Domain-Partitioned Parallel Sort-Merge Join

by

Johan Gustav Larson

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Johan Gustav Larson
Domain-Partitioned Parallel Sort-Merge Join
Master of Science, 1995
Department of Computer Science
University of Toronto

Abstract

This thesis studies the utility of parallelizing sort-merge join in a shared-memory processing environment. The investigated algorithm partitions the work of the sort-merge equi-join by dividing the domain of the join attribute into a set of non-overlapping ranges. Two distinct ways of sorting the relations in parallel for the sort-merge join are investigated: one, based on a concurrent heap used for replacement selection, and a second, using in-memory sorting.

The investigation includes both theoretical analysis and implementation results, with direct comparability to previous work on parallel hash- and nested loops joins. Both processor- and I/O-bound environments are considered. It is concluded that parallel sort-merge join is inferior to hash-based join algorithms unless the joining relations are already sorted. Parallel run forming in a shared-memory environment, and the parallel sorting of arrays with large elements are also discussed.

Acknowledgements

Thanks are in order to the taxpayers of Canada, who funded the author during his studies through the Natural Sciences and Engineering Research Council.

Credit is also due Prof. P.-Å. Larson, for suggesting the topic of parallelizing the sort-merge join algorithm, as well as the notion of joining attribute value ranges. Prof. Larson also arranged access to the Sequent multiprocessor at the University of Waterloo that was used for the experiments conducted for this thesis.

Prof. K.C. Sevcik, the author's supervisor, deserves credit for holding a firm hand on the tiller of this project, and for showing no fear when facing early drafts of this thesis. The author is indebted to Drs. Larson and Sevcik for much advice and patience in the course of the project.

Finally, the author wishes to thank Prof. T. Abdelrahman, for generously agreeing to be the second reader of this thesis.

Contents

1 Introduction **1**

1.1 Background and Motivation 1

1.1.1 Other Work 3

1.1.2 The Companion Paper 4

1.2 Thesis Overview 11

2 Domain-Partitioned Parallel Sort-Merge Join **14**

2.1 Classic Sort-Merge Join 14

2.2 Overview of the Parallel Version 15

2.2.1 Phases of Processing 16

2.2.2 Secondary Information Collected 17

2.3 Heap-Based Run-Forming Phase 18

2.3.1 Use of the Heap 20

2.3.2 The Concurrent Heap 22

2.3.3 Use of the Tuple Buffer 25

2.4 In-Memory Run-Forming Phase 28

2.4.1 Three Components 28

2.4.2 Permutation Module 29

2.4.3 Flush Module 35

2.5 Calculation Phase 36

2.5.1 Selecting Partition Values 37

2.5.2 Computing Partition Blocks 39
2.6 Join Phase 40
2.6.1 Reading and Merging the Runs 42
2.6.2 The Actual Merge-Join 42

3 Performance Study **45**
3.1 Heap Based Run-Forming 45
3.1.1 Processor Time 45
3.1.2 I/O requirements 50
3.2 In-Memory Run-Forming 50
3.2.1 Processor Time 52
3.2.2 I/O Requirements 53
3.3 Joining Phase 57
3.3.1 Processor Time 57
3.3.2 I/O Requirements 60
3.4 Integration of the Models 60
3.4.1 Join Phase Modeling 62
3.4.2 Sort Phase Modeling 63

4 Experiments **64**
4.1 Description of the Experiments 64
4.1.1 Experiments with Real I/O 65
4.1.2 Experiments with Simulated I/O 66
4.2 Performance of the Algorithms 67
4.2.1 Experiments with Real I/O 67
4.2.2 Experiments with Simulated I/O 72
4.2.3 Summary 76
4.3 Accuracy of Performance Models 77
4.3.1 Aggregate Estimates 77

4.3.2 Heap-Based Run-Forming 80
4.3.3 Joining 82
4.3.4 Sorting 82
4.3.5 Permuting 83
4.3.6 Flushing 83
4.3.7 Overview Of Prediction Accuracy By Phase 84

5 Other Issues **90**

5.1 Buffer Use 90
5.1.1 Buffer Use Limits 91
5.1.2 Buffer Use Problems of Sort-Merge Join 92
5.1.3 Eliminating Double Buffering 93
5.2 Sort-Merge Joins with Indexes 94
5.2.1 Conditions 94
5.2.2 Division of Index-Hybrid Join 95
5.2.3 The First Merge-Join: A and XB 96
5.2.4 The Second Merge-Join: IT and B 99
5.3 Pre-Sorted Relations 100
5.3.1 Pre-Sorted Join versus Standard Hash-Based Join 101
5.3.2 Pre-Sorting versus Pre-Hashing 104
5.3.3 Conclusions Regarding Pre-Sorting 108

6 Conclusions **110**

6.1 The Algorithm 110
6.2 Experiments Performed 111
6.3 The Models and their Performance 111
6.4 How to Form Runs 112
6.5 The Verdict on the Algorithm 114
6.5.1 Four Weaknesses of the Algorithm 114

6.5.2 Agreement with Past Research 115
6.6 Applicability of This Research 117
6.6.1 Range of Applicability of the Algorithm 117
6.6.2 Applicability to Larger Relations 119
6.7 Further Research 120
6.8 Hindsight 121

A The Flush Module's Processing Requirements 126

B The Permuter Process 130

C Experiment Results: Proportion Analysis 136

D Experiments Regarding Heap Processing Costs 142

List of Figures

1.1 Experiments in Martin et al. paper 9

2.1 Phases of Sort-Merge Join 16

2.2 Pseudocode for In-Memory Run-Forming 17

2.3 Information Flow in Heap-Based Run-Forming. 19

2.4 Pseudocode of Heap Adjustment, with Locking 23

2.5 Processing Time (10,000 tuples, heap height 16) 24

2.6 Relation, with corresponding index and its graph representation. . . . 31

2.7 Pseudo-code for Permute. 33

2.8 Cycle Clipping. 34

2.9 Partitioning. 37

2.10 Pseudocode for Joining. 44

3.1 Average Filling Cost Per Tuple Per Processor 47

3.2 Prediction of *fill* Relative to Average Filling Cost 47

3.3 Average Emptying Cost Per Tuple Per Processor 49

3.4 Prediction of *empty* Relative to Average Filling Cost 49

3.5 Sequential I/O Routines' Performance (MB/s) 51

3.6 Execution Time for the Sort Module Per Tuple 54

3.7 Execution Time for the Permute Module Per Tuple 54

3.8 Permutation Efficiency (P=256, M=320) 55

3.9 Time Per Flush (B=256, M=50, size=4M) 56

3.10 Flush Processing Speed (B=256, P=4, relation size = 2M) 56
3.11 Flush Module Optimal W Values 57
3.12 Flush Time 58
3.13 Flush Speed(MB/s) (P=4, relation size=2M) 59
3.14 Random Reading Speeds(MB/s) 61
3.15 Random Reading Speed for Experiments 62

4.1 Experiments with Real I/O 65
4.2 Relations for Experiments with Simulated I/O 66
4.3 Parameters for Experiments with Simulated I/O 66
4.4 Parameters for Martin Models 67
4.5 Total Time for ProcessorVaryRealIO Experiments 68
4.6 Total Time for SizeVaryRealIO Experiments 68
4.7 Total Time for RatioVaryRealIO Experiments 69
4.8 Division of Running Time of ProcessorVaryRealIO Experiments with In-Memory Run-Forming 69
4.9 Division of Running Time of ProcessorVaryRealIO Experiments with Heap-Based Run-Forming 70
4.10 Total Time for ProcessorVaryNoIO Experiments 73
4.11 Total Time for MemoryVaryNoIO Experiments 73
4.12 Division of Running Time of ProcessorVaryNoIO Experiments with Heap-Based Run-Forming 74
4.13 Division of Running Time of ProcessorVaryNoIO Experiments with In-Memory Run-Forming 75
4.14 Aggregate Prediction Accuracy for ProcessorVaryRealIO 78
4.15 Aggregate Prediction Accuracy for SizeVaryRealIO 78
4.16 Aggregate Prediction Accuracy for RatioVaryRealIO 79
4.17 Aggregate Prediction Accuracy for ProcessorVaryNoIO 79
4.18 Aggregate Prediction Accuracy for MemoryVaryNoIO 80

4.19 Prediction Accuracy for ProcessorVaryRealIO Experiments, Run-Forming Phases 81
4.20 Prediction Accuracy for SizeVaryRealIO Experiments, Run-Forming Phases 85
4.21 Prediction Accuracy for ProcessorVaryNoIO Experiments, Run-Forming Phases 86
4.22 Prediction Accuracy for MemoryVaryNoIO Experiments, Run-Forming Phases 87
4.23 Prediction Accuracy for ProcessorVaryRealIO Experiments, Join Phase 87
4.24 Prediction Accuracy for SizeVaryRealIO Experiments, Join Phase . . 88
4.25 Prediction Accuracy of ProcessorVaryNoIO Experiments, Join Phase 88
4.26 Prediction Accuracy for MemoryVaryNoIO Experiments, Join Phase . 89

5.1 Buffer Use Analysis Parameters 91
5.2 B+-tree, Whole Structure. 97
5.3 B+-tree, Interior and Exterior Nodes. 97
5.4 ProcessorVaryRealIO Series Pre-Sorted Times 101
5.5 RatioVaryRealIO Series Pre-Sorted Times 103
5.6 I/O Activity, by Phase 104
5.7 ProcessorVaryNoIO Series Pre-Sorted Times 105
5.8 MemoryVaryNoIO Series Pre-Sorted Times 105
5.9 ProcessorVaryRealIO Series Pre-Sorted and Pre-Hashed Times 107
5.10 RatioVaryRealIO Series Pre-Sorted and Pre-Hashed Times 108
5.11 ProcessorVaryNoIO Series Pre-Sorted and Pre-Hashed Times 109
5.12 MemoryVaryNoIO Series Pre-Sorted and Pre-Hashed Times 109

A.1 Flush Processing Time (usec/tuple) 127
A.2 Flush Processing Time(microseconds) 129

C.1 Division of Time for SizeVaryRealIO Experiment with In-Memory Run-Forming 137
C.2 Division of Time for RatioVaryRealIO Experiment with In-Memory Run-Forming 138
C.3 Division of Time for SizeVaryRealIO Experiment with Heap-Based Run-Forming 139
C.4 Division of Time for Gamma Experiment with Heap-Based Run-Forming 139
C.5 Division of Time for MemoryVaryNoIO Experiment with Heap-Based Run-Forming 140
C.6 Division of Time for MemoryVaryNoIO Experiment with In-Memory Run-Forming 141

D.1 Heap Processing Costs(milliseconds), Full Locking (10,000 Tuples) . . 143
D.2 Heap Processing Costs(milliseconds), Locking Top Element Only (10,000 Tuples) 143
D.3 Heap Processing Costs(milliseconds), No Heap Adjustment (10,000 Tuples) 144
D.4 Heap Processing Costs(milliseconds), No Tuple Copying (10,000 Tuples) 144

Chapter 1

Introduction

1.1 Background and Motivation

Relational databases manipulate tables of records, with each record consisting of fields, such as integers or strings. Tables, records, and attributes are termed "relations", "tuples", and "attributes" in the field of relational databases. The join operation is one of the fundamental operations of the relational data model. In the most general terms, the join is defined as a subset of the cartesian product of two relations. The cartesian product of two relations A and B is in turn defined as every combination (α, β) of tuples such that $\alpha \epsilon A$ and $\beta \epsilon B$.

