Title:

Dynamic Adaptation in a Query Processing Web Service under Variable Memory Constraints

Henrique Paques, Calton Pu, and Ling Liu

Contact Author:
Henrique Paques
Software Design Tools Inc.
R. Lauro Vannucci, 1050, Campinas, SP 13087-548
BRAZIL
Phone: +55(19) 3296 5766
Email: paques@softwaredesign.com.br

Abstract

With the availability of ever increasing data, more sophisticated queries will take longer time, and more likely the statically-generated initial query processing plan will become suboptimal during execution. For example, selectivity estimates and main memory availability in the system may change during execution. These discrepancies cause additional I/O and thrashing that may seriously lengthen query processing time. In this paper, we apply and combine several apparently unrelated dynamic adaptation methods such as choosing sort-merge join when operands are sorted, switching the operands for hash joins when the left operand is found to be larger, and selecting memory allocation strategies (between Max2Min and AlwaysMax as appropriate). We use the Ginga service, an adaptive query processing web-service based on the notion of adaptation space, to organize and combine these adaptation methods systematically. We also use Ginga's query processing simulation system to evaluate the effectiveness of these methods in isolation and in combination. Our experimental results show that combined query adaptation can achieve significant performance improvements (up to 70% of response time gain) when compared to individual solutions.

Keywords: Adaptation space, memory constraint, query processing, web services.

Dynamic Adaptation in a Query Processing Web Service under Variable Memory Constraints

Henrique Paques
Software Design Tools Inc.
Campinas, SP 13087-548 BRAZIL
paques@softwaredesign.com.br

Calton Pu and Ling Liu
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332-0280 USA
{calton,lingliu}@cc.gatech.edu

Abstract

With the availability of ever increasing data, more sophisticated queries will take longer time, and more likely the statically-generated initial query processing plan will become suboptimal during execution. For example, selectivity estimates and main memory availability in the system may change during execution. These discrepancies cause additional I/O and thrashing that may seriously lengthen query processing time. In this paper, we apply and combine several apparently unrelated dynamic adaptation methods such as choosing sort-merge join when operands are sorted, switching the operands for hash joins when the left operand is found to be larger, and selecting memory allocation strategies (between Max2Min and AlwaysMax as appropriate). We use the Ginga service, an adaptive query processing web-service based on the notion of adaptation space, to organize and combine these adaptation methods systematically. We also use Ginga's query processing simulation system to evaluate the effectiveness of these methods in isolation and in combination. Our experimental results show that combined query adaptation can achieve significant performance improvements (up to 70% of response time gain) when compared to individual solutions.

1 Introduction

Despite the continued evolution of computer systems under Moore's Law, real world database systems almost always need more main memory than is available. This is the case for complex decision support queries involving large relations and also in multi-user environments where a number of concurrently executing queries compete for a finite amount of main memory. Careful memory management for query execution under memory constraints has been studied by several researchers, as summarized in Section 5.

During the execution of a complex query, the relationship between memory resource availability and query processing requirements may change for a number of reasons. On the query processing requirements side, for example, if intermediate join result sizes turn out to be significantly different from what is estimated at query plan generation time, the execution may thrash due to insufficient memory. Similarly, a large number of concurrent queries in the system may result in a query execution receiving fewer memory blocks than the requirements established at the query plan generation time. In this paper, we investigate several dynamic adaptation techniques that overcome the variable memory constraint bottlenecks when available memory no longer matches query processing requirements at runtime.

We have previously developed the Ginga service [9, 10] as a platform to study dynamic adaptation in web service providing query processing. In a previous paper [9], we studied the trade-offs of different adaptation strategies in query execution in the presence of network delays. In this paper, we use Ginga to explore three techniques to accomplish dynamic adaptation to get around variable memory constraints. First, we study the effects of changing the query plan by changing the join algorithms, for example, replacing a hash join with a sort-merge join when one of the operands is sorted. Second, we fix the order of operands of a hash join when the actual sizes contradict earlier expectations or estimation (the left operand should be smaller than the right operand). Third, we choose a memory allocation strategy appropriate for the query plan and memory available.

The first main contribution of this paper is the use of Ginga's concept of *adaptation space* [8] to organize and combine the three dynamic adaptation techniques to manage variable memory constraints, which we described in the previous paragraph, into a framework. This framework supports a systematic approach that combines several previously unrelated heuristics and techniques for query adaptation into a cost-based optimization solution.

The second main contribution of this paper is an experimental evaluation of the three adaptation techniques and their combinations using Ginga's dynamic query execution simulation facility. For a variety of representative system configuration and workloads with memory bottlenecks, we show significant improvements in response time under dynamic adaptation. Each adaptation method by itself provides modest gains as expected, and their combination achieves significant performance improvements, up to 70% of response time gain over the response time obtained by individual adaptation methods. These results show the promise of the Ginga approach to combine dynamic memory adaptation techniques in a systematic way.

The rest of the paper is organized as follows. Section 2 outlines the main components of Ginga and the concept of adaptation space. Section 3 summarizes the application of three adaptation techniques to overcome memory bottlenecks. Section 4 studies the performance of these techniques for representative scenarios. Section 5 summarizes related work and Section 6 concludes the paper.

2 Overview of the Ginga Service

2.1 Two-Phase Query Adaptation

Ginga [9] is an adaptive query processing service that divides query processing into two phases. At query submission time, in a proactive adaptation engagement phase, Ginga builds an initial optimized query plan P_0 for the input user query and generates some alternative execution plans $\{P_i, i = 1, \cdots, n\}$ that may be needed due to runtime variations in the environment. During query execution, in a reactive control phase, Ginga monitors the system resources availability through execution progress, determines *when* to change the query plan and *how* to adapt by choosing an alternative plan created in the proactive phase.

In Ginga, we organize the opportunities for adaptation and alternative query plans into an adaptation space. An adaptation space is a directed graph where the nodes are called *adaptation cases* and the arcs are called *adaptation triggers*. An adaptation_case(P_i) is a pair (P_i, adaptation_condition(P_i)), where P_i is a query plan and adaptation_condition(P_i) is the set of memory constraint predicates under which P_i was optimized (e.g., how many memory blocks have been allocated and the assumptions made about join operand sizes). An adaptation trigger is a triple (adaptation_event(P_i), adaptation_action(P_i, P_j), adaptation_condition(P_j)) where P_i is the current plan executing under adaptation_condition(P_i). When a change in system parameters invalidates a memory constraint predicate in adaptation_condition(P_i) (e.g., unexpectedly large intermediate results), adaptation_event(P_i) is fired. Then, adaptation_action(P_i, P_j) finds adaptation_case(P_j) such that adaptation_condition(P_j) has become valid by the adaptation_event. Since P_j is the plan optimized for the new set of memory constraint predicates, the adaptation_action switches from

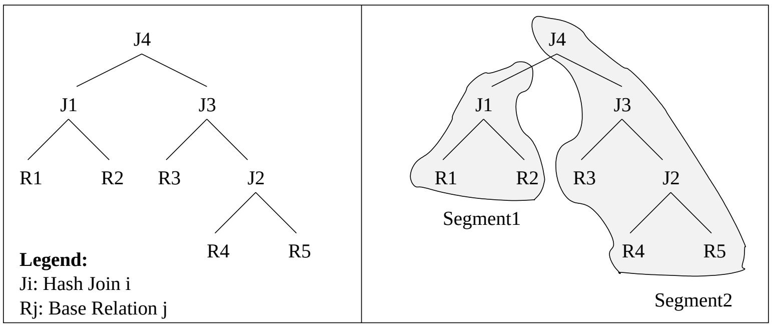
P_i to P_j by replacing code or runtime flags in all the nodes involved in processing this query.

