

Exploiting Pipeline Interruptions for Efficient Memory Allocation

Josep Aguilar-Saborit⁺ Mohammad Jalali⁺ Dave Sharpe⁺ Victor Muntés-Mulero^{*}
⁺IBM Toronto Lab
8200 Warden Ave. Markham
ON L6G1C7, Canada
{jaguilar, mjalali, dsharpe}@ca.ibm.com
^{*}DAMA-UPC
Universitat Politècnica de Catalunya, Campus Nord
Edif. C6, C. Jordi Girona 1-3, 08034 Barcelona, Spain
vmuntes@ac.upc.edu

ABSTRACT

Efficiency of memory-intensive operations is a key factor in obtaining good performance during multi-join query processing. The pipelined execution of these queries forces the operations in the query plan to be processed concurrently. Making a wrong decision regarding the amount of memory allocated for such operations can have a drastic impact on the response time. However, some of the execution algorithms used at run time interrupt the pipelined execution, ensuring that some operations are never executed concurrently. Because of this, it is essential to explore new approaches in order to improve memory exploitation.

In this paper, we address the problem of memory allocation for multiple concurrent operations in a query execution plan. First, we study the dynamic needs for memory during the execution time, and define when two operations coexist. Then, we propose a post-optimization phase, which (i) identifies the operations that are concurrently executed at run time and (ii) costs different memory allocation combinations to find a near optimal solution for any type of query execution plan. We have implemented our techniques in the IBM[®] DB2[®] Universal DatabaseTM (DB2 UDB) showing that they achieve significant execution time improvements compared with previous approaches, especially for multi-join queries accessing large data volumes.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems

General Terms

Algorithms, Performance

1. INTRODUCTION

The execution of a query in database management systems (DBMSs) is usually divided into different phases. First, a semantic analysis of the query is carried out by the parser and the rewriter; second, a cost-based analysis of the rewritten

query is performed by the optimizer that generates a query execution plan (QEP); third and final, the resulting plan is executed by the runtime query processor. Among several challenges related to this process, performance during the execution of multi-join queries is one of major importance and it is tightly dependent on the memory utilization. Specifically, depending on the distribution of memory resources among the different join operations, the execution time of a query can change drastically. This problem arises in very common scenarios for multi-join query processing, such as those involving star join query processing [5].

These type of queries pose a serious problem for query optimization. If we just consider the join ordering problem, the search space of possible QEPs to solve a specific query grows exponentially to the number of joins in the query. The complexity of this problem is increased even further when we consider the use of different scan and join implementations, or the exploration of bushy trees instead of left-deep trees. In these scenarios, adding any extra degree of complexity, like considering the different memory distributions for the join operation in the QEP, may lead the optimizer to be unable to traverse the whole search space [9]. Commercial DBMSs such as DB2 UDB or Oracle fall into static approaches where (i) the DBA sets the start-up parameters that the compiler gathers to compute the query execution plan, and (ii) the memory assigned to all the memory-intensive operations in the QEP is usually distributed equally. The main drawback of this approach is twofold: first, it relies on the expertise of the DBA; second, it is sub-optimal because it does not take into account that different operators in a QEP can have completely different memory needs.

As an alternative, research has been carried out in order to statically assign memory via rule-based approaches during compile time, or dynamically readjust the memory available for each operation during the execution of the query [4, 11, 18]. Static strategies using heuristics during compile time are very limited and return results that are usually far from the optimal solution. On the other hand, dynamic adjustments can be expensive depending on the point of the query execution when they have to be applied or, as happens in the study presented in [18], they do not take effect after several executions of the same query. The work presented in [15] tries to solve this problem from a cost-basis perspective during the compilation time of a query, organizing the pipelined operations into the so-called *shelves*. However, using the definition and properties of shelves, the authors do not consider bushy trees, which are commonly used in commercial systems. In addition, they base their proposals on a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM'08, October 26–30, 2008, Napa Valley, California, USA.

Copyright 2008 ACM 978-1-59593-991-3/08/10 ...\$5.00.

theoretical cost function that differs significantly from those in real systems in terms of complexity, and makes them difficult to be applied in practice.

In this paper, we tackle the problem of memory allocation at compile time among the operations in a QEP. Although most of the DBMSs follow a pipelined philosophy in order to execute the queries, a large number of common QEP operations interrupt the data flow by requiring to read at least one of the input data sets completely before starting to return results. The main idea of our work is to exploit the available memory of the system by reusing the memory released by these operations. Specifically, we focus on the hybrid hash join (HHJ) operation that, among all join algorithms, is characterized by being memory-intensive and very dependent on the available memory to achieve good performance. Also, the HHJ is one of the most common join operations used by commercial DBMSs and, as an example of the operations mentioned before, it partially interrupts the pipeline execution because of the need for building an in-memory hash table on one of its input data flows.

We study the coexistence of multiple operations competing for memory during the QEP processing. We propose a method to identify clusters of operations that compete for memory during the execution of a query and that, opposite to shelves presented in [15], allow for bushy trees. We also propose the *Memory Chunk Distribution* (MCD) algorithm. MCD is a method to distribute the available memory, after logically dividing it into chunks, among the different operations in a cluster in order to achieve a better utilization of the available memory, independently of the complexity of the cost function. Finally, we implement our algorithm in DB2 UDB, and show that our memory allocation techniques improve the query execution performance compared with the current DB2 algorithm and other algorithms presented in the literature, both in terms of I/O and time.

This paper is organized as follows. We explain the basic concepts and assumption in Section 2. In Section 3, we formulate the memory allocation problem, and describe our proposal to solve it. Then, we present the experimental results in Section 4. Finally, we explain the related work in Section 5 and draw our conclusions in Section 6.

2. BASIC CONCEPTS

Opening the optimizer search to the exploration of bushy trees has proven to be better than restricting the search to just left-deep trees [13]. After these results, most commercial database systems allow for these types of trees when generating the best QEP. Usually, the execution of the operations in these QEPs is pipelined in order to reduce the amount of memory used during the process. Still, some of the operations in the QEP are highly memory-intensive since they partially interrupt the pipeline and accumulate data in memory in order to speed up the overall execution. A good example of these are HHJ operations. Similar to previous studies on this subject [15], in this paper we concentrate exclusively on the hybrid hash implementation of joins. Note, however, that this does not imply a loss of generality of the ideas presented. Note that, although we assume a static amount of memory for each query at compilation time, this does not prevent us from combining our technique with existing dynamic memory reallocation algorithms used at run-time.

The HHJ [8] is the evolution of the grace hash join algo-

rithm [10, 14], and has become the usual implementation of the hash join operation in commercial DBMSs such as the IBM DB2 UDB, Microsoft[®] SQL-Server and the Oracle database products.

Let us call h , a hash join operation implemented using the HHJ algorithm. For the following discussions in this section, we denote by R and S two relations representing the incoming data flow to be joined during the pipelined execution of a QEP, where $|R|$ and $|S|$ represent the size in pages of each relation. We denote by M the memory pool available to the algorithm in pages. The execution of operation h is divided into three phases:

Build phase. In this first phase, by applying a hash function, relation R is partitioned into a set of $B + 1$ disjoint buckets $R_0, R_1, \dots, R_B$. Assuming that the data is uniform, this first phase must ensure that each bucket fits within the memory assigned for the operation, thus, B depends on M . The tuples in bucket R_0 are used to build an in-memory hash table and the B remaining buckets are stored into disk. For our work, we are assuming that the HHJ operation does not allocate memory for the hash table until the arrival of the first tuple from the build side. Note also that, once all the tuples coming from the build side have been read and the hash table and all the different buckets are created, all the lower operations in the QEP belonging to the build side have already projected all the qualifying tuples and, therefore, they release memory allocated for their execution. We denote by h^b a hash join operation h during the build phase.