The most common form of join is the single-attribute equi-join. For example, consider two relations `Customer` and `CityData`, with `Customer` having attributes

`(Name, Street_Address, City, Postal_Code)`

and `CityData` having attributes

`(City, Population, Country).`

For these relations, a single-attribute equi-join could be used to answer the question "In what countries do we have customers?" First, `Customer` and `CityData` would be joined on on the `City` fields of `Customer.City` and `CityData.City`. This would generate the set of tuples

`(Name, Street_Address, Customer.City, Postal_Code, CityData.City, Population, Country) where Customer.City = CityData.City.`

This is a "single-attribute" join since only one attribute in each joining relation considered when determining which tuples are members of the join result. It is an "equi-join", since the relationship between the two attributes is equality. From the set that is the join result, the unique values of the `Country` attribute can be chosen to answer the question.

In practice, the join operation is an extremely common and expensive part of non-trivial database queries. General join processing has been surveyed by Mishra and Eich [14] and joins as a general component of relational database processing have been surveyed by Graefe [8]. There are basically three techniques for processing non-trivial equi-joins: nested loops, hashing, and sort-merge join. With the increasing popularity of parallel computers, the issue of parallel execution in database joins has become a topic for research.

The work of this thesis is based on a paper by Martin, Larson, and Deshpande [13], in which the authors analyzed and implemented three hash-based join algorithms in a shared memory environment. The main thrust of this thesis is a parallel version of sort-merge join. The most significant feature of the algorithm is the partitioning of the equi-join by join-attribute values. The algorithm is implemented in the same system, and generally using the same methods, as used by Martin et al., to generate fully comparable results. In fact, the performance of the parallel sort-merge join is compared against the performance of the three algorithms, based on the performance estimation function from the paper. Essentially, this thesis augments the treatment of hash-based joins by Martin, Larson and Deshpande to include the parallel sort-merge join.

1.1.1 Other Work

The notion of parallelizing sort-merge join, and comparing its performance against parallel versions of other join algorithms is not new. The verdict on the performance of parallel sort-merge join to date has been mixed. Valduriez and Gardarin [21] analyzed the performance of parallel versions of nested-loop, hash, and sort-merge joins. The analysis was for a specialized architecture, and based on a theoretical model which did not overlap processor- and I/O-times. They found that parallel sort-merge join was the best algorithm for cases with very large relations, but that it did not exploit large number of processors well. DeWitt and Gerber [6] also compared sort-merge join and hashing on a specialized parallel architecture. They found hashing to be superior, but their results are incomplete, since they did not include results for a parallel implementation of sort-merge join. Wolf, Dias, and Yu [22] addressed the issue of parallel sort-merge join, focusing on the problem of data skew [1] in a shared-nothing or highly-NUMA environment. They assumed joining would be processor-bound, and focused on a complex iterative method for partitioning to minimize data transfer between the processors. Richardson, Lu, and Mikkilineni [17] compared two types of parallel sort-merge join and four types of parallel hash-based joins using a complicated theoretical model of a collection of single-bus shared-memory multiprocessors. They concluded that sort-merge join is generally worse than hashing, but they did not partition by join-attribute values. Finally, Qadah and Irani [15] also studied the performance of nested-loop, hashing, and sort-merge algorithms using a complicated model of a parallel database machine, focusing on the use of various types of filtering.

No study using performance data from an actual implementation of a parallel sort-merge join algorithm for shared memory has yet appeared in the literature. Also, no

[1] Data skew refers to an uneven distribution of join-attribute values. It is a serious problem for parallel join algorithms, since many of them partition the work of the join on the basis of join-attribute values. When this distribution is uneven, the partitions become of uneven size, leading to poor load-balancing.

study has yet analyzed parallel sort-merge join using models that assume overlapped processor- and I/O-times. This thesis includes both of these.

There have been a few other papers relevant to various aspects of this thesis, and to hash-based parallel joins in particular, although they have not addressed parallel sort-merge joins directly. In his paper on database systems with large main memories, Shapiro [20] mentioned that sort-merge joins are generally the best method of performing non-index joins on sequential architectures. In discussing the sizes of modern memories, he also ventured a guess that relations in the future would be relatively small compared to the memory available (i.e., larger by only a small multiple). It is also noteworthy that the models he used did not include overlapped I/O- and processor-time. Wolf, Dias, Yu, and Turek have published a paper [23] extending their complex iterative algorithm of balancing the load of skewed joins in shared-nothing environments from sort-merge joins to hash-based joins. DeWitt et al. [7] addressed the issue of joins where almost the entire relations fit into memory. They were surprised to find that hash-based joins performed better than sort-merge joins under these conditions. Finally, Graefe and Thakkar [9] discussed the tuning of a parallel database algorithm–external sort, in particular–for performance. Most of their gains were realized by increasing the efficiency of buffer use.

1.1.2 The Companion Paper

Introduction

The Martin, Larson, and Deshpande paper [13] analyzed three hash-based algorithms for equi-joins on a shared-memory system with a memory large enough to store a substantial fraction of the relations. The algorithms were Hashed Loops, GRACE, and Hybrid. These algorithms had been analyzed earlier without implementation by Lu, Tan and Shan [12]. Each of the algorithms was analyzed using a model that included both processor-time and I/O-time estimates, and which assumed full overlap of I/O and processor-time for each phase of each algorithm. Each algorithm was also implemented on a Sequent Symmetry [19] system using the μSystem [2] package to allow the use of lightweight threads. The actual running times were investigated for both uniformly distributed and skewed relations. Graphs were plotted to compare the model results and actual execution times, while varying the amounts of memory and numbers of processors. The authors concluded that the model was sound; it predicted performance within 25% of the actual value. They furthermore concluded that Hybrid was the best algorithm for small amounts of memory (less than 20% the size of the smaller relation), that Hashed Loops was the best algorithm for moderate amounts of main memory (20-80% the size of the smaller relation), and that Hybrid and Hashed Loops had similar response times for larger main memories. The excellent performance of Hashed-Loops was a surprise, given the fundamentally quadratic nature of its I/O costs.

The hardware platform was a Sequent Symmetry multiprocessor with 32 MB of memory, ten Intel 80386 processors on a bus, and four disks usable for experimentation. The authors developed the software based on the model of lightweight tasks using the μSystem lightweight thread package. Thus, the authors were able to write programs with many small specialized tasks which spent much of their time waiting on semaphores for some condition to become true.

Algorithms

Hashed Loops The Hashed Loops algorithm is a variant of the traditional nested loops join algorithm. Essentially, the algorithm fills up the memory with a portion of the R (smaller) relation, organized into a hash table, and then scans through all of S (the larger relation), probing the hash table for each tuple of S, and writing out a result tuple whenever a match is found. Since R is typically larger than main memory, this process is repeated several times, each time building the hash table with a different portion of R, and scanning through all of S.

The chief disadvantage of this method is its quadratic nature; the amount of reading is proportional to the product of the sizes of R and S, since the number of iterations is proportional to the size of R, and all of S has to be read for each iteration. Thus, the cost of this algorithm would be prohibitive for relations that are very large compared to the size of main memory, but given the decreasing costs of main memory, substantial memories are expected to be the norm [20]. This algorithm has two advantages. The first of these is its use of sequential I/O. Assuming the R and S relations are each stored contiguously, then the reads of R and of S are sequential, which avoids the high seeks costs of random-access I/O. The second advantage is skew resistance. Since there is no partitioning of the relations, data skew cannot degrade the performance of the algorithm; the algorithm performs exactly the same operations on R and S regardless of the distribution of join-attribute values in the two relations.

The description above addresses a sequential implementation; the issue of how the algorithm is parallelized must be considered. The parallelization is on a per-block basis. There are parallel build tasks that take R-blocks as they are read in, and for each block insert the tuples therein into the single shared table (with appropriate synchronization to maintain the coherence of the table). The operation on the S-relation's blocks are similarly parallelized, with join tasks taking S-blocks as they are read in, and probing the hash table for each tuple in the blocks. The processors must be synchronized with a barrier at the end of processing each memory load of R-tuples.

GRACE The GRACE algorithm avoids the quadratic I/O cost associated with Hashed Loops by partitioning the relations R and S into B buckets each, $\{R_1, R_2, \ldots, R_B\}$ and $\{S_1, S_2, \ldots, S_B\}$, such that each R-bucket fits in memory (assuming R to be the smaller relation.) The partitioning is done by hashing on the join-keys. Then, for each $1 \leq i \leq B$, the algorithm performs a Hashed-Loops join on buckets R_i and S_i. Assuming the partitioning works correctly, each such Hashed-Loops invocation will be able to fit all of R_i into memory, meaning that S_i only has to be read through once. Thus, since the S-buckets are disjoint partitions of S, each tuple of S only has to be read in twice (once during the partitioning, and once during the Hashed Loops invocation), rather than multiple times, as in the full Hashed Loops algorithm.