During the proactive adaptation engagement phase, Ginga builds an adaptation space for executing query Q through three main steps. First, Ginga generates the initial query plan P_0 under the initial assumptions made about the runtime environment (e.g., expected memory size) in adaptation_condition(P_0). Second, Ginga creates the set of monitors that trigger an adaptation_event when one of the memory constraint predicates in adaptation_condition(P_0) becomes invalid. Third, for each such event, Ginga creates a new set of memory constraint predicates and an alternative query plan optimized for it (adaptation_condition(P_j)) and adaptation_trigger(P_0). This process is repeated for each one of P_j until all the memory constraint predicates have been covered. The construction of an adaptation space can be eager, as outlined above, or lazy, when the alternative query plans are generated only when needed. In general, the trade-offs in adaptation space generation are a subject for future research and beyond the scope of this paper. For a relatively small adaptation space, it is reasonable to generate it at initialization time. This is the assumption made in this paper.

During the reactive control phase, Ginga adapts the execution of a query Q by navigating the adaptation space. When P_i is executing, Ginga monitors the validity of memory constraint predicates in adaptation_condition(P_i). If one (or more) of memory constraint predicates in adaptation_condition(P_i) becomes invalid, an adaptation_event(P_i) is generated and adaptation_trigger(P_i) is fired. Ginga makes the transition in the adaptation space to the next query plan P_j by matching the new set of memory constraints with adaptation_condition(P_j). The query execution continues with P_j.

2.2 Query Execution Model

In Ginga, query plans are processed according to the Segmented Execution Model (SEM) as described in [12]. In this model, a query plan P_i is first decomposed into a set of right-deep segments. Then, a segment execution schedule, $sch(P_i)$, is generated by assigning numbers to each segment in a left-deep recursive manner. Segments are executed one at a time. For example, suppose that P_i is the query plan shown in Figure 1(a). According to SEM, P_i will be divided into two segments, $Segment_1$ and $Segment_2$ (Figure 1(b)), where $sch(P_i) =< Segment_1, Segment_2 >$. Segments are further represented as a sequence of operators. In this example, $Segment_1 =< J_1 >$ and $Segment_2 =< J_4, J_3, J_2 >$.



(a) Query Plan P_i (b) Segmented Query Plan P_i

Figure 1: Query plan decomposed into right-deep segments.

Segments are right-deep because right-deep trees allow us to pipeline the execution of hash joins and achieve inter-operator parallelism. If enough memory is available for constructing the hash tables (or at least part of them, if partitioning is needed) for the join operators involved in a right-deep (sub) tree, then we can execute these operators concurrently in a pipeline fashion. The advantage is that we can eliminate the need for caching intermediate results into disk (as needed) and consequently improve the query total response time. SEM assumes that join operators are hash-based

Considering that at any given moment there will be only one segment $s_k \in sch(P_i)$ being executed, in the rest of this paper, we study how to adapt the execution of s_k when there is a violation of memory constraint predicates in adaptation_condition(P_i) that affects the execution of s_k. Observe that by adapting s_k, Ginga generates a new query plan P_j.

3 Application of Ginga for Managing Memory Constraints

In this section, we first present three adaptation actions that Ginga can fire to accomplish dynamic adaptation to cope with variable memory constraints: (1) changing join algorithms, (2) switching operands of a hash join if left operand is larger than right operand, and (3) choosing a memory allocation strategy appropriate for the query plan and memory available. Then, we describe how to organize these actions into an adaptation space G, and how Ginga navigates G to get around the changes in memory constraints.

3.1 Changing Join Algorithm

In a relational database environment, three join algorithms have been commonly used: nested-loop join (NLJ), hash join (HJ), and sort-merge join (SMJ). When compared, NLJ has the worst performance. In general, HJ offers superior performance [11]. However, SMJ can outperform HJ if at least one of the operands is sorted (see Section 4.1). Based on this observation, we present our first adaptation action, the "Changing Join Algorithm" (Figure 2): Given s_k, the current segment to be executed from input plan P_i, for all join operator $op_j \in s_k$, if op_j is a HJ and at least one of its operands is sorted, then change op_j to SMJ.

```
ADAPTATION ACTION adapt_action_chgJoinAlgo
Input: $P_i$: current plan.
Output: $P_j$: plan $P_i$ adapted.
Let $s_k =< op_1, \cdots, op_n >\in sch(P_i)$ be the current segment to be executed.
$P_j$ = copyOf($P_i$); // Make a copy of $P_i$ to generate the adapted plan $P_j$.
// For each join operator in $s_k$, check whether we should change its method.
for $\forall op_i \in s_k$ do
    if ($op_i.method \neq$ sort_merge_join) and (isSorted($op_i.leftOperand$) or isSorted($op_i.rightOperand$)) then
        $op_i.method =$ sort_merge_join;
return $P_j$
```

Figure 2: Adaptation Action "Changing Join Algorithms."

3.2 Switching Operands of Hash Joins

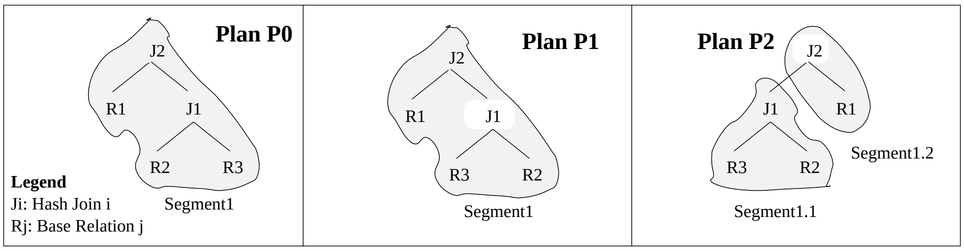
The adaptation action "switching operands of hash joins" is a simple one: if the actual size[1] of the left operand of a hash join is larger than the size of the right operand, then switch the operands. However, despite its simplicity, this action has interesting properties, as we illustrate next. We assume that operands are switched *before* starting the execution of the hash join. Figure 3 depicts the adaptation action for switching operands.

Example 1 Consider the query plan P_0 shown in Figure 4(a) where $sch(P_0) =< Segment_1 >$. Also, assume that sizeOf(R_2) > sizeOf(R_3) and sizeOf(R_1) > sizeOf(J_1), where sizeOf(X) returns the size of X in *bytes* and sizeOf(J_1) is the result size of J_1.

[1] The observed size at runtime of a base relation or intermediate result.

```
ADAPTATION ACTION adapt_action_switchJoinOprnd
Input: $P_i$: current plan.
Output: $P_j$: plan $P_i$ adapted.
 1: Let $s_k =< op_1, \cdots, op_n >\in sch(P_i)$ be the current segment to be executed.
 2: $P_j$ = copyOf($P_i$);
    // For each hash join operator in $s_k$, check whether we should switch its operands.
 3: for $i = n, \cdots, i$ do
 4:   Let $op_i \in s_k$ // Iterate thorugh the operators of $s_k$ in reverse order.
 5:   if ($op_i.method ==$ hash_join) and (sizeOf($op_i.leftOperand$) $>$ sizeOf($op_i.rightOperand$)) then
 6:     $aux = op_i.leftOperand$;
 7:     $op_i.leftOperand = op_i.rightOperand$; // Switching operands of $op_i$.
 8:     $op_i.rightOperand = aux$;
 9:     $op_i.method =$ hash_join_improved; // Flag that this join was improved.
        // Check whether the switching of operands will result in changing the shape of the query tree.
10:     if $op_i$ is not the last operator in $s_k$ then
11:       $P_j$.reshapeQryTree($op_i$); // Divide $s_k$ at $op_i$ and change the shape of the tree.
12:       break; // Interrupt the for-loop.
13: return $P_j$
```

Figure 3: Adaptation Action "Switching Operands of Hash Joins."



(a) Initial Segment (b) Switching operands of J1 (c) Switching operands of J2

Figure 4: Effects of switching operands.