Probe phase. In this second phase, relation S is partitioned using the same hash function as in the build phase. A total of $B + 1$ buckets are created. While tuples belonging to S_0 are immediately probed into the hash table of bucket R_0 for possible matches, the remaining B partitions are stored into disk. Note also that, once all the tuples coming from the probe side have been read, all the lower operations in the QEP belonging to the probe side have already projected all the qualifying tuples. Therefore, they finish their execution, releasing the previously allocated memory. We denote by h^p a hash join operation h during the probe phase.

Joining phase. Finally, in this phase pairs of buckets $\langle R_i, S_i \rangle$, for $1 \leq i \leq B$, need to be joined in the following two steps: (i) build an in-memory hash table with the tuples from R_i and (ii) read the tuples from the counterpart bucket S_i looking for matches in the hash table. We denote by h^j a hash join operation h during the joining phase.

As it can be deduced from the different phases, choosing the right value for the memory pool M is essential for ensuring high performance in an HHJ operation. An accurate modeling of the HHJ algorithm is presented in [8, 17]. These previous works prove the importance of the memory available to the hash join algorithm. In general, the larger the memory available for HHJ, the larger the portion of relation R that fits in memory and, therefore, the faster the HHJ operation runs. However, commercial DBMSs cost functions for hash joins are very sophisticated and take into account many other aspects such as the buffer pool utilization, the length of the chains in the hash table, or the use of bloom filters [3]. As we see later in this paper, it is unrealistic to assume that an HHJ cost function is a linear function of the available memory and, therefore, any attempt to use a linear programming problem solver such as *simplex* is unsuitable.

Finally, note that considering a pipelined execution does not restrict our proposal to serial environments; intra-operator

parallelism implemented in symmetric multiprocessor (SMP) environments is perfectly compatible with the pipelined execution of a QEP. The extension of our work to inter-operator parallelism is beyond the scope of this paper.

3. MEMORY ALLOCATION PROBLEM

Most well-known commercial query optimizers open the search space to bushy trees in order to increase the probability of finding a QEP with a cost closer to the optimum. In the case of bushy trees, not all the operations coexist in memory because, as we explained in the previous section, the pipeline is usually interrupted and the lower operations finish their execution releasing their allocated memory. In this section, we define the memory allocation problem for hash join operations.

3.1 Preliminary Definitions

We assume that a QEP contains scan and join operations. We consider the join operations to be competing for the memory available in a hash join-dedicated memory pool. Therefore, we only consider the distribution of the memory among the different hash join operations. In order to clarify the study, we do not represent scan operations in our formalism and we consider binary trees that only contain hash join operations, although scan operations are present in the actual QEP. Nevertheless, our study could be extended to QEPs containing any kind of operations. Following these assumptions, we define a QEP as a set of hash join operations. The formal definition follows.

DEFINITION 3.1. We define a QEP $Q = (V, E)$ as a labeled directed strongly binary tree, where V is the non-empty set of vertices in Q , and E represents the non-empty set of edges in Q where:

- V is a set of nodes or hash join operations $h_{x_1}, \dots, h_{x_m}$ in Q such that $\forall i : 1 \leq i \leq m, (\forall j : 1 \leq j \leq m \wedge j \neq i, x_i \neq x_j) \wedge (x_i = 1 \vee (x_i > 1 \wedge \exists k : 1 \leq k \leq m \wedge k \neq i, \lfloor \frac{x_i}{2} \rfloor = x_k))$
- E is the set of edges in Q and $\forall i : 1 \leq i \leq m, \forall j : 1 \leq j \leq m \wedge \lfloor \frac{x_i}{2} \rfloor = x_j, \exists e : e \in E, e = (h_{x_i}, h_{x_j})$ where $e = (x, y)$ represents a directed edge from hash join x to hash join y .

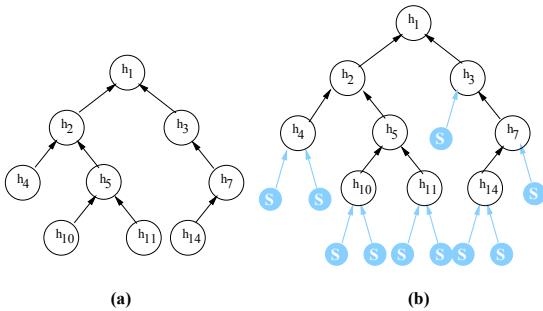


Figure 1: Binary tree for a query execution plan.

Given a QEP $Q = (V, E)$, we call h_1 the root hash join operation in V . Given a hash join operation $h_x \in V$, we

name h_{2x} and h_{2x+1} the hash join operations that provide the probe and the build tuples to h_x , respectively. Therefore, $h_{\lfloor \frac{x}{2} \rfloor}$ is the parent hash join operation of h_x .

Note that, with this definition, we are assuming that the build side of the hash join comes from the right side of the hash join operation, while the probe side of the hash join comes from the left side. This assumption is held all throughout this paper.

Figure 1.a shows an example of a query execution plan that fits Definition 3.1. Figure 1.b shows the extended binary tree corresponding to the actual representation of the first QEP, where special nodes (representing the scan operations) are added wherever a null subtree was present in the original tree, so that each node in the original tree (except the root node) has degree of three.

DEFINITION 3.2. Given a QEP $Q = (V, E)$, we say that a hash join operation $h_y \in V$ is a descendant of $h_x \in V$ if $\exists i : i \geq 1, \lfloor \frac{y}{2^i} \rfloor = x$. Let us denote by D_x the set of descendants of h_x .

For instance, taking as an example the binary tree represented in Figure 1.a, $D_2 = \{h_4, h_5, h_{10}, h_{11}\}$.

3.2 Memory Coexistence

Now, we discuss all the situations where two hash join operations coexist in memory, assuming the operational model of the HHJ presented in Section 2. Afterwards, we formally define when two join operations coexist in memory.

Assuming a QEP fitting Definition 3.1, there are three different situations in which a hash join operation h_x coexists in terms of memory with another hash join operation h_y in the QEP:

Condition 1: h_x is the parent operation of h_y . This condition establishes that a join operation h_y does always coexist with its parent operation h_x . There are two possible cases, both depicted in Figure 2.a. In the first case in the leftmost QEP, operation h_y is located in the *build* side of h_x . In this case, as h_y provides tuples during its *probe* phase, after probing them against its hash table, ht_y , h_x inserts these qualifying tuples into its hash table ht_x . Therefore, the two tables have to coexist in memory. In the second case (rightmost QEP in Figure 2.a), operation h_y is located in the *probe* side of h_x . In this case, both *build* and *probe* phases of h_y need to create and check the hash table ht_y , respectively. Therefore, the two hash tables ht_x and ht_y have to coexist in memory at the same time. Note that this figure only depicts the case where h_y is executing the probe phase.

Condition 2: h_y^p is included in the set of operations executing their probe phase to provide tuples during the build phase of h_x , h_x^b . The second condition describes the scenario where a set of operations are feeding, through a pipelined execution, the hash table of an upper operation during its build phase. Figure 2.b depicts this scenario. When the child operation h_{2x+1} located in the build side of h_x starts providing tuples to h_x , in turn, the child hash join operation located in the probe side of h_{2x+1} , h_{4x+2} , is also providing tuples. In this situation, all these hash join operations coexist in memory with h_x .