The advantage of GRACE is its linear I/O cost. Each tuple is read or written three times: once to read each tuple, once to write to its bucket, and once to read for the final join, assuming the partitioning works correctly. Thus, the amount of reading and writing is linear in the sum of the sizes of R and S, rather than being related to the product of the relation sizes, as is the case for Hashed Loops. Thus, GRACE would be expected to dominate Hashed Loops for very large relations. The GRACE algorithm has three main weaknesses: random I/O, poor memory usage, and skew vulnerability. The issue of random I/O is difficult to separate from the particular implementation, and thus only the actual implementation in the paper will be discussed here. In it, each of the buckets is maintained in a sequential file, and since they are created concurrently during the partitioning phase (and all on the same disk), the I/O is necessarily based on random seeking between the contiguous extents of each bucket file. Fortunately, once the R and S-buckets have been created, the final join phase for each pair of buckets can use sequential I/O.

The GRACE algorithm does not exploit memory well. Since we are assuming a main memory that is large enough to hold a substantial portion of the R-relation, and since our goal is to partition R into segments small enough to allow each partition of R to be held in memory, there is no reason to create vast numbers of partitions. However, during the partitioning phase, memory is used only for input and output buffers, and since they will be relatively few, due to the low number of partitions, much of the memory goes unused during the partitioning phase. The Hybrid algorithm, described below, better exploits the available memory during the partitioning phase.

The GRACE algorithm is parallelized essentially as the Hashed Loops algorithm is. All processors work on the hashing into buckets during the partitioning phase, and all processors work on the current partition with a single global hash table during the

Hashed Loops phase.

Hybrid The Hybrid algorithm is a modification of the GRACE algorithm which is intended to eliminate one of the problems of GRACE, namely that much of the memory goes unexploited during the partitioning phase. At the same time, the algorithm reduces the number of R- and S-tuples that are written to disk as temporary results. The Hybrid algorithm differs from the GRACE algorithm by the addition of one extra hash table bucket (R_0), which is kept in memory. The R-relation is processed first, and a hash table (entirely separate from the hash function used for the partitioning into buckets) is constructed for the R-tuples which land in the R_0-bucket. The other R-tuples are processed exactly as for the GRACE algorithm. Thus, R-tuples hashing to bucket zero are never written out to disk, and the I/O cost of the algorithm is correspondingly reduced. When the S-tuples are processed, Hybrid again proceeds like the GRACE algorithm, except that tuples hashing to bucket zero are used to probe the R_0 hash table, and if a match is found, an output tuple is written immediately. S-tuples hashing to other buckets are written to memory as in the GRACE algorithm. Thus, the tuples hashing to bucket zero are never written out as temporary results. The R-tuples hashing to bucket zero are retained in memory while those hashing to other buckets are written out. Similarly, the S-tuples hashing to bucket zero are used to probe the in-memory hash table of R-tuples immediately, rather than being written out as temporary results. After the partitioning phase, Hybrid proceeds exactly like the GRACE algorithm, albeit with fewer tuples, since those hashing to the in-memory bucket have already been processed. Hybrid has essentially the same benefits and drawbacks as GRACE does. The two algorithms can be parallelized in the same way.

Parameter Varied	Memory Size	Number of Processors
Number of Processors	50% of R	1-8
Memory Size	10%, 25%, 50%, 100% of R	6

Figure 1.1: Experiments in Martin et al. paper

Modeling

In their paper, Martin et al. presented a modeling technique, and used it to model each of their three algorithms. Since they also implemented all their algorithms, they could use the actual execution times to see how close the models came to reality.

The authors were concerned with representing the concurrency of I/O and processor-time. Thus, the costs for the model were broken down into I/O and processor costs for each algorithm phase. The algorithms were divided into phases, as indicated by the points at which full barrier synchronizations were performed. For each phase, I/O and processor operation were assumed to be fully overlapped, and thus the cost of a phase was the maximum of its I/O and processor costs.

The models prepared by the authors worked reasonably well. When compared to implementation times, the estimates of the models tracked the trends in the performance closely. In no case was the estimate off by more than 25%. The models made no allowance for data skew, but performance shifts due to skew were moderate in any case.

Results

The authors' results included the three algorithms' performance, with and without I/O costs, and with either the amount of memory or the number of processors varied. The parameter values in the experiments were as shown in Figure 1.1.

In each case, the relations had a tuple size of 128 bytes, with 4-byte integer keys. In all cases, the joins were one-to-one or one-to-many; i.e. in no cases did several R-tuples join with several S-tuples.

The Processing Time These results were gathered without any I/O being performed. The R- and S-relations, the intermediate data, and the results were stored in memory. In these cases, both the R- and S-relations each had 10,240 tuples. The processor time estimates of the models closely followed the actual timings, and were consistently somewhat pessimistic. None of the algorithms were significantly affected by data skew. Basically, the authors found that Hashed Loops had the lowest processing costs; the other two algorithms had higher costs, mainly because they moved tuples about in memory. Hashed Loops and GRACE had almost perfect speedups as the number of processors increased. Because of high contention on a lock guarding the in-memory bucket, Hybrid's performance did not change significantly with the number of processors. The execution time decreased as expected for Hashed Loops with increases in memory size. The execution time for GRACE did not change at all, and execution times increased for Hybrid as the amount of main memory was increased. This deterioration in performance, which was predicted by the analytical model, was due to high contention for the single lock guarding the R_0 bucket in memory. The authors discussed the contention problem, but neither implemented nor analyzed any of the proposed solutions.

The Total Response Time These results were gathered with actual disk I/O; in this case, the R relation had 76,800 tuples, and the S relation had 10,240 tuples. Because only four disks were available, relations were not declustered across disks. In all cases, one disk contained R, one contained S, one contained any intermediate results, and one was reserved for the final results.

Because the configuration caused all the algorithms to be I/O bound, performance was essentially identical regardless of the number of processors. This fact was correctly predicted by the analytical model. Both Hashed Loops and Hybrid were able to exploit additional memory; response times dropped significantly. GRACE, predictably, did nothing to exploit additional memory. In all cases, the analytical model predicted trends, and came quite close to predicting actual performance.

Conclusions

The final verdict on the three algorithms was as follows: in an I/O bound system, Hybrid had the best response times for small memory sizes (less than 20% of the size of R); Hashed Loops had the best response times for moderate memory sizes (20-80% of the size of R); Hybrid and Hashed Loops had similar response times for large amounts of main memory. For processor-bound systems, Hashed Loops was the best algorithm for the given relation sizes. This superiority was mainly due to the fact that the Hashed Loops algorithm did not copy tuples within memory, except from the input buffer to the output buffer.

1.2 Thesis Overview

The focus of this thesis is the Domain-Partitioned Sort-Merge Join (DPSMJ) algorithm. It is a parallelization of the classic sort-merge equi-join algorithm, suitable for shared-memory environments. In chapter 2, the algorithm itself is described. The algorithm has three phases: Sort, Calculate, and Join. In the Sort Phase, the unordered relations are each arranged into sorted runs. In the Calculate Phase, the domain of join-attribute values is divided into distinct ranges. Each range constitutes one partition of the join. In the Join Phase, each partition is processed by a separate processor. All three phases are described in detail in chapter 2. The discussion goes beyond merely what each phase is supposed to accomplish, to include issues that are likely to arise in any implementation of the algorithm. In such cases, the discussion includes the range of options considered, and the selection made by the author in the implementation used in the experiments. The Sort Phase is a particular focus of attention, and two distinct versions of the Sort Phase are presented: one is based on replacement selection using a heap, and the second is based on in-memory sorting.

Chapter 3 presents a performance modelling study of the two versions of the DPSMJ algorithm. Appropriate settings of various internal parameters for performance of DPSMJ, such as buffer pool sizes and over-partitioning ratios are also explored. The study includes the I/O and processor-requirements of the Join Phase and the two versions of Sort Phase. The Calculation Phase is not studied; it is too brief to be significant in the performance evaluation. Explicit performance models are presented for the processor costs of the heap-based run forming, and the Join Phase. Data is provided to allow implicit interpolation of the other processor costs. I/O costs are modelled by explicit modelling of the number of blocks to be read in, and interpolation to derive the disk-based I/O speeds. Finally, the integration of the component models into a comprehensive performance model is discussed.

Chapter 4 describes five series of experiments performed to ascertain the performance of the algorithm in various environments. Three of the series use real I/O, and two have I/O simulated in memory for speed. Thus, both I/O-bound and processor-bound cases are explored using the actual implementation of the DPSMJ algorithm, in each of its versions. The relative performance of the heap-based and in-memory run-forming methods is studied using the experimental results. In-memory sorting is found to be far faster than heap-based run-forming. The performance of the two versions of the algorithm is also compared against the three hash-based join algorithms, GRACE, Hybrid, and Hashed Loops, whose performance is estimated using the published models by Martin et al. [13]. Both versions of DPSMJ are found to be inferior to the hash-based algorithm. Both the high I/O use and the high processor-cost of sorting hamper the DPSMJ algorithm. Finally, the models from Chapter 3 are evaluated by comparing their predictions against actual implementation results.

Chapter 5 examines three issues relating to the DPSMJ algorithm. First, the buffer use of the algorithm is studied using formal models, and compared to the buffer use by the GRACE algorithm. DPSMJ fares poorly, since its buffer use is related to the size of the sum of the two relations being joined, rather than only to the size of the smaller relation. It is an additional liability that its buffer use increases with the number of processors. Both of these factors reduce the range of utility of the algorithm.

Second, the applicability of the DPSMJ algorithm to joins using indices is studied. A means is found of breaking down what Cheng et al. term a Hybrid Join [4], which exploits an index, into two DPSMJ operations for parallelism. A way of partitioning a B+-tree [5] index for DPSMJ without reading the leaf nodes is also devised.