When Ginga executes adapt_action_switchJoinOprnd, the current segment of P_0 to be executed is $Segment_1$ and J_1 is the first operator considered in the for-loop of Line 3. J_1 satisfies the condition in Line 4 and its operands are switched generating a new plan, P_1 (Figure 4(b)). A similar process is executed for the next operator J_2, resulting in query plan P_2 shown in Figure 4(c).

When Ginga switches the operands of J_1, the original tree shape of P_0 is not affected. However, by switching the operands of J_2, the original right-deep tree is transformed into a left-deep tree. In this case, Ginga needs to change $Segment_1$ by dividing it in two sub-segments, namely, $Segment_{1.1}$ and $Segment_{1.2}$, and updating the segment schedule of P_2 to $sch(P_2) =< Segment_{1.1}, Segment_{1.2} >$. The division of $Segment_1$ is necessary because we are using SEM, where each segment is a right-deep (sub) tree. By switching the operands of J_2, we violated this property. The process of detecting the change in shape of a query plan tree and updating the respective segment schedule is captured in Lines 9 and 10 (Figure 3).

The process of re-shaping the query tree by simply switching the operands of a hash join has two interesting properties. First, we can perform a simple re-optimization of the current query plan by improving the performance of hash joins. Second, if the size of the left operand is larger than expected, it is likely that the result of the associated hash join operator will also not match its estimated size. By changing the shape of the query tree, we are able to (1) stop the propagation of inaccurate estimated size of intermediate results up into the tree and (2) handle the estimation error when we schedule the next segment generated in the re-shaping process. This is possible because Ginga adapts each segment of a plan P_i as they are scheduled (see Section 2.2).

3.3 Choosing Memory Allocation Strategy

Our third adaptation action for coping with variable memory constraints chooses the appropriate memory allocation strategy for distributing the available memory among the operators of a segment so that the total query response time is minimized. Different memory allocation strategies were proposed in the literature [6, 2]. In this paper, we focus on two strategies demonstrated to be the most efficient in a number of different scenarios: Max2Min [6, 2], and the strategy proposed in [2], which we refer in this paper as AlwaysMax.

Briefly, *Max2Min* follows the heuristic described in [13]: in order to obtain better return on memory consumption, allocate the maximum amount of memory to operators with small maximum memory requirements. *AlwaysMax* attempts to always provide the operators with their maximum memory requirements. When there is not enough memory to match these maximum requirements, the segment is divided in two sub-segments, which will be executed sequentially.

```
ADAPTATION ACTION adapt_action_chooseMemAlloc
Input: $P_i$: current plan.
Output: $P_j$: plan $P_i$ adapted.
Let $s_k =< op_1, \cdots, op_n >\in sch(P_i)$ be the current segment to be executed.
$P_j$ = copyOf($P_i$);
// Calculate the cost of executing $s_k$ using Max2Min.
$minCost$ = segmentExecCost($s_k$, max2min, $memory\_size$);
$allocAlgo$ = max2min;
// Calculate the cost of executing $s_k$ using AlwaysMax.
$alwaysMaxCost$ = segmentExecCost($s_k$, alwaysMax, $memory\_size$);
// Choose the allocation strategy that yields the least cost.
if ($minCost \geq alwaysMaxCost$) then
    $allocAlgo$ = alwaysMax;
$s_k$.memAllocAlgo = $allocAlgo$; // Records the selected memory allocation strategy.
return $P_j$; // $s_k \in sch(P_j)$.
```

Figure 5: Adaptation Action “Choosing Memory Allocation Strategy.”

Figure 5 depicts the adaptation action for choosing the appropriate memory allocation strategy. Given s_k, the current segment to be executed from input plan P_i, Ginga first estimates the costs of executing s_k using Max2Min and AlwaysMax. Then, Ginga chooses the strategy that yields the least response time. An important parameter for estimating the execution cost of s_k is $memory\_size$. Considering that each segment is executed one at a time, $memory\_size$ is equal to the memory allocated by the Memory Manager for executing P_i. Ginga estimates the cost of executing s_k using the function `segmentExecCost` described in the Appendix.

3.4 Constructing and Navigating the Adaptation Space

So far, we have described three adaptation actions for coping with variable memory constraints during the execution of query plan P_i. In this section, we briefly describe how to organize these actions into an adaptation space G, and how Ginga navigates G to get around changes in memory constraints.

Ginga will adapt the execution of P_i when at least one of the memory constraint predicates listed in Table 1 is invalidated. The associated adaptation events and actions to be fired are summarized in Table 2. For example, if the predicate Operand_Predicate is invalidated, then event *large_operands* is fired and `adapt_action_switchJoinOprnd` is executed. Based on these events and actions, Ginga generates the adaptation space for P_i shown in Figure 6. We now describe how Ginga navigates this adaptation space.

Table 1: Memory Constraint Predicates for plan P_i, where $s_k \in sch(P_i)$ is the current segment to be executed.

Predicate Name	**Description**
Algo_Predicate	$\forall op_i \in s_k$, **if** $op_i.method == SMJ$, **then** `isSorted`$(op_i.leftOperand)$ == TRUE **or** `isSorted`$(op_i.rightOperand)$ == TRUE;
Operand_Predicate	$\forall op_i \in s_k$, **if** $op_i.method == HJ$, **then** `sizeOf`$(op_i.leftOperand) \leq$ `sizeOf`$(op_i.rightOperand)$.
MemorySize_Predicate	memory allocate to P_i is greater or equal to the expected memory size (i.e., the memory size assumed during the optimization process).

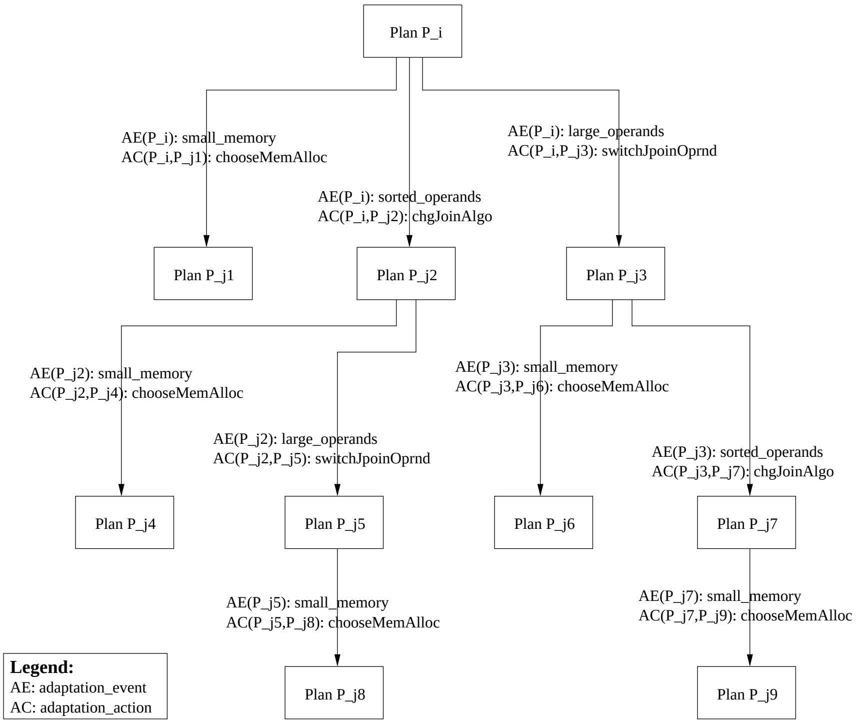


Figure 6: Adaptation space for coping with memory constraints.

Let $evnts(P_i)$ be the set of adaptation events of P_i that were fired, and let $comb\_evnts(P_i)$ be the set of all possible combination of these events. We call each combination $ap_k \in comb\_evnts(P_i)$ an *adaptation path*, which describes a possible path in the adaptation space that Ginga can navigate to find the appropriate alternative plan for P_i.