Condition 3: h_y is located in the set of descendants of the first descendant in the probe side of h_x . This case is graphically depicted in Figure 2.c. The hash table

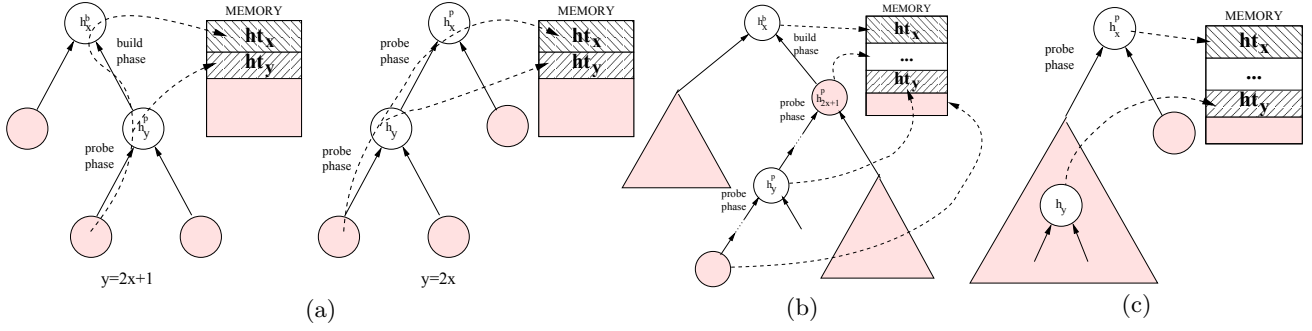


Figure 2: Summary of cases where a hash join operation h_x coexists with other hash joins in the QEP.

ht_x has already been created during h_x^b and the probe phase of h_x , h_x^p , is being executed. This requires the execution of all the operations in the probe side. Therefore, any memory allocated for one of these operations coexists with ht_x .

We formally summarize these three situations as follows.

DEFINITION 3.3. Two hash join operations h_x and h_y coexist in memory, $h_x \overset{M}{\Leftarrow} h_y$, when one of the following three conditions holds: 1) one operation is the parent operation of the other one: $x = \lfloor \frac{y}{2} \rfloor$ or $y = \lfloor \frac{x}{2} \rfloor$; 2) one of the operations is reached recursively traversing the join operations located in the probe side, starting from the join operation in the build side of the other operations: $\exists i : i \geq 1, (\frac{y}{2^i} = 2x + 1) \vee (\frac{x}{2^i} = 2y + 1)$; or 3) one operation is a descendant of the join operation located in the probe side of the other operation: $h_y \in D_{2x}$ or $h_x \in D_{2y}$.

Note that the operation $h_x \overset{M}{\Leftarrow} h_y$ is symmetric but it is not transitive. A simple counterexample for transitivity follows. Given a hash operation h_x , we know by the first condition that it coexists with both children operations, $h_x \overset{M}{\Leftarrow} h_{2x}$ and $h_x \overset{M}{\Leftarrow} h_{2x+1}$. However, by the operational model of a hash join operation explained in Section 2, we know that when h_x^b is over and before starting h_x^p , the memory allocated for h_{2x+1} is released. Only when h_x^p starts, tuples from the probe side are retrieved and, therefore, memory for operation h_{2x} is allocated. Therefore, $h_{2x} \overset{M}{\Leftarrow} h_{2x+1}$.

DEFINITION 3.4. Given a QEP $Q = (V, E)$, where $V = \{h_{x_1}, \dots, h_{x_m}\}$, we define a cluster $C \subseteq V$ as a set of hash join operations $C = \{h_{y_1}, \dots, h_{y_n}\}$ that holds $\forall i : 1 \leq i \leq n, \forall j : 1 \leq j \leq n \wedge i \neq j, h_{y_i} \overset{M}{\Leftarrow} h_{y_j}$ and $\forall i : 1 \leq i \leq m \wedge h_{x_i} \in V \setminus C, \exists j : 1 \leq j \leq n, h_{x_i} \overset{M}{\Leftarrow} h_{y_j}$.

That is, a cluster is a complete set of all the operations that coexist in memory at the same time. As a practical example, the clusters in the QEP in Figure 1 are $C_1 = \{h_3, h_7, h_{14}\}$, $C_2 = \{h_1, h_3\}$, $C_3 = \{h_1, h_5, h_{11}\}$, $C_4 = \{h_1, h_2, h_5, h_{10}\}$ and $C_5 = \{h_1, h_2, h_4\}$. Note that a single operation can appear in several clusters as it happens with h_1 , opposite to the definition of *shelves* presented in [15]. The basic difference is that while *shelves* only allow for left-deep trees, clusters allow for any possible tree shape.

3.3 Clustering Algorithm

Based on Definition 3.4, we propose an algorithm to obtain the clusters in a QEP. However, before presenting the algorithm, we formalize some necessary definitions.

DEFINITION 3.5. Given a QEP $Q = (V, E)$, we define a path of hash join operations $P \subseteq V$, as a set of hash join operations $P = \{h_{y_1}, \dots, h_{y_n}\}$ such that $\forall i : 1 < i \leq n, \exists e : e \in E, e = (h_{y_{i-1}}, h_{y_i})$.

Note that, because of the direction of the edges in our QEP definition, paths are always represented in bottom-up direction. Being P a path, we call P_{first} the first element in P , h_{y_1} and P_{last} the last element in P , h_{y_n} .

LEMMA 3.1. Given a QEP $Q = (V, E)$ and two hash join operations $h_x, h_y \in V$ such that $h_y \in D_x$, there exists a path P such that $P_{first} = h_y$ and $P_{last} = h_x$.

PROOF. >From Definition 3.1, given $Q = (V, E)$ such that $V = \{h_{x_1}, \dots, h_{x_m}\}$, we know that $\forall i : 1 < i \leq m, \exists j : 1 \leq j \leq m \wedge i \neq j, \lfloor \frac{x_i}{2} \rfloor = x_j$. Therefore, starting from $h_y \in V$, it is possible to generate a set of hash joins $P = \{h_y, h_{\lfloor \frac{y}{2} \rfloor}, h_{\lfloor \frac{y}{4} \rfloor}, \dots, h_x\}$, since we know that $\exists i : i \geq 1, \lfloor \frac{y}{2^i} \rfloor = x$, from Definition 3.2. Given two consecutive elements in P , $h_{x_{a-1}}$ and h_{x_a} , it is true that $\lfloor \frac{x_{a-1}}{2} \rfloor = x_a$ and therefore, from Definition 3.1, $\exists e : e \in E, e = (h_{x_{a-1}}, h_{x_a})$. Then, P is a path and $P_{first} = h_y$ and $P_{last} = h_x$. $\square$

LEMMA 3.2. Given a QEP $Q = (V, E)$ and a cluster $C \subseteq V$, there always exists a path $P \subseteq V$ such that $C \subseteq P$.