Third, the DPSMJ and hash-based join algorithms are compared in the case where pre-sorted relations are available. In this case, the DPSMJ algorithm is found to do well, beating all of the hash-based algorithms when the system is I/O bound, and when relations are relatively large. The performance of DPSMJ with pre-hashed relations is also compared against GRACE with pre-hased relations. Here, GRACE is consistently superior.

The conclusions of the thesis are presented in Chapter 6.

Chapter 2

Domain-Partitioned Parallel Sort-Merge Join

This chapter describes the central algorithm of this thesis, Domain-Partitioned Sort-Merge Join (DPSMJ). The algorithm performs single-attribute equi-joins of two relations. The relations are in general larger than memory, and are assumed to be unordered and without associated indices. The algorithm is parallel, and is expressed in terms of a shared memory programming model.

2.1 Classic Sort-Merge Join

The classic sort-merge algorithm [14] for uniprocessors proceeds in two phases toward the goal of joining two relations which are larger than memory. In the Sort Phase, the two relations to be joined are sorted on the joining attribute. In the Join Phase, the classic sort-merge join algorithm simultaneously scans down both sorted relations, looking for matches on the join attribute. Only a single pass over the relations is necessary in the Join Phase, because the relations have been put into sorted order.

Since the cases of interest with join algorithms involve relations that are larger than memory, the Sort Phase is generally an external sort, such as Mergesort. Here, the relations are first processed into sorted runs which are as long as the available memory will permit. The usual algorithm for creating such runs is heap-based replacement selection [10], which generates runs that are on average twice as long as a memory load. Another alternative is to read in memory-sized runs of tuples, sort them, and then write them out as the sorted runs.

However the runs are produced, they are then successively merged to arrive at a single sorted run. Given the size of modern memories, there will typically be only a relatively small number of runs. This merging requires one or two buffers per run, a relatively small amount of memory. This means that it is possible to merge the runs in one step to form the final sorted sequence. Also, in the case of the sort-merge join, the final (and only, typically) merge of the external sort can be combined with the Join Phase. The final sorted relations can be created on the fly by merging the sorted runs in the Joining Phase. In this way, the final sorted versions of the relations to be joined are never actually formed; they appear only virtually as pointers are moved in memory to derive the results of the merge, as they are requested by the actual join routines. This saves both I/O and processing time, since one cycle of relation writes to and reads from disk is avoided, and the virtual merge can take place without tuple moves in memory. (Instead, tuple pointers are advanced along each of the runs.)

2.2 Overview of the Parallel Version

The new Domain-Partitioned Parallel Sort-Merge Join algorithm is a parallel version of the classic sort-merge algorithm, designed for shared-memory use. The Sort Phase and Join Phase are both parallelized, and a new phase, called the Calculation Phase, is performed between the two major phases(as indicated in figure 2.1).

Sequential	Parallel
Sort	Sort
	Calculate
Join	Join

Figure 2.1: Phases of Sort-Merge Join

2.2.1 Phases of Processing

Two distinct versions of the Sort Phase are considered in this work. The first is based on replacement-selection, with all processors accessing a single common heap. The heap data structure is augmented to permit concurrent access and modification, and other data structures exist to minimize the amount of tuple-copying, and to collect information needed for the subsequent phases. The second approach to run-forming is based on in-memory sorting. Pseudocode for it is shown in figure 2.2. Tuples are read into memory until it is full and then they are sorted. The implementation uses an index of the tuples, which is sorted in parallel, and a parallel permutation algorithm, which places the actual tuples in the correct order with a minimal number of tuple movements. The permutation algorithm is based on decomposing the permutation, as specified by the sorted index, into cycles. The in-memory run-forming module also includes a facility for overlapping input and output processing, although neither is overlapped with the actual sorting of the tuples. The two run-forming modules produce compatible output; the Join and Calculation Phases are identical, regardless of which of the two front-ends is used.

The work to be done in the Join Phase is divided into separate partitions, each of which can be processed independently. This partitioning is performed in the Calculation Phase, based on the statistics gathered in the Sort Phase. The partitions are generated by dividing the domain of tuple values (of the join attribute) into a set of non-overlapping ranges. Each partition then consists of the tuples from each relation with join attributes in that range. Since the ranges are non-overlapping, each

```
for each relation
  read in initial memory load
  loop
    derive sorted order
    permute tuples in memory
    exit when relation exhausted
    write old memory load as a run; read in new memory load
  end loop
  write final memory load as a run
end for
```

Figure 2.2: Pseudocode for In-Memory Run-Forming

partition can be processed independently. The Calculation Phase involves choosing partitioning values from the domain of the join attribute values, and determining which blocks of each run belong to each partition. After the Calculation Phase is finished, the Join Phase can begin. Here, all processors are used, and each processes one or more partitions. The work to be performed for each partition is essentially the same merging of runs and matching of tuples performed by the single processsor in the sequential algorithm, but for only a part of each of the sorted runs.

2.2.2 Secondary Information Collected

Conceptually, the input to the run-forming step is a set of tuples, and the corresponding output is a set of runs of tuples, with each run sorted in ascending order on the join attribute. In practice, the work is somewhat more complicated. First, there is the issue of how the output runs are to be placed on disk. Also, secondary information is collected during the run-forming step for use later in the algorithm.

This secondary information describes the placement of the runs on disk. It is used in later phases of the program to partition the work of performing the actual join, and to locate particular blocks when they are needed. This information takes up relatively little space, and is therefore retained in memory. Three types of data are gathered: the run counts, BlockIndex, and RunInfo. All three types are gathered separately for each relation.

Each of the two run counts is the number of runs generated from the two relations. RunInfo is collected in the form of an array. The array has an entry for every run generated from the relation, with the following fields:

- the number of tuples included in the run,
- the number of blocks required to hold the run, and
- the block number of the first of these blocks.

Since all the blocks of a particular run are stored contiguously on single disks, the last field suffices to locate all the blocks of a run.

BlockIndex is also an array; it has one entry for every block output to disk. The entry records the value of the join attribute of the last actual tuple in the block. Note that since the runs are monotonic non-decreasing, the last tuple has the highest join attribute value in the block. These samples are important for the partitioning in the Calculation Phase.

2.3 Heap-Based Run-Forming Phase

The first version of the Sort Phase uses a parallel version of heap-based replacement selection. The work is parallelized by having multiple RunWorker processes (one for every processor) accessing a single global heap. Locks are used to maintain heap coherence during concurrent access. The module includes its own input and output routines. The structure of the heap-based run-forming module is shown in figure 2.3.

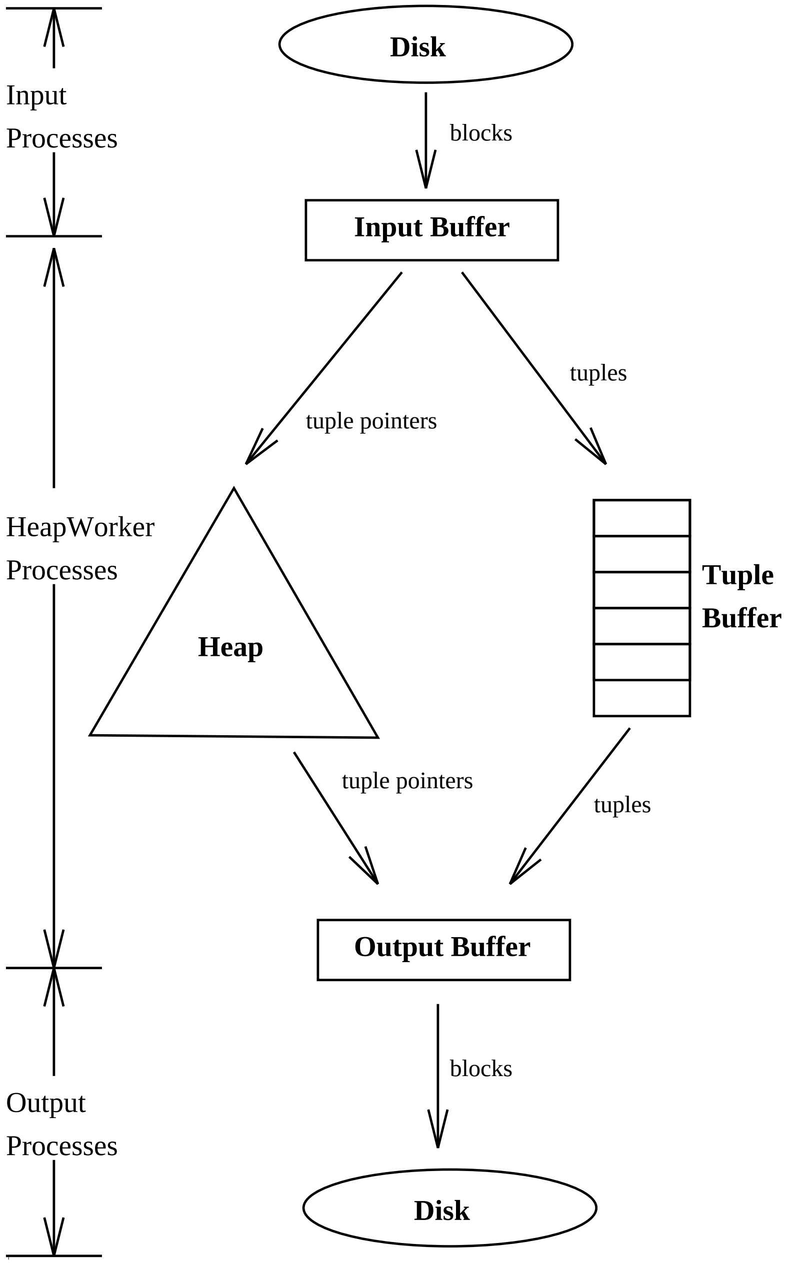


Figure 2.3: Information Flow in Heap-Based Run-Forming.