Example 2 Consider the adaptation space shown in Figure 6. Now, suppose that both Algo_Predicate and Operand_Predicate of plan P_i were invalidated, firing events *sorted_operands* and *large_operands*, respectively. In this case, we have $comb\_evnts(P_i) = \{$ (*sorted_operands, large_operands*); (*large_operands, sorted_operands*)$\}$. If Ginga follows the first adaptation path in $comb\_evnts(P_i)$, plan P_{j5} will be selected (see Figure 6). On the other hand, following the second adaptation path, Ginga will choose plan P_{j7}. The decision on which plan to use is made by comparing the cost of each plan. Ginga selects the plan with the least cost. The algorithm for navigating the adaptation space from Figure 6 is described in the Appendix.

It is important to observe that not all paths in $comb\_evnts(P_i)$ are relevant. For example, if both Algo_Predicate and MemorySize_Predicate are invalidated, the alternative plan generated by executing first

Table 2: Adaptation events and adaptation actions.

adaptation_event(P_i)	**Invalidated Predicate**	**adaptation_action(P_i, P_j)**
sorted_operands	Algo_Predicate	adapt_action_chgJoinAlgo
large_operands	Operand_Predicate	adapt_action_switchJoinOprnd
small_memory	MemorySize_Predicate	adapt_action_chooseMemAlloc

adapt_action_chooseMemAlloc followed by adapt_action_chgJoinAlgo is not a good solution. The selection of the appropriate memory allocation strategy is based on the memory requirements of the operator algorithms in the segment. If after the selection is done we change the operator algorithms, this will invalidate the decision made adapt_action_chooseMemAlloc. The adaptation space depicted in Figure 6 shows only the relevant adaptation paths.

When Algo_Predicate or Operand_Predicate are invalidated, MemorySize_Predicate may also become invalid. This occurs because as Ginga changes the configuration of join operators, the expected memory size for executing the new plan may also change. Consequently, *small_memory* may be fired if the modified join operators require more memory than what was originally allocated to P_i.

4 Performance Evaluation by Simulation

For all experiments reported in this paper, we use a simulator (based on the CSIM toolkit) that models a server machine running Ginga. Table 3 lists the main parameters used for configuring the simulator. All base relations involved in the queries submitted to Ginga are assumed to be available on the server's local disk. We use different sizes of base relations, where each tuple in a relation has 200 bytes.

The experiments are divided into three groups. The first group (Section 4.1) studies the effects of changing join algorithms or operand ordering. The second group (Section 4.2) studies the effects of changing memory allocation strategies. The third group (Section 4.2) studies the combination of different algorithms and strategies.

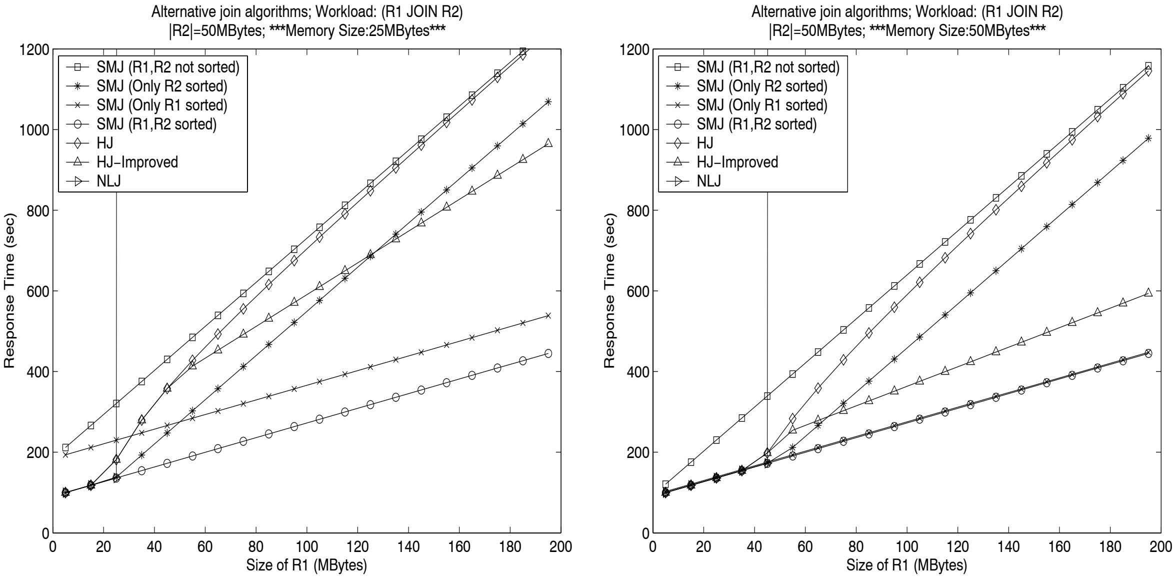
Table 3: Simulation Parameters

Parameter	**Value**	**Description**
Speed	100	CPU speed (MIPS)
AvgSeekTime	8.9	average disk seek time (msecs)
AvgRotLatency	5.5	average disk rotational latency (msecs)
TranferRate	100	disk transfer rate (MBytes/sec)
DskPageSize	8192	disk page size (bytes)
MemorySize	$10 \cdots 100$	memory size (MBytes)
DiskIO	5000	instructions to start a disk I/O
Move	2	instructions to move 4 bytes
Comp	4	instructions to compare keys
Hash	25	instructions to hash a key
Swap	100	instructions to swap two tuples

4.1 Join Algorithms

To study the effects of changing join algorithms and operand ordering, we use a simple workload: a single join $J = R_1 \bowtie R_2$, where we fix $|R_2| = 50MBytes$ and vary the size of R_1 from $5MBytes$ to $100MBytes$ using two different memory sizes, namely, $25MBytes$ and $50MBytes$. Arguably, more sophisticated joins will contain this elementary configuration as a subset and the results of this simple experiment will apply to them as well. Although the join algorithms are well known, we were surprised by the non-trivial differences among them. We ran experiments with a variety of operand and memory sizes to test the sensitivity of the parameter settings, and found similar results.

The graphs in Figure 7 show the response time of executing join J as a function of R_1 size using four different methods, namely, nested-loop join (NLJ), sort-merge join (SMJ), hash join (HJ), and hash-join improved (HJ-Improved). HJ-Improved switches the operands when the left is larger than the right operand. For sort-merge join, we study four subcases: with R_1 sorted, R_2 sorted, both sorted, and neither sorted.



(a) (b)

Figure 7: Response time of alternative join algorithms using two memory sizes: (a) 25MBytes and (b) 50MBytes.

When there is enough memory to fully execute J in main memory, all four join methods have similar performance. The difference between their response time stems from the different number of CPU instructions. However, as the size of the left operand (R_1) increases, requiring I/O operations, significant differences arise. First, as expected, NLJ has the worse performance. The vertical lines at 25MB in Figure 7 (a) and at 50MB in Figure 7 (b) show the well understood scalability problems of NLJ and we do not discuss it further.

For the main memory size of 50MB, Figure 7 (b) shows that SMJ is the fastest join method when R_1 or both R_1 and R_2 are sorted. It also shows that when only R_2 is sorted, SMJ is still better than HJ, but HJ-Improved now performs better than SMJ. This is an interesting observation, since SMJ is slower than HJ in the general case, when both R_1 and R_2 are unsorted. The figure shows that when one of the operands is sorted, it is worth switching to SMJ in most situations. Similarly, HJ-Improved is a significant improvement over static HJ, since HJ-Improved is able to cope with an increasingly larger size of left operand by always switching the smaller operand to the left.

For a smaller main memory size of 25MB, Figure 7 (a) shows that the relative performance of HJ, HJ-Improved, and SMJ becomes more intricate when the memory constraints are tighter. While the shape of the curves remained the same, vertical displacements changed the trade-offs among them. First, since R_2 does not fit in main memory anymore, when only R_1 is sorted, SMJ is significantly slower than when both operands are sorted. Second, the advantage of HJ-Improved starts later, losing to SMJ R_2-sorted for a significant range of smaller R_1 sizes.