PROOF. Using the subindexes to order the elements in $C \subseteq V$, we obtain a set of hash join operations $C_{ord} = \{h_{y_1}, \dots, h_{y_n}\}$ where $y_{i-1} > y_i$ for $1 < i \leq n$. Let us take any pair $(h_{y_{a-1}}, h_{y_a})$ from the n elements of C_{ord} . From Definition 3.4, we know that any pair of joins $h_{y_i}, h_{y_j} \in C_{ord}$ holds that $h_{y_i} \overset{M}{\Leftarrow} h_{y_j}$, meaning that they hold one of the three conditions explained in Definition 3.3. Therefore, if $x_a = \lfloor \frac{x_{a-1}}{2} \rfloor$ then it is true that $\exists i : \lfloor \frac{x_{a-1}}{2^i} \rfloor = x_a$ where $i = 1$ and, by Definition 3.1, $\exists e : e \in E, e = (h_{y_{a-1}}, h_{y_a})$. Therefore, there exists a path $P_a \subseteq V$ such that $P_a = \{h_{y_{a-1}}, h_{y_a}\}$. If the second condition holds, $\exists i : i \geq 1, \frac{a-1}{2^i} = 2a + 1$, then it is also true that $\exists i : i \geq 1, \lfloor \frac{a-1}{2^i} \rfloor = a$. This means that $h_{y_{a-1}} \in D_{y_a}$, by Definition 3.2. From Lemma 3.1, there exists a path $P_a \subseteq V$ such that $P_{a_{first}} = h_{y_{a-1}}$ and $P_{a_{last}} = h_{y_a}$. Note that, $\exists i : h_i \in P_a, h_i \in V \setminus C_{ord}$. Finally, from the same lemma, if the third condition holds, $h_{a-1} \in D_{2a}$, we directly know that there exists a path $P_a \subseteq V$ such that $P_{a_{first}} = h_{y_{a-1}}$ and $P_{a_{last}} = h_{y_a}$. Therefore, since we have proven that there exists a path $P_a \subseteq V$ joining any pair $(h_{y_{a-1}}, h_{y_a})$ in C_{ord} , it is also true that there exists a path $P \subseteq V$ such that $P_{first} = h_{y_1}$, $P_{last} = h_{y_n}$ and $\forall i : 1 \leq i \leq n \wedge h_{x_i} \in C, h_{x_i} \in P$, and so, $C \subseteq P$. $\square$

DEFINITION 3.6. Given a QEP Q , we define PS_Q , as the set of paths $P_1, \dots, P_p \in Q$ such that $\forall i : 1 \leq i \leq p : P_{i_first} = h_x \wedge P_{i_last} = h_1 \wedge D_x = \emptyset$.

In other words, PS_Q is the set of all the paths going from the leaf nodes to the root. Given a QEP Q , the algorithm proposed in this subsection must traverse all the paths in the PS_Q in order to find the clusters contained in them.

Algorithm 1 Clustering algorithm

```

1:  $\mathcal{P}_{List} := \{P_i\}, P_i \in PS_Q$ 
2:  $\mathcal{C}_{List} := \emptyset$ 
3:  $\mathcal{C}_{Final} := \emptyset$ 

4: for all  $P_i \in \mathcal{P}_{List}$  do                                 $\triangleright$  Build Clusters
5:   for all  $h_j \in P_i$  do
6:      $C_{|\mathcal{C}_{Final}|+|\mathcal{C}_{List}|+1} := \{h_j\}$ 
7:      $\mathcal{C}_{List} := \mathcal{C}_{List} \cup \{C_{|\mathcal{C}_{Final}|+|\mathcal{C}_{List}|+1}\}$ 
8:     for all  $C_k \in \mathcal{C}_{List}$  do
9:       if  $\forall l : h_l \in C_k, h_j \stackrel{M}{\approx} h_l$  then
10:         $C_k := C_k \cup \{h_j\}$ 
11:       end if
12:     end for
13:   end for
14:    $\mathcal{C}_{Final} := \mathcal{C}_{Final} \cup \mathcal{C}_{List}$ 
15:    $\mathcal{C}_{List} := \emptyset$                                  $\triangleright \mathcal{C}_{List}$  can be removed here
16: end for

17: for all  $C_i \in \mathcal{C}_{Final}$  do  $\triangleright$  Remove duplicates or subsets
18:   if  $\exists C_j : C_j \in \mathcal{C}_{Final} \wedge i \neq j, C_i \subseteq C_j$  then
19:      $\mathcal{C}_{Final} := \mathcal{C}_{Final} \setminus \{C_i\}$ 
20:   end if
21: end for
return  $\mathcal{C}_{Final}$ 

```

Algorithm 1 describes the algorithm used to find the set of clusters in a QEP Q . The loop starting in line 4 constructs all the clusters, by following all the paths in the QEP from the leaf hash join operations to the root hash join operation. For each path, we find the sets of hash join operations that coexist in memory in such a way that there are not other operations in that path that coexist with all the operations in one of these subsets. Note that, as proved in Lemma 3.2, for any cluster C there exists a path that includes C . Because of this, we can remove $\mathcal{C}_{List}$ after processing each path (line 15), thus reducing the number of comparisons during the clustering formation process. There are two aspects that must be taken into account: (i) a single cluster can belong to two different paths and, therefore, the algorithm generates duplicates of the same cluster; and (ii) for any cluster C included in a path P_1 , if the number of paths is larger than one, this cluster is at least partially included in another path P_2 . This means that, when traversing P_2 a subset $C' \subset C$ will be considered to be a cluster in P_2 . Because of these two situations, a second loop starting in line 17 is in charge of removing any possible duplicate or subset of a cluster.

3.4 Problem definition

Given a QEP Q and its set of clusters $\mathcal{C}_{Final} = \{C_1, \dots, C_k\}$, a total available memory of M pages and a cost function f_i to evaluate the cost of a hash join operation h_i in the QEP, we define the hash join memory allocation problem, as the

problem of finding the memory assignments $m_1, \dots, m_n$ for the n hash join operations in Q such that $\forall i : 1 \leq i \leq k, \sum_{j:h_j \in C_i} h_j \leq M$ and that $\sum_{i=1}^n f_i(m_i)$ is minimum.

Memory assignments for clusters

Once the list of clusters in a QEP $Q = (V, E)$, $\mathcal{C}_{Final}$, is found and returned by Algorithm 1, the available memory must be distributed among the different hash join operations in Q . As mentioned in Section 2, the cost function used to evaluate the hash join cost can be very complex and dependent on the DBMS engine. Therefore, finding the optimal memory distribution by means of some optimization algorithms, such as simplex or convex optimization, in polynomial time, is not suitable for such functions. Based on previous literature [17], and as a simple proof of concept, we formulate the I/O cost incurred by the build and probe phases. As in Section 2, we call R to the data flow coming from the build side and S to the one coming from the probe side. The build phase must ensure that each bucket R_i (a portion of R) fits in the memory pool M shared by all the processes involved in the join. For out of core hash joins and assuming $|M| \gg 0$, the number of buckets B is calculated as follows $\frac{|R| \times F - |M|}{|M| - 1} \approx \frac{|R| \times F - |M|}{|M|}$, where F is a fudge factor to account for the increment in pages due to structures needed to deal with a given bucket R_i and $|X|$ represents the number of pages in X . Thus, the portion of the build phase that fits in memory $|R_0|$ is approximately $\frac{M-B-1}{F} \approx \frac{M-B}{F}$. This way, $I/O_{cost} = 2 \cdot (|R| - |R_0|) + 2 \cdot \left(|S| \cdot \frac{|R| - |R_0|}{|R|}\right)$ which after simple transformations give us:

$$I/O_{cost} = \left(\frac{2}{F} \cdot \left(1 + \frac{|S|}{|R|}\right)\right) \cdot (|R| \cdot F - |M|) \cdot \left(1 + \frac{1}{|M|}\right)$$

Therefore, by using this simple cost function as an example, we can already see that linear programming problem solvers such as *simplex* are unsuitable since the cost is not a linear function of the available memory. We considered using more complex solutions such as those studied in *convex optimization theory* to solve this problem, but still there are several factors that make this solution inappropriate. For example, cost functions in real systems calculate the cost using different cost formulas depending on several factors defined by the context. For instance, a hash join operation might consider using bit filters or not to improve performance. This makes the cost function not continuous. Taking into account that, the more complex the DBMSs the larger the number of these factors and the larger the complexity, the cost function presents a large number of discontinuities that make all these methods unsuitable. Later in the experiments section, we compare our proposal to other proposals based on the use of less complex cost formulas and *simplex*.