The unsorted input tuples arrive in blocks from the input routines, read from the disk by distinct InputReader processes. Each such block is received by some RunWorker, and the tuples therein individually entered into the heap. The actual tuples are not included in the heap itself. Instead, the tuples are copied into a tuple buffer, where they reside until their removal from the heap. Only pointers to the actual tuples (in the tuple buffer) are stored in the heap. The presence of a few extra tuple buffer entries allows the insertion into and deletion from the heap of tuples to take place with only two tuple copying operations required for each tuple. There is also a set of output routines, which collects the stream of tuples from the heap into blocks to be written to temporary files, from which the runs can eventually be accessed in the Join Phase. The writing of blocks of tuples to disk is done by OutputWriter processes. Of course, the heap-based run forming module also collects information to be used in the subsequent Calculation Phase, giving sample tuple values from all the blocks, as well as detailed structural information about the runs themselves and their division into blocks. (See section 2.2.2.)

2.3.1 Use of the Heap

Run forming by replacement selection is discussed at length by Knuth [10]; only a brief discussion of it is presented here. In the heap used for replacement selection, the heap elements have two fields: the run number, and the tuple value. When comparing elements, the comparison is primarily on the run number, and secondarily on the tuple value. The heap is initially filled with tuples, which are marked as being of the first run. Then, for each entering tuple, the lowest element in the heap is taken out of the heap, and the new tuple inserted in its place. The removed tuple is either written to the end of the current run, or it starts a new one. The great advantage of heap-based replacement selection is that the runs produced are on average twice the size of the heap.

Heaps are usually presented as implementations of the abstract data type Priority

Queue, and therefore support two operations: Insert and DeleteMin [1]. The heap as used for this algorithm is slightly different. It supports only one operation: SubstituteMin. This operation retrieves from the heap the lowest element, and replaces it in the heap with a given element. Thus, the heap never changes in size. As part of each SubstituteMin operation, the new element's position in the heap is adjusted by a series of element-interchanges to retain the heap property.

In the algorithm, the heap is never emptied. Instead, placeholder values are used which are less than or greater than any possible input. Thus, to simulate an empty initial heap, each heap entry is initialized with MININT, an element that is lower in value than any actual entering element. To empty the heap in the final stage of processing, elements are inserted which are higher in value than any actual element(MAXINT).

Thus, all phases of the run-formation can be performed using the same SubstituteMin operation. During the first phase, the RunWorkers carry out SubstituteMin, removing the top element and adding an element from the input. The element removed is a MININT, indicating that the heap is not yet full, and that no output needs to be performed yet. The second phase flows seamlessly from the first. With the heap full and containing no MININT elements, the element removed from the top of the heap is a reference to a real tuple, which must be output. Once all the input is exhausted, the processors begin emptying the heap in the third phase, by entering MAXINT elements. [1]

[1]Note that it is necessary for the RunWorker processes to synchronize using a barrier before beginning the third phase. This synchronization is necessary since it would be problematic to have some processors, without any remaining input, beginning to empty the heap while others still have input to insert into the heap.

2.3.2 The Concurrent Heap

The first step to performing the SubstituteMin operation is the removal of the root of the heap, and its replacement by the new element. Then, the inserted element is bubbled down the heap to its appropriate position. Only those elements on the path from the eventual destination to the root are modified, and in addition to this set, only the children of these nodes are accessed. The advantage of using SubstituteMin this way is that all the operations flow down the heap in the same direction, downward. The conventional Insert and DeleteMin operations for the heap flow in opposite directions in the heap, leading to deadlock problems in concurrent implementations. Rao and Kumar give a method for parallelizing the conventional insert and delete operations [16], but the version described here, using SubstituteMin, solves the problem more directly.

The comparison predicate evaluated for each bubbling operation is relatively complicated, since it potentially requires the comparison of four different values. Each node contains two elements: the run number and the element value. Comparison is primarily on the run number, and secondarily on the element value; i.e., the element value is consulted only in the event that run values are equal. The expense of the comparison predicate turns out to be significant. The metric is complicated, and so the cost of the decision predicate to be evaluated at each level of the heap is high. This is one of the major drawbacks of heap-based run-forming.

Locking in the Heap

The heap data structure incorporates one lock for every node within it. The heap adjustment algorithm, shown in basic form with locking in figure 2.4, neither accesses nor modifies the value of a node without first holding its lock. Note that the maximum number of locks within the heap held at any one time by a single RunWorker is three: a node, and its two descendants, which are held when comparing the element's value to those descendants. Since the adjustment process always holds on to at

```
set cur_element to heap root
lock cur_element
while cur_element has descendants do
  lock the descendants i and j of cur_element
  select the descendant with the lower value (say i)
  unlock j
  if ( cur_element <= i)
     unlock cur_element, i
     break loop (finished)
  else
     swap cur_element value with i value
     unlock cur_element
     set cur_element to i
  end if
end while
```

Figure 2.4: Pseudocode of Heap Adjustment, with Locking

least one lock in its traversal down the tree from the root to the inserted element's eventual destination, it is impossible for the heap's structure to be disrupted by having modifications interfering with one another. The locking protocol ensures that the heap operations are exactly serialized. The heap behaves exactly as if the parallel modifications had been made sequentially in the order in which the root node's lock was seized by the RunWorkers.

Other implementation techniques were tried. First, the per-element locking method described above was replaced by per-level locking. It was hoped that this would speed up the heap-adjustment process, since it required only one lock to be taken at each level, rather than the two required for the per-element locking. No significant advan-

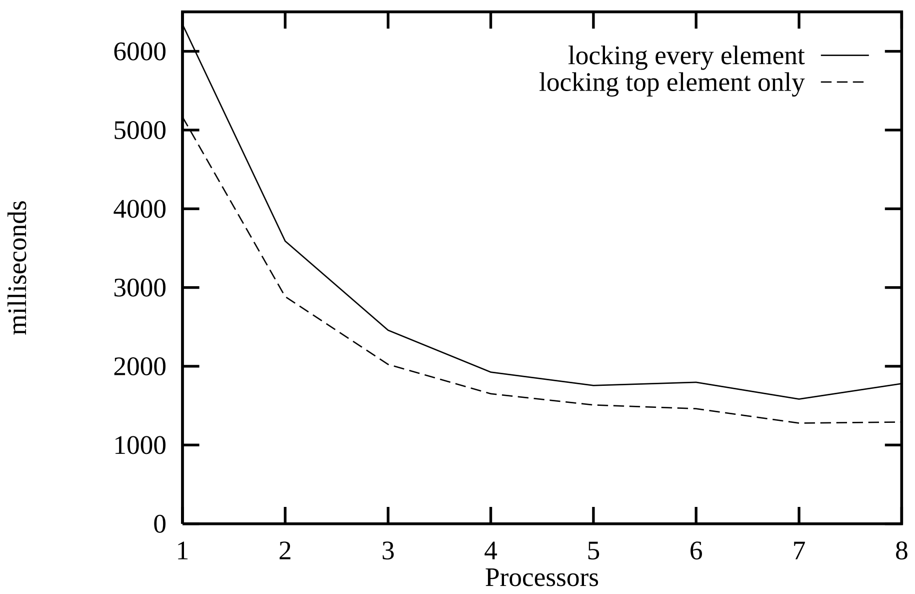


Figure 2.5: Processing Time (10,000 tuples, heap height 16)

tage was found in speed, and the per-element locking technique was retained. The per-element protocol was tried with both real queuing semaphores, and with faster spin locks. The spin locks were found to be somewhat faster in operation, and the order-preserving property of the queueing semaphores is not needed for the adjustment algorithm.

A number of experiments were carried out to assess the impact of locking and unlocking in the heap-adjustment process. Results and discussion are included in appendix D. The experiments were conducted with a proper heap, and with input and output tuple copying simulated by performing random copying in an array of tuples. The most useful results compared processing using a fully consistent heap, maintained using one lock for every element, with processing using an inconsistent heap, where only the top of the heap had a lock. The difference in processing times was a reasonable indication of the impact on processing time of locking in the heap. Results for a representative case appear in figure 2.5. Generally, processing times declined 15-30% when the heap was allowed to be inconsistent. Therefore, locking during heap adjustment is responsible for some of the (high) costs of heap-processing, but by no means all of them. The inherent complexity of heap adjustment is part of the problem, as are the two full tuple moves needed for every processed tuple.

The Critical Section

When a RunWorker is removing the head element from the heap, and inserting in its place another element, the RunWorker necessarily has the heap head element locked and unavailable to other processes. This time, before the RunWorker begins bubbling the top element down the heap in the heap adjustment process, is the start of the Critical Section. Since the operation can only be undertaken when the top of the heap is held, and since there is only one heap, the critical section is a strictly sequential phase in an otherwise parallel stage of processing. Since the sequential order is determined by the order in which the top of the heap is seized, the Critical Section is the time when a number of other operations related to tuple source and destination addresses are performed. Given the sequential nature, and all the processing to be done within it, the Critical Section could potentially strictly limit the possible parallelism. In its current configuration, after careful optimization, the so-called Critical Section does not turn out to be a bottleneck, since tuple copying and heap adjustment occur outside of it. This fact was determined during development by keeping track of how long RunWorker processes waited for access to the top of the heap; the waiting time was not significant.

2.3.3 Use of the Tuple Buffer

In the naive implementation, without a tuple buffer, each heap element would hold a full tuple, necessitating full tuple copy operations for every heap element exchange. Since such tuple copy operations are relatively expensive (76 microseconds for 128-byte tuples,) reducing their number is imperative. This is the purpose of the tuple buffer. The tuple buffer eliminates the need for tuple copying during the heap adjustment process. Each processed tuple is copied twice within the program: once from the input buffer into the tuple buffer, and a second time from the tuple buffer into the appropriate output buffer. By using the tuple buffer, the heap elements include only pointers to the associated tuples rather than the tuples themselves, allowing much faster element-interchange operations, and hence a much faster heap adjustment process.