4.2 Memory Allocation Strategies

To study the effects of different memory allocation strategies, we chose the simple example query plan P_0 (right-deep tree) depicted in Figure 8 with the following configuration: $|R_1| = 25MBytes$, $|R_2| = 50MBytes$, $|R_3| = 100MBytes$, and $|J_2| = 25MBytes$. We ran experiments with a variety of tree shapes, and operand and memory sizes to test the sensitivity of the parameter settings, and found similar results.

The graphs in Figure 9 show the response time of executing P_0 as a function of the memory size allocated to this plan using two different memory allocation strategies, namely, Max2Min and AlwaysMax. For a smaller intermediate result (50MB), Figure 9 (a) shows that AlwaysMax loses to Max2Min when main memory size is smaller (less than 20MB). The reason for that is because the intermediate result will not fit in main memory, and it will have to be completely cached into disk and read back again. For intermediate memory sizes (between 20MB and 75MB), AlwaysMax wins. This occurs because the extra I/O cost is amortized by the gain that we get by giving the maximum memory blocks to each operator. But, as we increase the size of the result of the J_2, even by amortizing the I/O cost, AlwaysMax will be no better than Max2Min (Figure 9(b)).

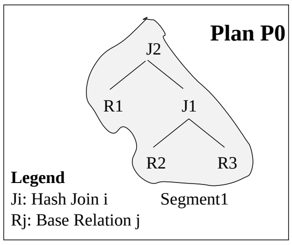
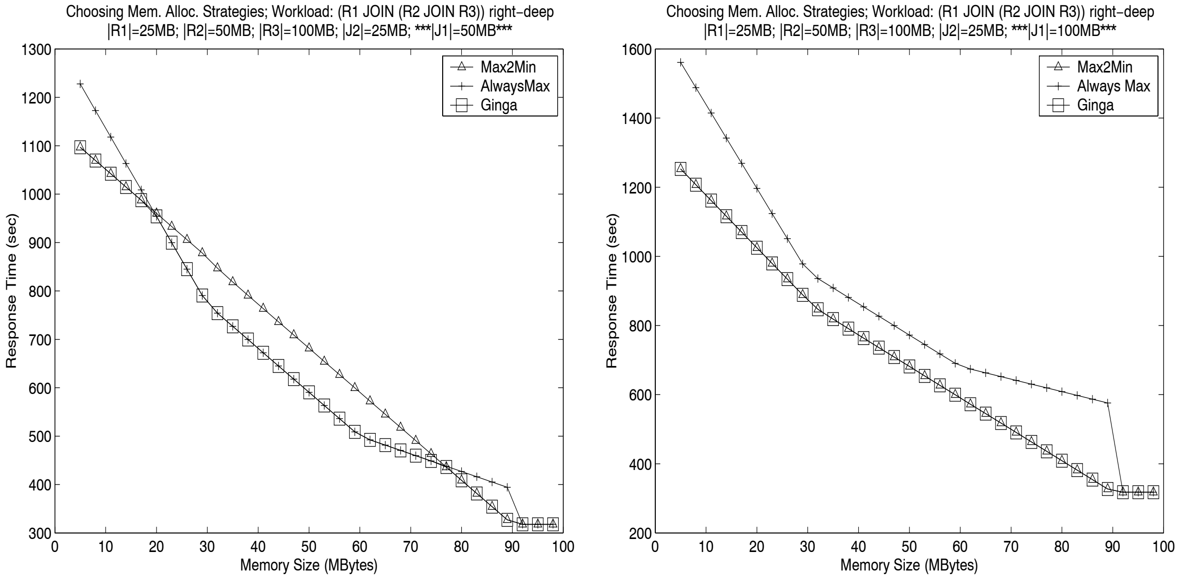


Figure 8: Query Plan P_0 used for the experiments described in Section 4.2 and Section 4.3.

In Figure 9(a) and (b), we can see 3 inflection points in the AlwaysMax curve, dividing it into 4 segments. In the first segment (up to about 30MB main memory), the response time rapidly decreases because each join operator is receiving all memory blocks that are available to execute P_0. The second segment (between 30MB and 60MB) the response time does not decrease as fast as before because now J_2 (the smaller of the two join operators w.r.t. memory requirements) receives all the memory that it needs to execute in its best performance (i.e., without incurring any I/O costs), but J_1 still does not receive its maximum required memory to also execute without I/O cost. The third segment shows intermediate results still present. Finally, when both joins can be executed in main memory, no more disk costs are necessary for caching the intermediate result. That explain why around 92MBytes we have a sudden drop in the response time for AlwaysMax. From this point on response time becomes dependent only on CPU cost for performing the actual joins and I/O costs for reading the base relation. Similar observations are applied to the graph shown in Figure 9(b).

In contrast, the Max2Min curve has one inflection point in Figure 9(a) and two in Figure 9(b). Up to 30MB of main memory, both operators increasingly receive more memory blocks, resulting in a fast decrease of the response time. For the smaller intermediate result, Figure 9(a), the same processing rate continues. However, for a larger intermediate result, Figure 9(b), the smaller join receives its maximum required memory, and the decreasing rate of the response time is given by the improvement in performance



(a) $|J_1| = 50MBytes$ (b) $|J_1| = 100MBytes$

Figure 9: Performance of alternative memory allocation under two scenarios as we vary the memory size: (a) $|J_1| = 50MBytes$; (b) $|J_1| = 100MBytes$.

of the execution of J_1, which receives more memory blocks. Finally, at around 92MBytes, the response time becomes dependent only on CPU costs.

Figure 9(a) shows a non-trivial relationship between the two memory allocation strategies. It is non-obvious when it is better to use one or the other. So we can conclude that indeed it is advantageous to choose the most appropriate memory allocation strategy for the current memory constraints. As it is clearly shown in this graph, there are moments where it is beneficial to use AlwaysMax and moments where Max2Min is a better alternative. A third line (the bottom most) illustrates the adaptive strategy, where it will always provide the response time using the memory allocation strategy that yields the fastest response time.

4.3 Combination of Adaptation Actions

One of the main results of using adaptation space is Ginga's ability to combine different adaptation strategies in a systematic way. Our next experiments combine the join method adaptations described in Section 4.1 with the memory allocation strategy adaptations described in Section 4.2. We again use query plan P_0 from Figure 8, but this time with the following configuration: $|R_1| = 25MBytes$, $|R_2| = 50MBytes$, $|R_3| = 100MBytes$, $|J_1| = 10MBytes$, and $|J_2| = 25MBytes$. Observe that with this configuration two adaptation events will be fired during the execution of P_0: *sorted_operands* and *large_operands*. We also assume that event *small_memory* is fired. We ran experiments with a variety of tree shapes, and operand and memory sizes to test the sensitivity of the parameter settings, and found similar results.

Table 4: Adaptation Paths for the Experiments

Adapt. Path	**Events**	**Sequence of actions**
1	(*memory_size*)	Choosing Memory Allocation Strategies
2	(*sorted_operands, memory_size*)	Changing Join Algorithm → Choosing Memory Allocation Strategies
3	(*large_operands, memory_size*)	Switching Hash Join Operands → Choosing Memory Allocation Strategies

The graphs in Figure 10 and Figure 11 show the response time of executing P_0 as a function of the memory size allocated to this plan when Ginga navigates the adaptation paths listed in Table 4. We omit the analysis of other adaptation paths such as (*large_operands, sorted_operands, memory_size*) and (*sorted_operands, large_operands, memory_size*) since they are subsumed by the cases in Table 4.

