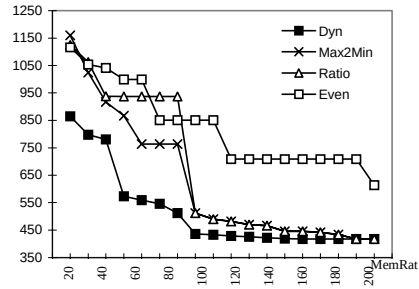


Figure 7 : Response time with one QEP

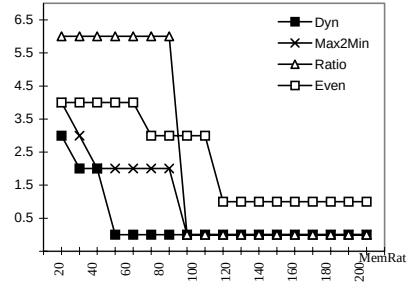


Figure 8 : Number of degraded operators with one QEP (no errors)

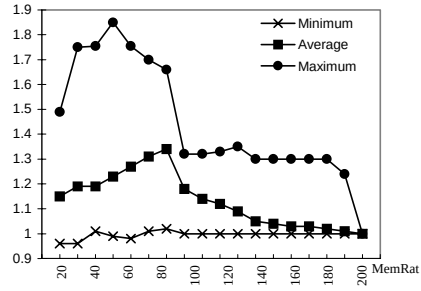


Figure 9 : Performance ratio with 50 QEP's (no errors)

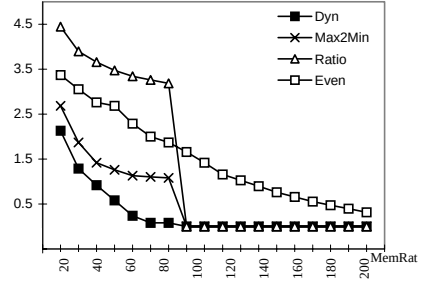


Figure 10 : Number of degraded operators with 50 QEP's (no errors)

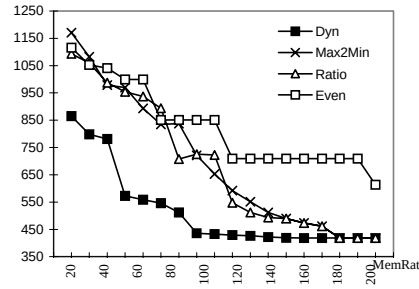


Figure 11 : Response time with one QEP error rate = 20 %

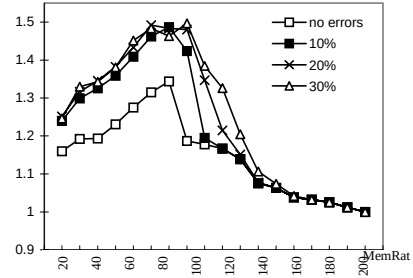


Figure 12 : Performance ratio with 50 QEP's and errors

factor up to 35% when $Memrat = 90\%$. After this peak, the gains smoothly decrease until 10%. The maximum and minimum curves show the best (respectively the worst) results of Dyn over the 50 QEP 's. The maximum gain of Dyn over Max2Min is 85% and is obtained with a $MemRat$ of 50%.

5.3. Performance with Estimation Errors

In this second series of experiments, 50 QEP 's were generated with perfect estimates. However, we introduced errors before the query execution⁷. To obtain a measurement with an error rate r , the cardinalities of each intermediate relation were distorted by a value chosen randomly in $[-r, +r]$. The measurements were performed with a realistic error rate between 0 and 30%.

Figure 11 presents the results for the same QEP as in the previous experiment with a 20% error rate. As expected, Even is not affected at all by wrong estimates. Dyn degrades by less than 5%, notably when $MemRat > 100\%$. This stems from the algorithm which statically computes the initial scheduling for Dyn (given in Appendix A). This algorithm still uses estimates to compute the "best" initial scheduling. However, this degradation is insignificant. Ratio and Max2Min degrade because they use static memory distribution with wrong estimates. For the same reason, Max2Min and Ratio degrade as well when $MemRat$ is between 100% and 140%, i.e. there is sufficient memory to execute all the operators at their M_{max}^{op} .

Figure 12 presents averaged results of Max2Min, the best static strategy wrt Dyn. The performance ratio indicates the performance gain that we can expect with a given error rate. The results show that even with a small error rate, eg. 10%, the maximum averaged gain of Dyn is 50%.