The tuple buffer holds one tuple-sized slot for every element in the heap plus one slot for each RunWorker process. This overall distribution remains fixed during all heap operations although the ownership of actual slots changes continually. In each cycle, to process the next input tuple, the RunWorker first copies the tuple into its slot in the tuple buffer. Since the RunWorker's slot in the tuple buffer is *not* referenced in the heap, this operation involves none of the elements of the heap, and thus no synchronization is required before it can be performed. The root of the heap does not need to be locked until *after* the tuple has been copied into the tuple buffer. Then, when the top element of the heap is to be removed, and the new tuple is to be inserted in its place, what is entered into the heap is a pointer to the new tuple in the tuple buffer, and what is removed is also a pointer to a tuple in the tuple buffer, the old tuple in this case. Thus, the RunWorker gives up one slot in exchange for another. The tuple to be copied out is in the newly-gained tuple buffer slot. After this tuple is copied to the output buffer, the new slot becomes the RunWorker's slot for the next cycle. It is significant that both the tuple copy into the tuple buffer and the tuple copy out of the tuple buffer take place outside of the heap adjustment process, without having any portion of the heap locked, since in each case, the slot of the tuple buffer copied into or out of is not referenced in the heap at the time the tuple copy takes place.

Options and Issues

Four issues were considered when evaluating design options for the tuple buffer:

(1). minimizing copying during heap adjustment,

(2). minimizing other copying,

(3). minimizing the number of slots, and

(4). minimizing waiting for slots.

The above design satisfies criterion (1) perfectly; no RunWorker ever performs any tuple copying while holding any locks within the heap. It also perfectly satisfies criterion (4), since the swapping of slots at heap insert/delete time means that no RunWorker ever has to wait for a slot. The performance with respect to criterion (3) is also quite good. The minimum for the number of slots needed is the size of the heap, since every element in the heap must have a space for its tuple. The chosen method only exceeds this lower bound by the number of processors. Since the heap size is at least thousands, while the number of processors (i.e. the number of RunWorkers) in shared memory multiprocessors is at most dozens, the number of extra slots is insignificant. This relatively small excess in the size of the tuple buffer is what enables the RunWorker processes always to avoid waiting for a tuple buffer slot, a great benefit.

One problem with the current algorithm is that every tuple has to be copied twice (once, into the tuple buffer from the input buffer, and a second time, from the tuple buffer to the output buffer.) Thus, the design performs only decently with respect to criterion (2.) Reducing the copying would be problematic, since the input order and the output order of the tuples can be very different. Eliminating one of the copying operations would require tuples to be moved directly from the input buffer to the output buffer with no intermediate copying. To do this would require a large set of input buffers, with tuples kept in the input buffers until their destination output buffers were known. This would be inefficient, since the buffers would on average be only half full, and could not be released until they emptied completely. Also, in the worst case, each buffer might contain only a single tuple, resulting in an unacceptably small heap. This method was considered but rejected due to its inefficient memory use and complexity.

2.4 In-Memory Run-Forming Phase

The second variety of run-forming module considered in this thesis is in-memory run-forming. Using this technique, the in-memory sorting module uses all available memory to read in a sequence of blocks of tuples of the relation being processed. The tuples, once brought into memory, are sorted into ascending order by join-field value and are written to disk as a run of the relation. The runs into which the relation is thus divided are equal in length to the memory available. In the course of the sorting, the same sampling data is collected as is gathered by the heap-based run-forming module (see section 2.3.)

2.4.1 Three Components

The in-memory sorting module is composed of three submodules, which execute in turn to process each memory load:

- Sort,
- Permute, and
- Flush.

The in-memory run-forming method needs to sort the tuples in memory. This is a complicated operation because the sort has to be done in parallel, and because the tuples are very large. This makes conventional sorting algorithms, which move elements about multiple times, impractical. Instead, the in-memory module uses an index of the actual tuples, which is sorted in parallel. The algorithm used is Parallel Sorting By Overpartitioning[11]. This sorting algorithm achieves good parallel performance and speedup, and is not further discussed here. What is of interest is what is done with the index once it has been sorted. The problem then is to permute the actual tuples into the order specified by the index, and to do so using as few tuple moves as possible. Of course, it is also desirable that this operation be done in parallel. This is done by the Permute module.

Of course, sorting the tuples in memory is only part of the problem. The blocks of tuples also have to be moved from the original relation file into memory, and from there into the temporary file. The Flush module uses multiple processes to write out the blocks in memory, and to simultaneously read in fresh input as these blocks are made available. As tuples are read in, Flush also fills in the index with the tuples' keys. Of course, the first and last flushing operations involve no writing and no reading, respectively. During processing, control alternates between the sorting and I/O components.

Processes Used

Each of the three modules of in-memory run-forming has its own processes, which are dormant when the other two modules are executing. The Flush module has FlushWorker processes that read in and write out blocks of tuples. The Sort module has PQOP (i.e., Parallel Quicksort by Over-Partitioning) processes that operate to derive the sorted order of the tuples. The Permutation module has Permute processes that put the array of tuples into the derived order.

2.4.2 Permutation Module

The Permute module works with the Sort module to produce a sorted run of tuples in memory. The Sort module presents to the Permute module the unsorted array of tuples in memory, and an index specifying their correct order. It is the task of the Permute module to put the tuples in the order specified by the index, and to do so with as few tuple movements as possible, since tuples are large, making their movement expensive. The Permute module, using all available processors, performs this task in parallel. The algorithm basically proceeds by considering the index as specifying a permutation, and decomposing this permutation into cycles. Each cycle of length i can then be processed with $i+1$ tuple moves, using the technique of cycle shifting.

Cycle Shifting

To begin with, the index specifies a permutation of the elements of the array. In particular, the index specifies for each index position i, the current index j of the tuple that should be at array index i. For example, in figure 2.6, position 3 of the index ($i = 3$) has value 1 ($j = 1$), indicating that the tuple currently in position 1 in the array should be moved to position 3. In general, any permutation on N elements can be represented as a directed graph on N elements with exactly one out-edge and one in-edge per element. Figure 2.6 shows a relation, before and after sorting. It also shows the corresponding index, and its representation as a graph. In this example, we adopt the convention that an edge pointing $A \rightarrow B$ indicates that the element at position A should be moved to position B.

The way to shift tuples to their destinations at minimal cost is based on the decomposition of the graph into its cycles. To begin with, trivial cycles, such as the one at position 2 in the example, require no action; they indicate correctly positioned elements. The other elements, members of non-trivial cycles, are permuted by shifting the elements of each cycle forward one step in the direction of the edges. This places each element of the cycle in its correct position, based on the above definition of edges, with one move per element. The processing of each cycle requires one additional move: the element in the position that is the target of the first move has to be stored to a temporary location, from which it is moved to its final location at the end of the

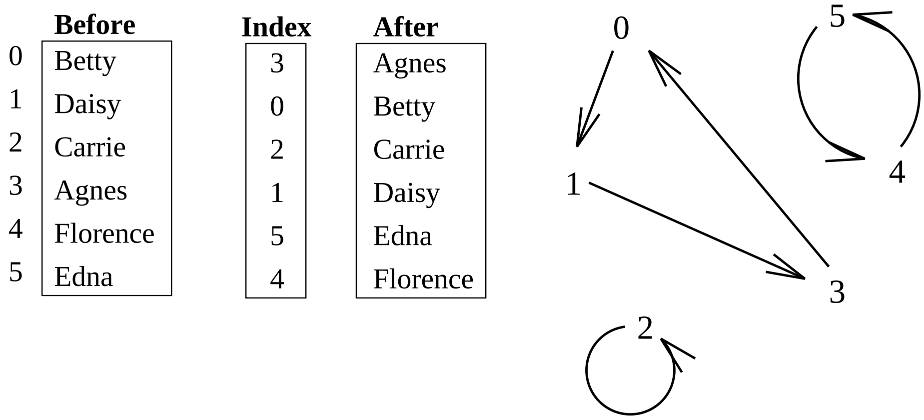


Figure 2.6: Relation, with corresponding index and its graph representation.

cycle's processing.

Of course, each cycle has to be found before it can be shifted in this manner. To find a cycle, we begin at an element which is not part of a known cycle, and step through the cycle of which it is a part, traversing the edges backwards, until we return to the original element. Note that graph edges are most easily followed backward. For example, in figure 2.6, the edge is from 1 to 3, and the entry at offset 3 of the index is 1. This requires only an array-lookup. In contrast, following an edge forward, such as the edge from 3 to 0, requires searching the entire index, an expensive operation. It is for this reason that cycles are traversed backwards. As a cycle is found, the elements comprising it are marked as members of a known cycle.

The algorithm uses several simple data structures to find and process the cycles, the first of which is the index, as discussed above. The permutation algorithm adds to this a boolean *busy* for each array index and a lock for every array index. The algorithm also maintains a *total_found* counter, specifying the number of entries put into their proper places, and a lock guarding the counter for concurrent access. None of these data structures is particularly large. The total overhead per tuple is four integer-sized fields:

- the key value,

- the index entry,
- the *busy* boolean, and
- the lock.

This comes to 16 bytes, which is small compared to the size of typical tuples. The overhead can be reduced to 12 bytes by reusing the key value field, which is only needed during the actual sorting of the tuples. Of course, if the implementer were willing to rewrite the concurrency primitives, the lock and the boolean could be collapsed to single bits, and merged with the index entry into one 32-bit integer. This would leave 30 bits for the index entry, an ample amount for most circumstances.

For the actual processing, the algorithm uses one Permute process per available processor to perform its task in parallel. The pseudo-code for the Permute processes is shown in figure 2.7. The full code for the Permute processes appears in appendix B.