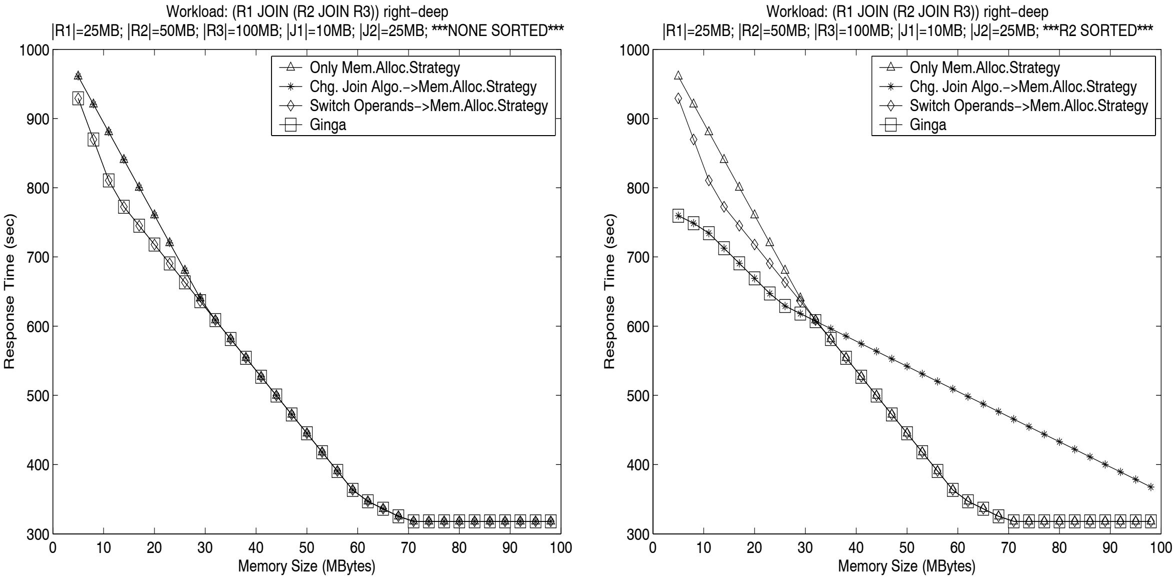
Figure 10(a) shows the benefits of switching the operands (Section 3.2) and called the HJ-Improved case in the figures in Section 4.1. (We include this case here to simplify the explanation of figure 10(b).) By switching the operands alone, Ginga can improve the response time modestly up to memory size of 30MBytes, when both operands fit in main memory and the response time of switching versus not switching the operands becomes the same. Figure 10(b) shows the effects of combining adaptive memory allocation with operand switching and changing to sort-merge join when one (or both) of the operands is sorted. As expected, if the sorted relation has a small size (e.g., R_1), the benefits are also small (not shown). When a larger relation (R_2) is sorted, Figure 10(b) shows an improvement of up to 20% compared to the reference line (adaptive memory allocation only). With an even larger relation (R_3) sorted, Figure 11(a) shows an improvement of up to 42% over the response time of the reference line. When all operands are sorted, Ginga shows an improvement in the response time of up to 70% for P_0 (Figure 11(b)), since the main memory requirements are minimized. The benefits of adaptation action "Changing Join Algorithms" (Adaptation Path 2) are maximized when the last operator (op_n) of a segment s is switched to SMJ. By changing op_n to SMJ, all join operators in s that depend on the result of op_n (i.e., all remaining operators $op_i \in s, 1 \leq i < n$) will also change to SMJ. Consequently, all SMJ join operators in s will have at least one operand sorted. As demonstrated in Section 4.1, with this configuration, SMJ outperforms HJ in most situations.

We now further analyze the benefits and trade-offs of adaptation action "Switching Hash Join Operands" (Adaptation Path 3) using the experiment results reported in Figure 12. The graphs in this figure show the response time of executing P_0 as a function of result size of J_1 using two different memory sizes, namely, 5MBytes and 10MBytes. The graph shows an improvement of up to 10%, when $|J_1| < |R_1|$. As $|J_1|$ becomes larger than $|R_1|$ (=25MB), Ginga will no longer switch the operands, and the response time becomes the same as the reference line (memory allocation only).

In Figure 12(a), the reference line (higher curve) has one inflection point, dividing it into two segments. In the first segment (up to $|J_1| =$13MB), Ginga chooses AlwaysMax as the memory allocation strategy. In this strategy, if each operator in the segment cannot receive their maximum required memory, the segment is cut, and the intermediate result between the two sub-segments must be cached into disk. AlwaysMax yields better performance than Max2Min for small intermediate result size. However, as the size of the intermediate result grows large, Max2Min becomes a better choice. In the second segment of the reference curve, Ginga uses Max2Min memory allocation strategy. Similar observations are applied to the graph shown in Figure 12(b).

In contrast, the lower curve for Adaptation Path 3 in Figure 12(a) has two inflection points, dividing it into 3 segments. In the first segment (up to 5MB), the hash table for the left operand of J_2, which now is J_1, can fit in main memory. In the second segment (between 5MB and 25MB), hash table for J_1 no longer fits in main memory, requiring increasingly I/O operations for execution J_2. In the third segment, because $|J_1| \geq |R_1|$, Ginga no longer switches the operands, and Adaptation Path 3 and reference line become the same. Similar observations apply for Figure 12(b).

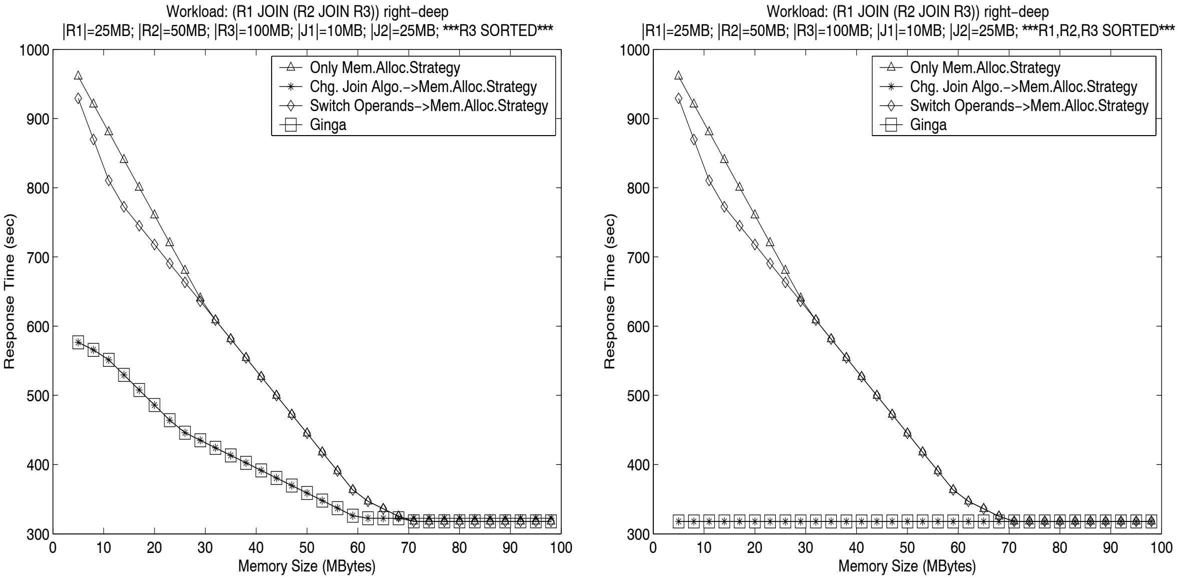
We have an interesting situation in Figure 12(a) when $17MB \leq |J_1| < 25MB$, where switching operands becomes worse than only selecting memory allocation strategy. The reason for that is the following. When Ginga switches the operands of J_2, query P_0 is transformed in the left-deep tree, which results in two segments, with a single operator each. Consequently, each operator will receive all memory available for its execution, and if the intermediate result does not fit in main memory, it will have to be cached into disk. For $5MB \leq |J_1| < 17MB$, the extra I/O due to caching $|J_1|$ is amortized by the improvement of switching the operands of J_2. However, when $|J_1| \geq 17MB$, the benefits of switching operands no longer



(a)

(b)