Memory Chunk Distribution Algorithm

In this section, we present the *Memory Chunk Distribution* algorithm. MCD is an exhaustive algorithm that is based on the division of the available memory into chunks, and the distribution of such chunks among the different hash join operations in Q . Dividing the memory into chunks that are larger than a memory page reduces the number of different memory distributions to be exhaustively explored, thus reducing the compile time. Note that, as stated before, the

amount of memory assigned to the hash join operations in a single cluster $C = \{h_{x_1}, \dots, h_{x_l}\} \in \mathcal{C}_{Final}$, $m_{x_1}, \dots, m_{x_l}$, holds that $m_{x_1} + \dots + m_{x_l} \leq M$. Therefore, if the number of clusters is larger than one, it can happen that, with $V = \{h_{x_1}, \dots, h_{x_m}\}$, $\sum_{i=1}^m m_{x_i} > M$. In other words, since

we can assign all the memory to each cluster, the addition of the memory assigned to all the clusters might be larger than the real available memory in the system.

Before discussing the algorithm in detail, we will formally prove that given a QEP containing more than one cluster, each cluster will share at least one hash join operation with another cluster in the QEP.

LEMMA 3.3. *Given a QEP $Q = (V, E)$ and its set of clusters $\mathcal{C}_{Final} = \{C_1, \dots, C_n\}$, $n > 1$, $\forall i : C_i \in \mathcal{C}_{Final}$, $\exists j : C_j \in \mathcal{C}_{Final} \wedge i \neq j, C_i \cap C_j \neq \emptyset$.*

PROOF. Let us prove it by contradiction. Let us suppose a cluster $C \in \mathcal{C}_{Final}$ such that $\forall i : C_i \in \mathcal{C}_{Final} \wedge C \neq C_i, C \cap C_i = \emptyset$. If $n > 1$ then $\exists i : 1 \leq i \leq |V|, h_{x_i} \in V \setminus C$. Since Q is a binary tree it is also a connected graph. Therefore, let us consider all those hash join operations in $V \setminus C$ that are directly connected to the hash join operations in C . By Definition 3.3, these nodes and their parents or children in C coexist in memory and therefore are included in another cluster C_2 such that $C \cap C_2 \neq \emptyset$. $\square$

Given a set of clusters $\mathcal{C}_{Final}$ and before generating the optimal memory chunk assignment, we sort them in order of execution to simplify the post-optimization process. More formally, given a QEP Q and being PS_Q the set of all the paths from the leaves to the root node, we consider sorting PS_Q following the execution order of the operations, that is, we first consider the path from the rightmost leaf node to the root. Afterwards, we choose the path from the second rightmost leaf node to the root, and so on. Let us illustrate this ordering following the example in Figure 1. In this case, the ordered set of paths would be $P_1 = \{h_1, h_3, h_7, h_{14}\}$, $P_2 = \{h_1, h_2, h_5, h_{11}\}$, $P_3 = \{h_1, h_2, h_5, h_{10}\}$ and $P_4 = \{h_1, h_2, h_4\}$. Once the paths are ordered this way in a list $PS_{Q_{ord}}$, we sort the sets obtained by the clustering algorithm in a list $\mathcal{C}_{order} = \{C_1, \dots, C_n\}$ such that $\forall i : 1 \leq i \leq n, \forall j : 1 \leq j < i, \forall k : 1 \leq k \leq |PS_{Q_{ord}}| \wedge C_i \subseteq P_k \in PS_{Q_{ord}}, \forall l : 1 \leq l \leq |PS_{Q_{ord}}| \wedge C_j \subseteq P_l \in PS_{Q_{ord}}, l < k \vee (l = k \wedge \max_i < \max_j)$, where $\max_x$ is the largest hash join index in cluster C_x . Once the clusters are sorted, we proceed to test the different memory distributions among the hash join operations in the QEP.

Algorithm 2 shows the cost-based method we use, once the memory is divided into N chunks, to make the optimal memory assignment for every operation in the QEP, taking into account all the clusters generated by Algorithm 1. In order to find the optimum solution in the search space of all the possible memory distributions, our algorithm exhaustively explores all the possibilities, calculating an associated cost for each one and taking the combination with lower cost. As we show later, we use some pruning techniques that allow us to skip a large number of combinations without losing the optimal, achieving significant reductions on the exploration space. We present a recursive version of the algorithm for the sake of simplicity.

The leading idea behind this algorithm is to assign N memory chunks per cluster. Lemma 3.3 proves that each

Algorithm 2 Memory assignment optimization algorithm

```

1: procedure OPTALLOC( $P_{set}, C_{set}, c, A_{opt}, c_{opt}$ )
2:    $C := next(C_{set})$   $\triangleright$  Next cluster in execution order
3:   if  $C = \emptyset$  then
4:     return ( $P_{set}, \min(c, c_{opt})$ )
5:   end if
6:    $C'_{set} \leftarrow C_{set} \setminus \{C\}$ 

7:    $A_{int} \leftarrow \{(h, m), (h, m) \in P_{set} \wedge h \in C\}$ 
    $\triangleright$  set of assignments fixed in previous clusters
8:    $N_C \leftarrow N - \sum_{(h_i, m_i) \in A_{int}} m_i$ 
9:    $n_h = |C| - |A_{int}|$ 
10:   $c_{min} = c_{opt}$ 
11:   $A_{min} = A_{opt}$ 
12:  for all  $m_1, \dots, m_{n_h} \in [0, N_C], \sum_{i=1}^{n_h} m_i = N_C$  do
13:     $A \leftarrow \emptyset$ 
14:    for all  $i \in 1, \dots, n_h$  do
15:       $B \leftarrow \{(h_{x_i} \in C, m_i), \nexists m, (h_{x_i}, m) \in A_{int}\}$ 
    $\triangleright$  assuming order in  $C$ 
16:       $A \leftarrow A \cup B$ 
17:    end for
18:     $c' \leftarrow c + \sum_{(h_i, m_i) \in A} f_{cost}(h_i, m_i)$ 
19:    if  $c' \geq c_{min}$  then
20:      continue  $\triangleright$  Cancel and process next
21:    end if
22:     $P'_{set} \leftarrow P_{set} \cup A$ 
23:     $(A', c'_{min}) \leftarrow optAlloc(P'_{set}, C'_{set}, c', A_{min}, c_{min})$ 
24:    if  $c'_{min} < c_{min}$  then
25:       $c_{min} = c'_{min}$ 
26:       $A_{min} = A'$ 
27:    end if
28:  end for
29:  return  $A_{min}, c_{min}$ 
30: end procedure

```

cluster intersects at least with another cluster, i.e., some operations are included in more than one cluster. In order to calculate the cost, our algorithm must be aware of which operations have been assigned a cost in the processing of a previous cluster. Our recursive function has five input parameters ($P_{set}, C_{set}, c, A_{opt}$ and c_{opt}) and two output values. The final result is a list of pairs (h_i, m_i) representing the number of memory chunks m_i assigned to each operation h_i . P_{set} keeps the list of memory assignments (h_i, m_i) that have already been processed in previous clusters. Therefore, $P_{set} = \emptyset$ in the first call of the recursive function. C_{set} represents the list of clusters to be processed and it is initialized with the set of clusters $\mathcal{C}_{Final}$ generated by Algorithm 1. Parameter c is the cost of the operations processed so far and it is initially set to zero. A_{opt} is the best memory configuration found up to the moment and can be initialized with any set of values. Finally, c_{opt} is the best cost calculated up to that moment. Its initial value is $c_{opt} = \infty$.