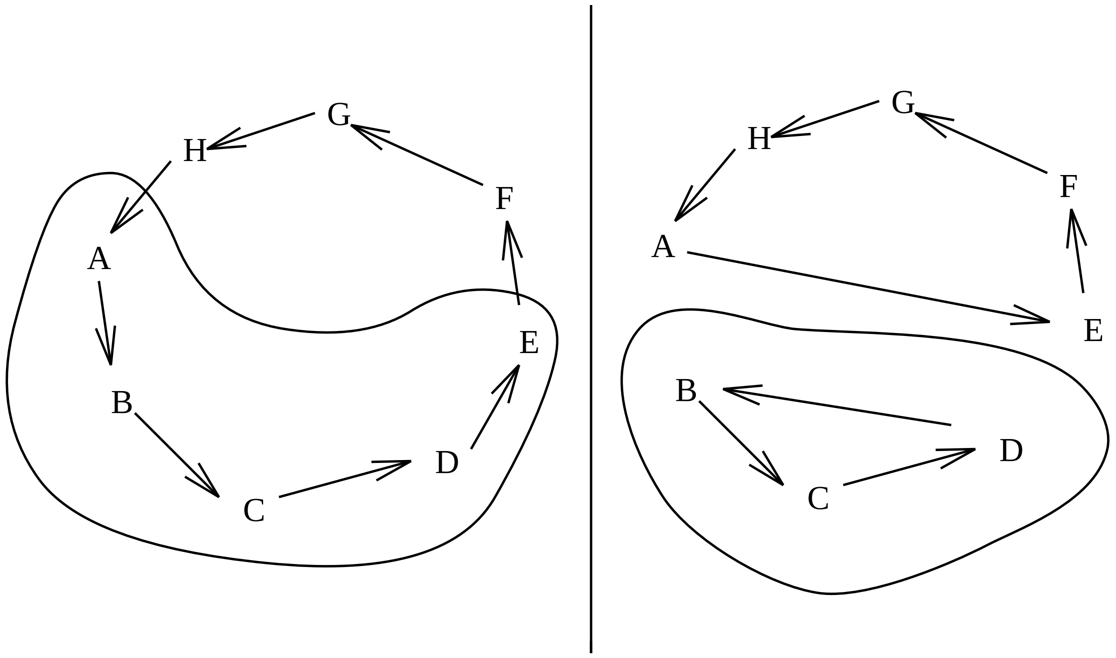
Cycle Clipping

What the Permute process does, in the case where it cannot claim an entire cycle, is to rearrange the elements of the array and the index so as to link together the first and last elements claimed, and to link the other claimed elements into a separate cycle. The two elements, now linked together in the original cycle, are abandoned (de-claimed by setting their busy entries to false), and the new cycle detached from them can be shifted on its own. Conceptually, this requires the redirection of two edges in the graph. In practice, it requires the swapping of two index entries and two tuples within the array.

Consider the graph in figure 2.8. The initial state is at left. Here, the Permute process began at entry E, and has claimed the elements up to and including A, but H is already claimed by some other Permuter. The desired state is at right. Here, the permuter has redirected two edges (A to E, rather than A to B; D to B, rather than D to E) and abandoned two entries to the original cycle (A and E.) It has formed a

```
begin with current positions for all Permuters distributed throughout
      the array
while not all elements correctly positioned
  step through the index, looking for start_index, not part of an
       existing cycle
  starting at start_index, claim entries, following the cycle backwards
       until arriving at start_index or at a locked entry
  if not found a full cycle
     if length < 3
        abandon all claimed elements
     else
        clip cycle
  if have a cycle
     increment number of elements found by length of cycle
     shift cycle
end while
```

Figure 2.7: Pseudo-code for Permute.



Data

Location	Value
A	Bob
B	Carl
C	Dennis
D	Edward
E	Felix
F	George
G	Harold
H	Aaron

Index Array

Destination	Source
A	H
B	A
C	B
D	C
E	D
F	E
G	F
H	G

Figure 2.8: Cycle Clipping.

cycle of its own (B to C to D, and back,) which it can now shift, since it has already claimed all its elements. This cycle clipping requires the switching of elements A and D in the array, and the switching of elements B and E in the index, as shown in the figure. From this formulation, it is clear that cycle clipping should not be done when a Permute process has claimed fewer than three elements. In such cases, the Permuter abandons all its elements, and continues searching through the array for another cycle start.

Cycle clipping is done for two reasons. First, it is a means of resolving conflicting claims to a cycle. It will often happen that two or more Permute processes will begin claiming elements of a cycle, hence preventing any of them from claiming the entire cycle. The simplest way to resolve this would be to have the processes abandon the claimed elements when a conflict is found, but this would lead to poor load balancing. The second reason for clipping cycles is precisely this load-balancing issue. If no clipping were done, the permutation workload would divide naturally into the cycles in the original permutation. The work represented by each cycle is roughly proportional to the number of elements in it. Unfortunately, permutations generally have a few very large cycles, which account for most of the elements. In a series of experiments with 25 600-element permutations, the largest cycle was found to account for on average 63% of the elements, and the second largest accounted for 19%. This type of workload distribution would produce poor results for any conceivable scheduling algorithm, and would lead the very poor parallel speedups. Hence, cycle clipping must be performed.

2.4.3 Flush Module

The Flush module is a component of the in-memory run generation module. Its purpose is to write out the blocks of tuples currently in memory, while simultaneously reading in the next sequence of blocks as the buffers become free. The Flush module operates in conjunction with the Sort and Permute modules of the in-memory run generation module. It is invoked to read in an initial sequence of blocks to be sorted, and is re-invoked until the entire relation to be sorted into runs has been read in. It is then re-invoked one final time to write out the final sorted run, without reading in anything from the (now fully read) input file.

Note that the version of in-memory run-formation described here does not overlap the sorting of the tuples with I/O processing. This is because all the memory is used to form a single run. If the memory were divided into two buffers, one could be sorted while the other was being flushed, permitting I/O to overlap with sorting. Of course, this would produce approximately twice as many runs as the single-buffer version, and four times as many runs as the heap-based run forming method. This was felt to be excessive, particularly since the algorithm is sensitive to the number of runs produced, particularly as the number of processors increases.

2.5 Calculation Phase

The Calculation Phase has two tasks to accomplish. First, it must select the partitioning to be used to divide the work of the join phase among the available processors. It does so by selecting a set of partitioning values, which together divide the set of join-attribute values into a set of non-overlapping ranges. Each partition, which can then be processed independently by a single processor, consists of all the tuples from each relation with join-attribute values in the range of the partition. Second, the Calculation phase must determine, for every partition, which blocks in every one of the already-generated sorted runs might have tuples which belong to that partition's range. When that partition is processed, these are the blocks that must be read in from disk and searched for tuples that are to be part of the join. Figure 2.9 illustrates the partitioning. The Calculation Phase performs these two tasks, the partitioning and the block assignment, using the secondary data generated at run-forming time. The Calculation Phase involves little data movement and no I/O. It therefore takes an

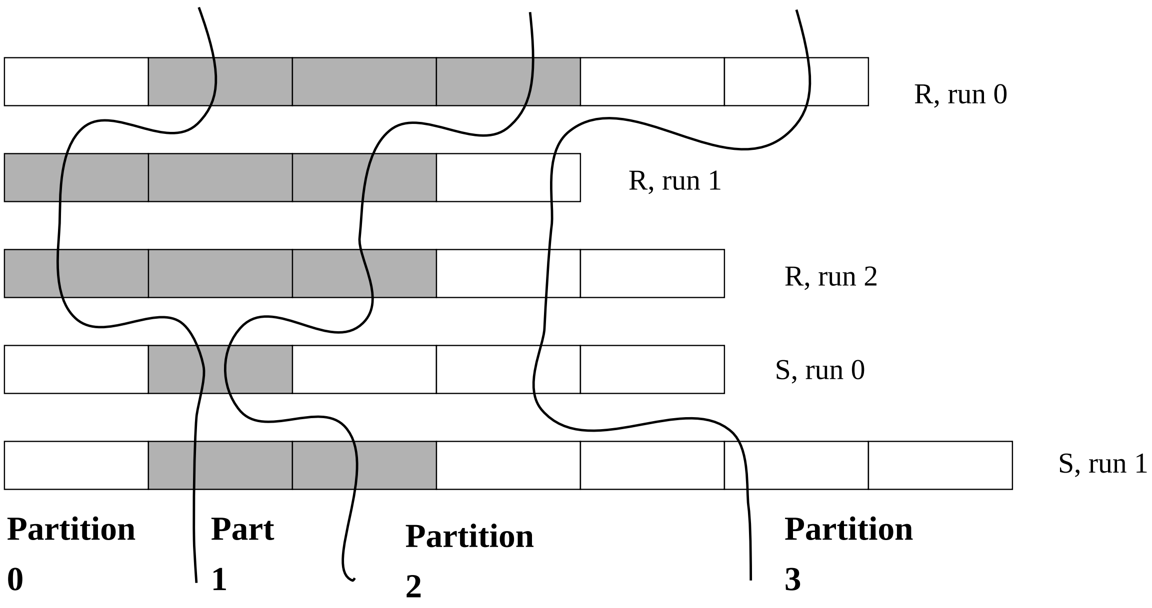


Figure 2.9: Partitioning.

insignificant amount of time, and is not parallelized. There are no distinct processes started up for the Calculation Phase.

2.5.1 Selecting Partition Values

For reasons of load-balancing during the join phase, the Calculation phase generates twice as many partitions as there are processors. [2] First, to derive the $2p$ partitioning ranges, the program selects $2p-1$ distinct partitioning elements $\{e_1, \ldots, e_{2p-1}\}$ These partitioning elements define the $2p$ partitions $P_0, \ldots, P_{2p-1}$ as follows. A tuple x, with join attribute $f(x)$ is part of

P_0 if $f(x) \leq e_1$,

P_{2p-1} if $f(x) > e_{2p-1}$, or

P_i if $e_i < f(x) \leq e_{i+1}$.

Thus, since the partitioning elements are distinct, each tuple x belongs to exactly one partition P_j for some j. It would be desirable to have each of these partitions

[2] The number of partitions is discussed in section 2.6.

containing the same number of tuples. This would lead to good load-balancing. The program attempts to bring this about by carefully selecting the partition values.

The partition values are selected using the secondary information gathered during run-formation. (See section 2.2.2.) Recall that for every block of every run, BlockIndex records the join-attribute value of the last tuple within that block. The collection of these values, for each of the two relations, can be considered a statistical sampling of the join-attribute values of the tuples of each relation. During the calculation phase, these two collections of values are merged into one and sorted. The algorithm then selects the $2p - 1$ partition values from this list such that they approximately equipartition the list. Thus, it is hoped, the partition elements will equipartition the full set of tuples in each relation, by equipartitioning the list of samples.

It is possible that repetitions occur within the sample values. This is problematic since the definition of the set of partition value becomes invalid if any of the partition values are repeated. To resolve this problem, the algorithm, having found a partition value, advances the sort to the next non-equal element. This eliminates the possibility that the next partition value will be equal to the current one. The rest of the list of samples is then equipartitioned into the remaining number of segments.