6. Conclusion

In this paper, we have addressed the problem of efficient memory management for large query execution. We have proposed two solutions. The first one statically distributes the available memory among the query operators, at start-up time. We have proposed several algorithms to do this. The winner is the one which degrades the smallest number of operators, by giving the maximum memory to those which have the smallest demand. The advantages of static algorithms are their simplicity and the fact that the execution model need not be changed. However, static allocation suffers from cost estimate errors. Furthermore, it may not give the best overall performance because it tries to execute at any cost a maximum number of operators in a pipeline fashion.

Therefore, we proposed a more efficient dynamic execution model which performs memory-adaptive scheduling of the query. It resolves at run-time the memory allocation problems as they occur, by dynamically changing the scheduling of the query plan. Our model can "cut" a pipeline chain at the point where it incurs minimal overhead if there is not enough memory for concurrent execution of all its operators. In the

⁷We also tried to introduce errors during optimization. However, this makes difficult the analysis of the results since the generated QEP 's are not comparable anymore.

same context, static strategies would resort to degrading operators. Thus, our dynamic execution model is more suited to give the best possible performance.

To validate and compare static and dynamic strategies, we have performed experiments using a prototype implementation. The experiments with many queries show significant gain over static strategies, even when considering perfect estimates, *i.e.* when the memory consumption of the query is predicted with no errors. This case is the most favorable to static strategies. However, the experiments show an important performance gain of the dynamic strategy (between 10% and 35 % on average, with a maximum gain of 85%). This result is explained by the much higher number of degraded operators incurred by static strategies. With estimates errors, this performance difference increases (from 35% with no errors until 50% with an error rate of 10%).

These experiments demonstrate the significant performance gains that can be achieved by our dynamic execution model. Considering the trend towards multiuser workloads of large queries, we believe such execution model is crucial for the best utilization of memory. Future work will extend this model to work in a distributed environment.

References

- [AMS 96] AMSALEG L., FRANKLIN M. J., TOMASIC A., URHAN T., "Scrambling Query Plans to Cope With Unexpected Delays", *Conf. on Parallel and Distrib. Inf. Systems (PDIS)*, 1996.
- [ANT 96] ANTONSHENKOV G., ZIAUDDIN M., "Query Processing and Optimization in Oracle Rdb", *VLDB Journal*, vol. 5, nr 4, 1996.
- [BOU 96] BOUGANIM L., FLORESCU D., VALDURIEZ P., "Dynamic Load Balancing in Hierarchical Parallel Database Systems", *Int. Conf on VLDB*, 1996.
- [CHO 85] CHOU H.-T., DEWITT D. J., "An Evaluation of Buffer Management Strategies for Relational Database Systems", *Int. Conf. on VLDB*, 1985.
- [COL 94] COLE R. L., GRAEFE G., "Optimization of Dynamic Query Evaluation Plans", *ACM-SIGMOD Int. Conf.*, 1994.
- [COR 96] CORRIGAN P., GURRY M., Chapter: What causes performance problems, in *Oracle Performance Tuning*, 1996.
- [DAV 94] DAVISON D. L., GRAEFE G., "Memory-Contention Responsive Hash Joins", *Int. Conf on VLDB*, 1994.
- [DAV 95] DAVISON D. L., GRAEFE G., "Dynamic Resource Brokering for Multi-User Query Execution", *ACM-SIGMOD Int. Conf.*, 1995.
- [DEW 84] DEWITT D., KATZ R., OLKEN F., SHAPIRO L. D., STONEBRAKER M., WOOD D. A., "Implementation Techniques for Main Memory Database Systems", *ACM-SIGMOD Int. Conf.*, 1984.
- [GAR 97] GAROFALAKIS M., IOANNIDIS Y., "Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources", *Int. Conf. on VLDB*, 1997.
- [GRA 89] GRAEFE G., WARD K., "Dynamic Query Evaluation Plans", *ACM-SIGMOD Int. Conf.*, 1989.
- [GRA 93a] GRAEFE G., "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys*, vol. 25, nr 2, p. 73–170, June 1993.
- [GRA 93b] GRAEFE G., MCKENNA W. J., "The Volcano Optimizer Generator: Extensibility and Efficient Search", *Int. Conf. on Data Engineering*, 1993.
- [GRA 96] GRAEFE G., "The Microsoft Relational Engine", *Int. Conf. on Data Engineering*, 1996.