Each recursive call to our function implies the processing of a new cluster in execution order (line 2). First, cluster C is removed from the list of clusters C_{set} (line 6). Second, the algorithm identifies those operations in the current cluster that have been assigned a number of chunks in the processing of a previous cluster (line 7). Third, it calculates the number of chunks that are still available for the current cluster, N_C , after subtracting those already assigned. The

loop in line 12 calculates all the possible memory assignments for the operations in the current cluster the available memory of which has not been decided yet. For each iteration, these assignments are stored in A . In line 18, the cost of these operations is added to the total cost. Then, the memory assignments in A are added to the list of processed assignments P_{set} (line 22). At this point, the function calls itself recursively with the new parameters to process the next cluster.

Algorithm improvements. Note that we are propagating the best cost found so far allowing to prune those branches the cost of which is already higher than the optimum. In this situation, we can directly discard that combination without finalizing the calculus, thus reducing the computation time (line 19). Finally, the algorithm returns the list of the optimum memory assignment and the cost associated to it. In order to run the experiments presented later in this paper, we use an even more efficient version of this algorithm that assumes a lower bound for the memory required by each operation. This way, the number of combinations to be tested is reduced drastically. We have not included this improvement in the algorithm for the sake of simplicity. Next, we calculate the number of possible memory distributions.

Numbering Memory Assignments

Single Cluster Scenario. Let us assume the scenario where all the operations coexist with the other operations in the QEP, as in the case of a left-deep tree (or a right-deep tree, following DB2 UDB conventions). In other words, there exists a single cluster $C \in \mathcal{C}_{Final}$. In this situation, the number of different distributions of memory chunks computed by this algorithm would be $d_1 = CR_{|C|,N} = \binom{|C| + N - 1}{N}$, where $CR_{n,m}$ is the combination with repetition of n elements in groups of m elements, $|C|$ is the number of hash join operations in the cluster. Note that this can be calculated using the number of N -combinations with $|C|$ elements, which is the result of choosing one out of the $|C|$ hash join operations for each one of the N memory chunks.

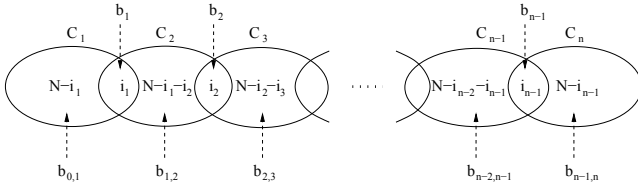


Figure 3: n clusters where hash join operations only belong to at most two. i_j is the memory assigned to an intersection and b_j the number of hash joins.

Multiple-Clusters Scenario. As the complexity of the QEP increases, the probability of having more than a single cluster also increases. Figure 3 depicts a scenario where n clusters are detected and sorted as mentioned above. In this case, each hash join operation belongs at most to two clusters and cluster C_x only shares operations at most with C_{x-1} and C_{x+1} . This would be the typical case in a right-deep tree (or a left-deep tree in our case), for example. We have $n - 1$ subsets representing the intersections of the n clusters.

Given two clusters C_x and C_{x+1} , there are i_x memory chunks to be divided among the b_x hash join operations in the intersection of both clusters. For the remaining operations in cluster C_x , not included in the intersections with C_{x-1} or C_{x+1} , we have $M - i_{x-1} - i_x$ memory chunks to distribute among $b_{x-1,x}$ hash join operations. Given the following definition:

$$K_x^y = \begin{cases} CR_{x,y} & \text{if } x > 0 \\ 1 & \text{if } x = 0 \end{cases}$$

the number of possible memory distributions in this scenario would be calculated as follows:

$$d_n = \sum_{i_1=0}^N \sum_{i_2=0}^{N-i_1} \cdots \sum_{i_{n-1}=0}^{N-i_1-i_2} \left[K_{b_{n-1,n}}^{N-i_{n-1}} \prod_{j=1}^{n-1} K_{b_j}^{i_j} K_{b_{j-1,j}}^{N-i_j-i_{j-1}} \right] \quad (1)$$

assuming that $i_0 = 0$.

We are calculating all the different memory assignments for the intersections of the clusters, and calculating the number of possible distributions in each case. Note that, this scenario represents the worst case. As we deduced in Lemma 3.3, all clusters share at least one hash join operation with another cluster. Therefore, the situation where hash join operation are shared by at most two clusters is the less restrictive case and, as a consequence, the case presenting the largest number of possible memory distributions. As the number of clusters sharing a single hash join operation is increased, as it happens for example in Figure 1 where a hash join operation is shared by 3 clusters, this reduces the freedom to assign different memory values to a single operation. As a consequence, the number of possible combinations is decreased. Therefore, Equation 1 can be also used as an upper bound of the number of possible combinations.

Practical example

Figure 4 shows an example of a QEP for a query based on the TPC-H schema. This QEP is later used in the experiments section. As we can see in the figure, this QEP contains four different clusters. Assuming again that we have divided the available memory into 20 chunks (each chunk contains 5% of the available memory), we can calculate the number of possible memory distributions using Equation (1). In this example, the total number of memory distributions is 44506. Reducing the size of the memory chunks to be 2% of the total available memory (50 chunks), the number of combinations is 146541. Note that these are theoretical upper bounds. Thanks to the algorithm improvements commented before to prune unnecessary combinations, these figures are drastically reduced in practice, as we see in the experiments presented in the next section.

4. EXPERIMENTS

We implement our solution into the IBM database management system, DB2 UDB. Our tests are performed on SuSE Linux[®] x86_64 operating system. The machines have two Intel[®] Xeon[®] CPU 5160 (3.00GHz) with 4 cores.

We perform all the experiments on a 10 GB TPC-H [1] database, using two queries Q1 and Q2. Q1 is query 9 in TPC-H. Q2 is a TPC-H based query that joins all relations in the database to stress out our technique, by increasing the number of clusters. Figure 5 shows the QEP and the clusters for Q1. Figure 4, used in the previous section, shows the QEP and the clusters for Q2. Note that, clusters in both figures are sorted as explained in Subsection 3.4, and that,

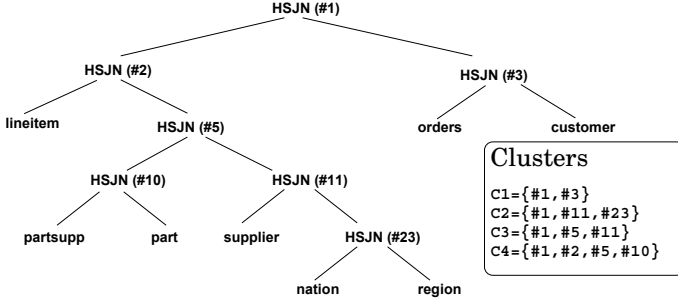


Figure 4: Example of a QEP on a query on the TPC-H schema and its clusters (C1, C2, C3, and C4).

following the conventions used in this paper, the build phase for a hash join (HSJN) corresponds to its right child, and the probe phase to its left child. For our tests, the size of the memory pool available is 100 and 140 MB for Q1 and Q2, respectively. This means that, in the case of the DB2 UDB baseline execution that divides the total amount of memory equally among all the hash join operations in the QEP, each hash join has a total amount of 20 MB available.