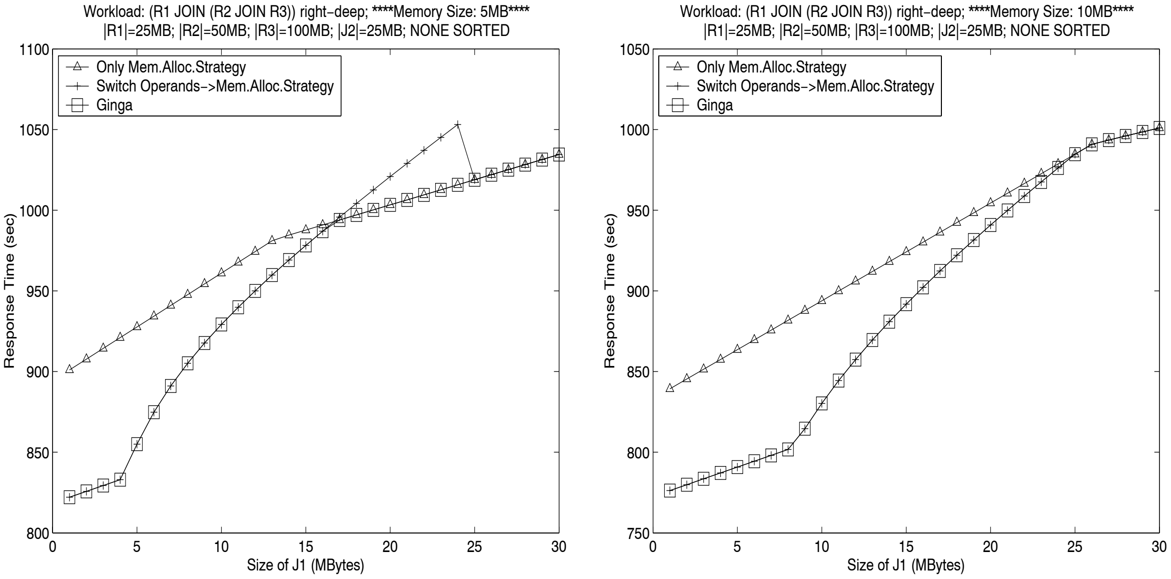
Figure 10: Performance of combining adaptation actions: (a) No operand is sorted; (b) Only R_2 is sorted.



(a)

(b)

Figure 11: Performance of combining adaptation actions: (a) Only R_3 is sorted; (b) All operands are sorted.



(a)

(b)

Figure 12: Performance of adaptation action Switching Operands, when no operands are sorted and memory size equal to (a) 5MBytes and (b) 10 MBytes.

pays off if we compare it with the reference line. At this point, the caching cost can no longer be amortized by the improvements for J_2, and the response time for Adaptation Path 3 becomes worse than the reference line. In Figure 12(b) this situation is not observed due to two reasons. First, with larger memory size (=10MB), J_1 and J_2 have better performance. Second, given the improved performance of the hash joins, the cost of caching $|J_1|$ can always be amortized by the improvement of switching the operands of J_2.

5 Related Work

There are three main problems related to memory management for query execution under memory constraints. The first problem focuses on how to allocate memory blocks to concurrent operators within a query in a way that will minimize the total query response time. Different approaches to memory allocation were proposed in the literature [6, 2] for addressing this problem. These approaches, which assume that query execution follows SEM, can be broadly classified into *static* and *dynamic* memory allocation strategies. The former is executed at start-up time whereas the later is applied at runtime, before executing each segment. [6] has proposed and evaluated four static memory allocation strategies, which included Max2Min (also called SMALL). As reported in [6], in general Max2Min presented best performance. However, the main problem with static strategies is that they allocate memory based on *estimated* size of intermediate results. If there is a significant difference between actual and estimated sizes of these results, query plans using static memory allocation strategies may have suboptimal performance. [2] addresses this problem with a dynamic strategy that we call in this paper AlwaysMax. As demonstrated in [2], in general AlwaysMax presents better performance than Max2Min. However, as shown in Section 4.2, there are situations where Max2Min can be a better choice than AlwaysMax. Ginga considers both strategies at runtime, before executing each segment, and selects the one that yields the best response time.

The second problem associated to managing memory constraints represents the situation where memory has been properly allocated to the query but the estimated sizes for the intermediate results were inaccurate, which may result in a significant increase of paging. Research work on query re-optimization [4, 1, 7] addresses this problem. Mid-query re-optimization [4] attempts to re-optimize the query execution whenever a significant difference between estimated and observed values for operators' selectivity is detected at runtime. Eddies [1] suggests the re-ordering of operators in the presence of configuration fluctuations of the runtime environment during query execution. [7] assumes that the estimated values for operators' selectivity and base relation sizes are not accurate and constructs the query plan at runtime as data become available. Ginga is similar to these approaches in the sense that it also provides query re-optimization. However, these approaches are limited to use only the adaptation actions that they propose. Extending these approaches to handle runtime changes other than inaccurate estimated values is a non-trivial task. In contrast, Ginga uses a unified simple model based on adaptation space that allows the incorporation of other dynamic adaptation methods to handle different runtime changes in memory constraints.

The last problem related to managing memory constraints is how to allocate memory blocks available in the system among concurrent queries, which is addressed by the research work on multi-query optimization [3, 13]. [3] proposes a global optimization where the idea is to optimize all query plans at once while determining how to distribute among the concurrent queries the memory blocks available in the system. [13] introduces the concept of *return on consumption* for studying the overall performance improvement of individual join queries due to additional memory allocation. The inclusion of multi-query optimization techniques into Ginga service is part of our future research work. In this paper, we assume that the Memory Manager will handle the problem of distributing the system memory blocks among the concurrent queries.

Methodologically, the Ginga service follows our previous research on program specialization [5]. In program specialization, we take advantage of quasi-invariants to eliminate unnecessary work. Similarly, Ginga uses the memory constraint predicates in adaptation conditions to optimize query execution. The monitoring of adaptation events is similar to the guarding of quasi-invariants, when an invalidation causes replugging of specialized code. Unlike program specialization, which relies primarily on program manipulation methods such as partial evaluation, Ginga uses many different techniques such as switching join algorithms, operand ordering, and memory allocation strategies to optimize query processing.

6 Conclusion

With the availability of ever increasing data, more sophisticated queries will take longer time, and more likely the statically-generated initial query processing plan will become suboptimal during execution. For example, inaccurate selectivity estimates may cause the left operand of a hash join to become larger than the right operand, causing the algorithm to require more main memory than necessary or additional I/O overhead. Similarly, as main memory availability changes due to system load, memory allocation strategies may slow down query processing at different degrees due to thrashing. While individual memory adaptation methods have been studied in the past, combining them has been a non-obvious task due to the non-trivial interactions among the methods and system components.

We have designed the Ginga service [9, 10] to investigate the effects of dynamic adaptation during query execution. In a previous paper [9], we studied the trade-offs of different adaptation strategies in query execution in the presence of network delays. In this paper, we focus on the adaptation techniques when query execution becomes suboptimal due to memory constraint changes (Section 3). While a number of unrelated adaptation techniques have been known to be effective in isolation, Ginga uses the concept of adaptation space to organize and combine them. In the case of memory constraints, we apply and combine several dynamic adaptation methods such as choosing sort-merge join when operands are sorted, switching the operands for hash joins when the left operand is found to be larger, and selecting memory allocation strategies, e.g., between Max2Min and AlwaysMax as appropriate.

We used Ginga's query processing simulation system to evaluate the effectiveness of these three methods in isolation and in combination (Section 4). Our experimental results confirm that each method improves query response time by reducing memory bottlenecks. More significantly, combined query adaptation can achieve significant performance improvements (up to 70% of response time gain for representative system and workload configurations) when compared to individual solutions. These results show that it is a good idea to combine adaptive methods to address memory constraints at runtime, and Ginga offers a systematic approach to organize and combine these methods in an effective way. The Ginga approach also offers promise for the incorporation of other dynamic adaptation methods to handle runtime changes in memory constraints.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continously adaptive query processing. In *ACM SIGMOD*, 2000.