- [HAA 89] HAAS L., FREYTAG J., LOHMAN G., PIRAHESH H., “Extensible Query Processing in Starburst”, *ACM-SIGMOD Int. Conf.*, 1989.
- [HAA 90] HAAS L., CHANG W., LOHMAN G. M., MCPHERSON J., WILMS P. F., LAPIS G., LINDSAY B. G., PIRAHESH H., CAREY M. J., SHEKITA E. J., “Starburst Mid-Flight: As the Dust Clears.”, *Trans. on Knowledge and Data Eng.*, vol. 2, 1990.
- [HAA 97] HAAS L. M., CAREY M. J., LIVNY M., SKUKLA A., “Seeking the Truth About ad hoc Join Costs”, *VLDB Journal*, vol. 6, nr 3, 1997.
- [HAS 94] HASSAN W., MOTWANI R., “Optimization Algorithms for Exploiting the Parallel Communication Tradeoff in Pipelined Parallelism”, *Int. Conf on VLDB*, 1994.
- [IOA 91] IOANNIDIS Y. E., CHRISTODOULAKIS S., “On the Propagation of Errors in the Size of Join Results”, *ACM-SIGMOD Int. Conf.*, 1991.
- [O’N 93] O’NEIL E. J., O’NEIL P. E., WEIKUM G., “The LRU-K Page Replacement Algorithm For Database Disk Buffering”, *ACM-SIGMOD Int. Conf.*, 1993.
- [Ora97] “Oracle8 Server Administrator’s Guide. Oracle8 Server Reference. Oracle8 Server Concept”, 1997.
- [PAN 94] PANG H., CAREY M. J., LIVNY M., “Managing Memory for Real-Time Queries”, *ACM-SIGMOD Int. Conf.*, 1994.
- [SAC 86] SACCO G. M., SCHKOLNICK M., “Buffer Management in Relational Database Systems”, *TODS*, vol. 11, nr 4, 1986.
- [SCH 89] SCHNEIDER D. A., DEWITT D. J., “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment”, *ACM-SIGMOD Int. Conf.*, 1989.
- [SHE 93] SHEKITA E. J., YOUNG H. C., TAN K.-L., “Multi-Join Optimization for Symmetric Multiprocessor”, *Int. Conf. on VLDB*, 1993.
- [YU 93] YU P. S., CORNELL D. W., “Buffer Management Based on Return on Consumption in a Multi-Query Environment.”, *VLDB Journal*, vol. 2, nr 1, 1993.

A The Algorithm SchedOpt

In this appendix, we motivate the necessity of an intelligent algorithm of the initial scheduling and describe our approach. The example is based on the *QEP* of Figure 1.

Our goal is to order the execution of the *pc*'s of the *QEP* optimizing the use of memory. Consider two possible initial schedulings: $S_1 = (pc_1, pc_2, pc_4, pc_5, pc_3, pc_6)$ and $S_2 = (pc_4, pc_5, pc_1, pc_2, pc_3, pc_6)$. For both schedulings, the memory utilization during the execution is given below:

$S_1 = (pc_1, pc_2, pc_4, pc_5, pc_3, pc_6)$	$S_2 = (pc_4, pc_5, pc_1, pc_2, pc_3, pc_6)$
1) HT_1	HT_4
2) $HT_1 + HT_2$	$HT_4 + HT_5$
3) HT_2	HT_5
4) $HT_2 + HT_4$	$HT_5 + HT_1$
5) $HT_2 + HT_4 + HT_5$	$HT_5 + HT_1 + HT_2$
6) $HT_2 + HT_5$	$HT_5 + HT_2$
7) $HT_2 + HT_5 + HT_3$	$HT_5 + HT_2 + HT_3$
8) result	result

The maximal amount of memory taken by the scheduling S_1 is $\max(HT_1 + HT_2, HT_2 + HT_5 + HT_4, HT_2 + HT_5 + HT_3)$ and maximal amount of memory taken by the scheduling S_2 is $\max(HT_5 + HT_4, HT_5 + HT_1 + HT_2, HT_5 + HT_2 + HT_3)$. The final decision where scheduling consumes less memory depends on the cardinalities of base relations and intermediate results. Thus, to find better scheduling, one has to calculate memory consumption of both schedulings and compare them.

The SchedOpt algorithm produces all possible schedulings, *i.e.* considers all possible combinations of the order of execution of $pc's$, and choose the one that uses a minimal amount of memory. It accomplishes its task by determining recursively the order of execution of operands for each pc . Thus, for our example, it orders pc_2, pc_3 and pc_5 wrt pc_6 .

Thus, the algorithm is exponential wrt the number of operands for each pc , *i.e.* the number of operators in a pc . In practice, bushy execution plans tend to have a fairly small number of operators in a pc . In addition, this complexity can be easily reduced by remarking that the pc with empty BSet are always scheduled last. Thus, these $pc's$ can be explicitly scheduled to be executed last and are not considered among all possible combinations of the order of execution of $pc's$. The possible gain of the algorithm is the cost of I/O of blocking inputs, *eg.* hash tables.