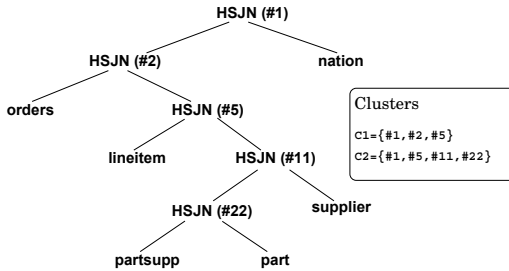


Figure 5: QEP and clusters (C1 and C2) for Q1.

In order to ensure a fair analysis, in addition to the baseline algorithm currently used by DB2, which does not consider clusters, we also compare the MCD algorithm to the algorithms proposed in [15]: EQUAL, BENEFIT, LINEAR and SMALL. All these algorithms were presented using the notion of shelves that only allowed for left-deep trees. The definition of a shelf in this previous work is a particular case of clusters when dealing with left-deep trees. EQUAL is a naive strategy that consists of dividing the available memory equally among all operators in a shelf. BENEFIT tries to favor those operators that will actually benefit the most from additional memory. In order to do this, BENEFIT is based on a very simple linear cost function. Note that, both EQUAL and BENEFIT look at only one shelf at a time and do not consider the global situation. LINEAR tries to remedy this by making its decision in one global optimization step treating the problem of memory allocation as a *linear programming* problem, and using the *simplex method* to solve it. Again this forces us to assume that the cost function is linear. Finally, the SMALL algorithm deals with the memory left over after giving all the operators in a shelf their minimum requirement, allocating more memory to operators that have a smaller demand.

For our experiments, we first compare these algorithms only considering left-deep trees, showing that in general they cannot compete with bushy trees and making it necessary to

use the notion of *cluster* presented in this paper. Afterwards, we analyze all the techniques again using clusters.

4.1 Bushy vs. left-deep tree plans

One important contribution of this paper is the generalization of complex memory allocation techniques to bushy trees. In this Section, we show that the bushy QEPs chosen by the DB2 UDB query optimizer to execute Q1 and Q2 does never generate a plan that takes longer to be executed than those generated by any memory allocation strategy applied to the best left-deep tree obtained by the optimizer for the same queries. Figure 6 shows left-deep tree query execution times for each memory allocation algorithm presented in [15]. The figure also shows the DB2 baseline query execution time for the left-deep tree plans, and for the QEPs in Figures 4 and 5, called *DB2.leftdeep* and *DB2.bushy* in this plot. The baseline using bushy trees not only outperforms the baseline using left-deep trees, but it is not outperformed by any of the memory allocation algorithms applied on left-deep trees, even when they take advantage of considering clusters but not the baseline. For this reason, from now on, we assume that the optimizer is open to the exploration of bushy trees. The comparison of the actual performance of the different algorithms is analyzed in the next subsections.



Figure 6: Query Execution time for previous memory allocation strategies and left-deep tree QEPs.

4.2 Compile phase complexity of MCD

In this section, we analyze the compile phase for the MCD algorithm presented in this paper. Table 1 shows the compile time, the actual number of combinations explored by the MCD algorithm and the theoretical upper bound explained in Subsection 3.4, both for Q1 and Q2. We test different chunk sizes ranging from 1% to 5% of the available memory, that is, from 100 chunks to 20 chunks. We do not test chunk sizes larger than 5% since, as we can see in the table, the time to calculate all the possible memory distributions is in general clearly lower than a second and, therefore, affordable during compile time. In addition, dividing the memory into less than 20 chunks would probably mean assigning close to the same fixed size of memory to all the operations, and that is already done by the baseline case. Note that, the larger the size of the chunk, the smaller the compile time. This is normal since fewer combinations are considered with larger chunk sizes. Also, we have tested chunks smaller than 1%. However, for this experiment, benefits were not significant. Note that the optimal chunk size may depend on the number of join operations in the QEP. Finding the formal relationship between them is out of the scope of this paper.

In addition, we compare the real number of combinations

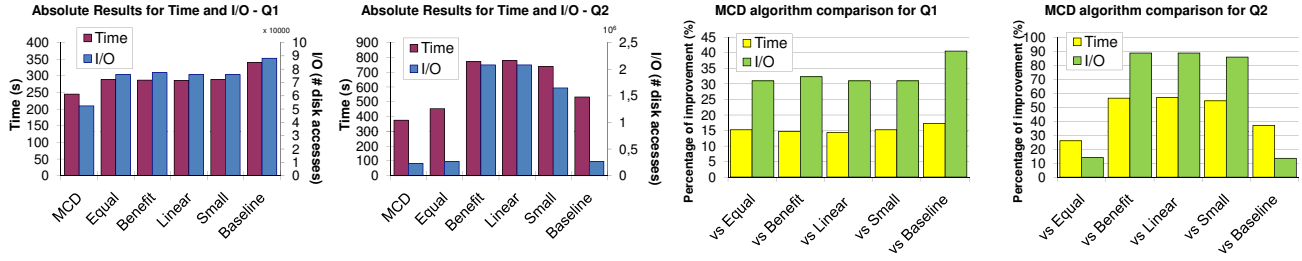


Figure 7: First two plots: I/O and time results for Q1 and Q2. Last two plots: Overall improvement of MCD for Q1 and Q2.

considered by our algorithm to the theoretical number of combinations. We can observe that pruning techniques significantly reduce the number combinations to be tested, especially when the number of chunks and clusters is large.

The compile times we obtain can be considered negligible compared with the whole query execution time. For these reasons, we only test the overall performance results using 100 chunks (1%).

	Compile time (sec)		Total # of comb.		Upper bound	
	Q1	Q2	Q1	Q2	Q1	Q2
chunks						
100	0.16	3.1	4851	389259	5151	22111526
50	0.03	0.47	1176	49788	1326	1465451
33	0.02	0.22	496	14548	595	295120
25	0.02	0.07	276	6390	351	102726
20	0.01	0.07	171	3209	231	44506

Table 1: MCD - compile phase for Q1 and Q2.

HSJN	Memory pages				
	MCD	Equal	Benefit	Linear	Small
1	12950	8750	200	200	29305
2	7350	8750	33913	33913	200
3	4808	4808	4808	4808	4808
5	687	687	687	687	687
10	7350	10302	200	200	10302
11	20	20	20	20	20
23	20	20	20	20	20

Table 2: Example of memory pages distribution for each hash join in Q2.

4.3 Algorithms Comparison

In this section, we discuss the I/O and query execution time results for the MCD algorithm, and compare it with the DB2 current algorithm, and to previous proposals proposed in [15] and explained in Section 4. Note that, in order to adapt these later memory allocation strategies to bushy trees, we run the experiments applying these strategies to the clusters obtained by Algorithm 1 proposed in this paper.