[2] L. Bouganim, O. Kapitskaia, and P. Valduriez. Memory-adaptive scheduling for large query execution. In *CIKM*, 1999.

[3] D. W. Cornell and P. S. Yu. Integration of buffer management and query optimization in relational database environment. In *VLDB*, 1989.

[4] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD*, 1998.

[5] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2), 2001.

[6] B. Nag and D. DeWitt. Memory allocation strategies for complex decision support queries. In *CIKM*, 1998.

[7] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac. Dynamic query optimization on a distributed object management platforms. In *CIKM*, 1996.

[8] H. Paques, L. Liu, and C. Pu. Adaptation Space: A Design Framework for Adaptive Web Services. In *Proc. of International Conference on Web Services - Europe*, 2003.

[9] H. Paques, L. Liu, and C. Pu. Ginga: A self-adaptive query processing system. In *ACM Symposium in Applied Computing (SAC)*, 2003.

[10] H. Paques. The Ginga Approach to Adaptive Query Processing in Large Distributed Systems. Ph.D. Thesis, Georgia Institute of Technology, November 2003.

[11] L. D. Shapiro. Join processing in database systems with large main memory. *ACM TODS*, 11(3), 1986.

[12] E. J. Shekita, H. C. Young, and K.-L. Tan. Multi-join optimization for symmetric multiprocessors. In *VDLB*, 1993.

[13] P. S. Yu and D. W. Cornell. Buffer management based on return on consumption in a multi-query environment. *VLDB Journal*, 2, 1993.

A Cost Estimation Functions and Adaptation Space

```
**FUNCTION** estimatedExeCost
**Input:** (1) $P_i$: current plan; (2) $memory\_size$: memory allocated to execute $P_i$.
**Output:** $relativeCost$: cost of executing $P_i$ starting at $s_k$, the current segment to be executed.
**Let** $sch(P_i) =< s_1, \cdots, s_n >$
**Let** $s_k \in sch(P_i)$ be the current segment to be executed and $k \leq n$;
$relativeCost = 0$;
**for** $j = k, \cdots, n$ **do**
    $relativeCost = relativeCost +$ segmentExecCost($s_j, s_j.memAllocAlgo, memory\_size$);
**return** $relativeCost$;
```

Figure 13: Segment cost estimation.

```
**FUNCTION** segmentExecCost
**Input:** (1) $s_k$: segment to be executed; (2) $allocStrat$: memory allocation strategy for $s_k$;
    (3) $memory\_size$: memory allocated to execute $s_k$.
**Output:** $segCost$: cost of executing $s_k$.
**Let** $s_k =< op_1, \cdots, op_n >$;
allocate_memory($s_k, allocStrat, memory\_size$); // Allocate memory to operators in $s_k$ using $allocStrat$ strategy.
$segCost = 0$;
**for** $i = 1, \cdots, n$ **do**
    $segCost = segCost + op_i.cost()$; // $op_i.cost()$ is the cost function of HJ or SMJ.
**return** $segCost$;
```

Figure 14: Cost estimation a query plan, starting from the current segment to be executed.

```
**ADAPTATION SPACE** MgtMemoryConstraints
**Input:** adapt_space($P_i$): adaptation space for $P_i$.
**Output:** $P_j$: plan $P_i$ adapted.
**Require:** at least one of memory constraint predicates in adaptation_ condition($P_i$) is invalidate.
$minCost = \infty$;
// **Adaptation Path:** (*small_memory*)
**if** (MemorySize_Predicate($P_i$) == FALSE) **then**
    adapt_action_chooseMemAlloc($P_{j1}, P_j$);
    $minCost =$ estimatedExeCost($P_j, memory\_size$);

// **Adaptation Paths:** (*sorted_operands*); (*sorted_operands, small_memory*); (*sorted_ operands, large_ operands*); and (*sorted_operands, large_operands, small_memory*).
**if** (Algo_Predicate($P_i$) == FALSE) **then**
    adapt_action_chgJoinAlgo($P_i, P_{j2}$);
    **if** (MemorySize_Predicate($P_{j2}$) == FALSE) **then**
        adapt_action_chooseMemAlloc($P_{j2}, P_{j4}$);
        $cost_{j4}$ =estimatedExeCost($P_{j4}, memory\_size$);
        **if** ($minCost > cost_{j4}$) **then**
            $minCost = cost_{j4}$; $P_j = P_{j4}$; // **Adaptation Path:** (*sorted_operands, small_memory*)
    **else if** (Operand_Predicate($P_{j2}$) == FALSE) **then**
        adapt_action_switchJoinOprnd($P_{j2}, P_{j5}$);
        **if** (MemorySize_Predicate($P_{j5}$) == FALSE) **then**
            adapt_action_chooseMemAlloc($P_{j5}, P_{j8}$);
            $cost_{j8}$ =estimatedExeCost($P_{j8}, memory\_size$);
            **if** ($minCost > cost_{j8}$) **then**
                $minCost = cost_{j8}$; $P_j = P_{j8}$; // **Adaptation Path:** (*sorted_operands, large_operands, small_memory*)
        **else**
            $cost_{j5}$ =estimatedExeCost($P_{j5}, memory\_size$);
            **if** ($minCost > cost_{j5}$) **then**
                $minCost = cost_{j5}$; $P_j = P_{j5}$;// **Adaptation Path:** (*sorted_operands, large_operands*)
    **else**
        $cost_{j2}$ =estimatedExeCost($P_{j2}, memory\_size$);
        **if** ($minCost > cost_{j2}$) **then**
            $minCost = cost_{j2}$; $P_j = P_{j2}$;// **Adaptation Path:** (*sorted_operands*)

// **Adaptation Paths:** (*large_operands*); (*large_operands, small_memory*); (*large_ operands, sorted_ operands*); and (*large_operands, sorted_operands, small_memory*).
**if** (Operand_Predicate($P_i$) == FALSE) **then**
    adapt_action_switchJoinOprnd($P_i, P_{j3}$);
    **if** (MemorySize_Predicate($P_{j3}$) == FALSE) **then**
        adapt_action_chooseMemAlloc($P_{j3}, P_{j6}$);
        $cost_{j6}$ =estimatedExeCost($P_{j6}, memory\_size$);
        **if** ($minCost > cost_{j6}$) **then**
            $minCost = cost_{j6}$; $P_j = P_{j6}$;// **Adaptation Path:** (*large_operands, small_memory*)
    **else if** (Algo_Predicate($P_{j3}$) == FALSE) **then**
        adapt_action_chgJoinAlgo($P_{j3}, P_{j7}$);
        **if** (MemorySize_Predicate($P_{j7}$) == FALSE) **then**
            adapt_action_chooseMemAlloc($P_{j7}, P_{j9}$);
            $cost_{j9}$ =estimatedExeCost($P_{j9}, memory\_size$);
            **if** ($minCost > cost_{j9}$) **then**
                $minCost = cost_{j9}$; $P_j = P_{j9}$;// **Adaptation Path:** (*large_operands, sorted_operands, small_memory*)
        **else**
            $cost_{j7}$ =estimatedExeCost($P_{j7}, memory\_size$);
            **if** ($minCost > cost_{j7}$) **then**
                $minCost = cost_{j7}$; $P_j = P_{j7}$;// **Adaptation Path:** (*large_operands, sorted_operands*)
    **else**
        $cost_{j3}$ =estimatedExeCost($P_{j3}, memory\_size$);
        **if** ($minCost > cost_{j3}$) **then**
            $minCost = cost_{j3}$; $P_j = P_{j3}$;// **Adaptation Path:** (*large_operands*)
**return** $P_j$;
```

Figure 15: Adaptation Space for Managing Memory Constraints.