Table 2 summarizes the memory in pages of 4 KB assigned by each algorithm to each join operation in Q2, as an example. All the algorithms make a memory assignment that is significantly different to that made by MCD. The most expensive operations in this QEP are hash join number 1 and 2. As we can observe, while MCD and EQUAL algorithms assign a large number of pages to these two hash

join operations, the other three algorithms starve either one or the other, since they use a simpler cost model than that used in a real DBMS. Comparing MCD and EQUAL, the first prioritizes hash join number 1 in detriment of hash join number 10 which is less expensive. This is not surprising since EQUAL is a naive algorithm which is not cost-based.

For the results, we show both the I/O incurred by hash join operations, and the total query execution time.

The two upper plots in Figure 7 show, for each memory allocation strategy, the query execution time, and I/O both for Q1 and Q2. The lower plots in Figure 7, show the percentage of improvement of the MCD algorithm compared with the other approaches. These plots show that MCD clearly outperforms any other memory allocation technique. The reasons have been explained through the paper and might be different depending on the algorithm we compare with:

Cost-based vs heuristics. Through the plots in Figure 7 we can observe that compared to SMALL, EQUAL and BASELINE, the MCD algorithm improves significantly either the query execution time or I/O i.e., improvements over 30% and 80% in query execution can be observed for Q2. The main reason of the gain obtained by MCD is that either SMALL, EQUAL, or BASELINE use heuristics as opposed to the cost-based strategy followed by MCD.

Complex cost function vs. simple. We can observe that for query execution time, MCD outperforms LINEAR and BENEFIT by more than 30% and 80% for Q1 and Q2, respectively. The reason for these large differences is because LINEAR and BENEFIT algorithms depend on the simplicity of the cost function, ignoring the complexity of a real DBMS such as DB2 UDB. Thus, memory allocation is clearly sub-optimal, generating a large amount of I/O, and as a consequence, a high query execution time. Note that the fact that BENEFIT and LINEAR perform worse than the heuristic proposals is again because of the same reason: bad decisions due to imprecise cost-based functions cause large hash join memory starvation, degrading the system significantly. Note that the fact that LINEAR and BENEFIT perform worse in Q2 compared to Q1 is because the complexity of optimizing Q2 is higher and its hash join operations are larger in terms of I/O accesses. Therefore, making a mistake implies a larger impact on performance. These results show that it is essential to use a memory allocation algorithm that does not impose any strong condition on the cost function, since this might be unrealistic for most commercial DBMSs.

5. RELATED WORK

Prior research has tackled the problem of memory allocation among the operators in a QEP. Both static and dynamic

rule-based memory assignments are proposed in [4, 11]. Dynamic approaches can be very expensive, since a hash join might be ready to spill its entire hash table to disk, and re-store it later. In [6, 18] the authors propose autonomic memory allocation strategies that apply cost-based rules during query execution time. In this work the operators in QEP are still bounded by the memory calculated during compile time and it is thus sub-optimal. Note also, that they could be easily combined without interference with the work presented in this paper. Other works propose techniques to adapt the hybrid hash join algorithm to the available memory at runtime query execution [7, 16, 19]. These proposals are applied only to the operator by itself, and do not consider the global performance of the query execution plan. Thus, a hash join might consume all memory available, starving the rest of the operators in the QEP. In [15] the authors propose four different memory allocation methods that take advantages of the malleability of hash-based operations. These methods have been discussed and compared against the MCD algorithm proposed in this paper.

Memory allocation for diverse classes of workloads executed concurrently has been studied in [2]. In this work, the authors present a compile-time solution for the problem of memory starvation due to multiple query compilations at the same time. Also, for multi-user environments, in [12] the authors describe a parallelization method to carry out scheduling and mapping as well as memory allocation onto a shared-nothing architecture.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we tackled one of the critical aspects during complex query execution: memory utilization. Depending on the memory distribution, we can significantly improve DBMSs performance beyond the possibilities of current optimizers. Specifically, we showed the relevance of using an effective memory allocation algorithm at compile time for memory-intensive operations in multi-join queries. Our proposal is not restricted by the topology of the QEP like other previous proposals in the literature, but it can be used with any QEP, including bushy trees. In addition, the technique presented in this work is independent of the cost model used by the DBMS and its complexity. We showed that it is possible to reduce the total execution time of complex queries significantly, by studying the memory coexistence of the operations in the QEP, exploiting pipeline interruptions.

As the next step, we would like to extend the formalism in order to be able to (i) analyze the memory usage in more detail and (ii) propose new techniques that allow for different memory assignments to the same operation depending on its execution state at run time. We would also like to study the case where different operation types are competing for the resources in the same memory pool. Finally, we would like to extend our work to allow for very large join queries and inter-operator parallelism.

All in all, a correct memory allocation has to be able to adapt to the cost model of complex DBMSs. The use of heuristics, or simplified cost models for memory allocation turns into sub-optimal memory distributions. The adaptability to the topology of the query execution plan is also important since query optimizers cannot be restricted to a certain topology of the plan for complex queries.

7. REFERENCES

- [1] www.tpc.org.
- [2] B. Baryshnikov, C. Clinciu, C. Cunningham, L. Giakoumakis, S. Oks, and S. Stefani. Managing Query Compilation Memory Consumption to Improve DBMS Throughput. *CIDR*, 2007.
- [3] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Comm. of the ACM*, 13(7):422–426, 1970.
- [4] L. Bouganim, O. Kapitskaia, and P. Valduriez. Dynamic Memory Allocation for Large Query Execution. *Networking and Information Systems*, 1(6):629–652, 1998.
- [5] S. Chaudhuri and U. Dayal. Data warehousing and OLAP for decision support. *SIGMOD*, 1997.
- [6] B. Dageville and M. Zait. SQL memory management in Oracle9i. In *VLDB*, pages 962–973, 2002.
- [7] D. L. Davison and G. Graefe. Memory-Contention Responsive Hash Joins. *VLDB*, pages 379–390, 1994.
- [8] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD*, pages 1–8, 1984.
- [9] M. N. Garofalakis and Y. E. Ioannidis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. *VLDB*, pages 296–305, 1997.
- [10] J. Goodman. An Investigation of Multiprocessors Structures and Algorithms for Database Management. Technical report ucb/erl, m81/33, University of California at Berkeley, May 1981.
- [11] G. Graefe, R. Bunker, and S. Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. *VLDB*, pages 86–97, 1998.
- [12] A. Hameurlain and F. Morvan. CPU and incremental memory allocation in dynamic parallelization of SQL Queries. *Parallel Comput.*, 28(4):525–556, 2002.
- [13] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. *SIGMOD Rec.*, 20(2):168–177, 1991.
- [14] Kitsuregawa, M., Tanaka, H., and T. Moto-oka. Application of Hash to Data Base Machine and Its Architecture. *New Generation Computing, Vol 1, No 1*, 1983.
- [15] B. Nag and D. J. DeWitt. Memory allocation strategies for complex decision support queries. *CIKM*, pages 116–123, 1998.
- [16] H. Pang, M. J. Carey, and M. Livny. Partially preemptible hash joins. *SIGMOD*, pages 59–68, 1993.
- [17] J. M. Patel, M. J. Carey, and M. K. Vernon. Accurate modeling of the hybrid hash join algorithm. *SIGMETRICS*, pages 56–66, 1994.
- [18] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive Self-tuning Memory in DB2. *VLDB*, pages 1081–1092, 2006.
- [19] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. *VLDB*, pages 186–197, 1990.

Trademarks. IBM, DB2, and DB2 Universal Database are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Intel and Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Microsoft is a trademark of Microsoft Corporation in the United States, other countries, or both. Other company, product, or service marks may be trademarks or service marks of others.