Mergesorting the samples

It should be noted that the set of samples is never explicitly sorted, with the equipartitioning elements then being selected from this list. Instead, the structure of the samples in the BlockIndex table is exploited to permit a much faster derivation of the sorted order. Essentially, given that the blocks on each disk appear as runs, and the fact that the tuples in each run are organized in ascending order by join-attribute value, the samples corresponding to each run are known to be in ascending order by join-attribute value. This fact can be used to mergesort each of the runs of samples. Of course, there is no need to actually produce in memory a sorted list of tuples. Since the number of samples is known, the program can precompute the indexes in the list that would be used to select the partitioning elements. Thus, we do not actually need to do anything with the other elements. The program can just let the mergesort keep working (moving pointers along the runs of samples) until it produces the element at the required index. Only the chosen partition elements themselves actually need to be kept.

2.5.2 Computing Partition Blocks

The calculation of the partition values that define the partitions is the first duty of the Calculation Phase. The second duty is to use this set of partitions to derive the set of blocks which have to be searched for tuples that are members of each partition. This task is performed at the run level; for every run, the algorithm calculates the set of blocks associated with each of the partitions. The set of blocks associated with each partition is then the union over the runs of the blocks associated with the partition. For example, in figure 2.9, the blocks that contain elements of partition 1 are shaded.

Given a run r of a relation R, the run is known to consist of a certain number of tuples n_r, and a certain number of blocks B_r. Both of these values are recorded in the RunInfo data structure, as discussed in section 2.2.2. The tuples in the run are known to be organized in non-decreasing order by join-attribute. The run r exists on disk as a collection of B_r contiguous blocks, $b_1, \ldots, b_{B_r}$, the ids of which are recorded in the BlockIndex data structure. Associated with this collection of blocks, the program has a run of samples $s_1, \ldots, s_{B_r}$, where s_i is the join attribute of the last tuple in the block b_i. Given the ordering of the tuples in the run, the samples are in non-decreasing order. The task is to generate, for each partition P_i, the associated set of tuples that might include tuples in the range of the partition. Given the structure of the runs, it turns out that this set of blocks will actually be a sub-range of the blocks $b_1, \ldots, b_{B_r}$.

To calculate the ranges of blocks associated with a particular partitioning $P_0, \ldots, P_{2p-1}$ (defined by partition elements $e_1, \ldots, e_{2p-1}$), the algorithm derives a set $c_1, \ldots, c_{2p-1}$

of blocks with each e_i associated with the corresponding c_i. These blocks are termed the *partition blocks* of the run. By definition, c_i is the first block (in the series $b_1, \ldots, b_{B_r}$) with a sample s_i which is greater than e_i, or b_{B_r} if no such block exists. For each partition P_i defined by e_i and e_{i+1}, the associated range of blocks is $c_i, \ldots, c_{i+1}$. The edge cases are defined analogously: with partition P_0, we associate $b_0, \ldots, c_1$, and with partition P_{2p-1} we associate $c_{2p-1}, \ldots, b_{P_r}$. Note that every partition block is a part of (at least) two partitions, since the ranges of blocks are inclusive.

Properties of the partitioning

The ranges of blocks generated for the partitions have some properties that bear commenting on. Ideally, a run will be long, and the partition values well distributed within it. This case will produce a distinct partition block for every partition value. These partition blocks will belong to both of the adjacent partitions; all other blocks will belong to only one partition. Thus, when I/O is performed, in the ideal case, every block in the run will be read once, and the partition blocks will be read a second time. This is, of course, the ideal case, reached only when there are enough blocks, and the partition elements are sufficiently well distributed among them, to produce a unique partition block for every partition value. In more problematic situations, it is possible for several successive partition elements to be located in the same partition block. The above definitions are not contradicted by this fact, and given the inclusive nature of the block-ranges, the join algorithms process them normally.

2.6 Join Phase

The join module performs two tasks at once. It merges the sorted runs composing each of the two source relations into the final sorted sequence for each relation, and it joins these two relations by performing what may be described as a virtual sort-merge join. These two activities take place simultaneously. During the Join Phase, each of the partitions generated in the Calculation Phase is independently processed by a Joiner process. Each processor runs a Joiner process of its own. The Joiner reads in the appropriate sequences of blocks from each sorted run and merges them into the final sorted sequence of tuples for each of the two relations. These two sorted sequences are then joined using sort-merge join, and the results are written to disk independently for each partition. The actual reading of blocks is handled by the TemporaryReader processes, and the result blocks are written out by OutputWriter processes.

For scheduling purposes, the partitions are sorted in descending order by estimated workload. Joiner processes then request partitions to process in this order, to improve load-balance. Since partition sizes vary, the number of partitions into which the relations are divided is twice the number of processors, again to improve load balance. A range of other partitioning values were tested experimentally during development, but an overpartitioning factor of two was found to be the best compromise between the improved load balancing possible with more partitions, and the extra cost involved in starting each partition.

Joiners request partitions to process by accessing a shared concurrency-controlled data structure. The estimate is the total number of blocks required to be read in to process the partition in question. The partitions are given to requesting Joiners in this order to improve load balance across the processors.

The output from the join phase is the set of joining tuples; for each such joining pair, both of the tuples are output together. The output will necessarily be in sorted order by join-field value. This conceptual output stream, resulting from the join of the two entire sorted relations, is divided by the partitioning process. Each partition involves processing an independent segment of the two joining relations, resulting in an independent segment of the join output. The output blocks of each partition are inter-mingled with the blocks containing results from all the other partitions. The

I/O routines precisely record where each block of each partition is placed on disk. Thus, although the output does not appear sorted on disk, it is sorted within each block, and the proper sorted order for the blocks is known. Of course, not all the output blocks are full, since the final block of output from each partition is typically not full.

2.6.1 Reading and Merging the Runs

The routines accessed by the Joiner are responsible for reading in the blocks of the current partition from each of the runs. To do this, each Joiner allocates two buffers for each run of each relation. These two buffers are used to read in the blocks of that particular run. Note that the number of buffers needed by this algorithm is large, being proportional to the number of processors (i.e., Joiners) and to the number of runs. Hence the need exists to keep the run count small. The two buffers per run per Joiner correspond to double buffering. Allocating more is of little benefit; fewer would hamper performance significantly. This issue is addressed further in section 5.1.3.

The final sorted order of tuples for each relation is produced by the Joiner by merging the tuples within the buffers. This merging is done without any tuple movement. Instead, pointers are advanced along each of the runs when their blocks are brought into memory. This is implemented by having a pointer in the Joiner to the next tuple in each of the runs. These pointers are organized into a heap by the join-attribute value of the tuple pointed to; the next tuple in the join is the one pointed to by the tuple at the top of the heap. Thus, the routines present to the actual join routine a single virtual pointer which can be advanced along the tuples of the partition. In practice, this pointer alternates among the runs comprising the relation.

2.6.2 The Actual Merge-Join

The current version of the joining routine works only for 1-1 joins, i.e. joins in which any tuple in one relation corresponds to at most one tuple in the second relation.

Minor alterations of the routine would allow it to perform N–1 or 1–N joins, i.e. joins whereby a tuple of one of the relations can correspond to more than one tuple in the other. The code would have to be substantially revised to accommodate N-N joins, in which it is possible for a group of tuples in one relation to correspond to a group of tuples in the other relation, with the join being the Cartesian product of the two groups.

The joining routine begins by initializing two pointers (one per relation) to the first tuple of the current partition in each relation. If the tuples have the same values, the two tuples are part of the join, and they are written to the output. Then both pointers are advanced one step along their respective relations. If, on the other hand, the join fields are not equal, then the pointer indicating the tuple with the lower value is advanced until the tuple pointed to has a join attribute value equal to or greater than the join attribute value of the other tuple. Then, the tuples pointed to can be compared again. In this fashion, the two pointers, working on separate relations, leapfrog each other until they both arrive at tuples with equal join fields. Pseudocode for this process is shown in figure 2.10.

This process take place, as mentioned before, without any tuple movements. Everything is done using pointers, which advance along the tuples located in the Joiners' input buffers. Tuple movement takes place only when an actual joining pair is found, and both tuples are copied into the output buffers.

To modify this routine for N–1 or 1–N joins, the only change required is that only one of the pointers be advanced after a pair of joining tuples has been found. This allows the routine, having found a tuple in one relation which joins to another in the second relation, to search for more tuples which join to the original tuple. Modifying the routine to permit N-N joins would be more difficult. Essentially, given a tuple a_j in relation A, which joins with tuples $\mathcal{B} = \{b_1, b_2, \ldots, b_n\}$ in relation B, the $\mathcal{B}$ set of tuples would have to be stored somehow, since they might also join with a_{j+1}. Since $\mathcal{B}$ could potentially be all of B, a complicated mechanism including temporary

```
set cur_1 and cur_2 to first tuples of relations 1 and 2
while neither relation is exhausted
      if cur_1.join_attribute = cur_2.join_attribute then
         output cur_1, cur_2
         advance cur_1 and cur_2 one tuple
      else if cur_1.join_attribute < cur_2.join_attribute then
         advance cur_1 until cur_1.join_attribute >= cur_2.join_attribute
      else if cur_1.join_attribute > cur_2.join_attribute then
         advance cur_2 until cur_2.join_attribute >= cur_1.join_attribute
end while
```

Figure 2.10: Pseudocode for Joining.

storage to disk would have to be devised to permit N–N joins with full generality.

Chapter 3

Performance Study

This chapter discusses the performance of the various components of the implementation. Both the processor costs and the I/O costs are considered. The performance is explored under a variety of conditions and parameter-settings. Models of the performance are constructed and validated for some of the more important modules. Section 3.4 addresses the integration of the models for the individual modules into a model for the entire Domain-Partitioned Sort-Merge Join algorithm.

3.1 Heap Based Run-Forming

During the heap based run-forming (see section 2.3), the processor cost is incurred largely in cycling tuples through the concurrent heap, and copying the tuples between buffers. The I/O routines read the original relation sequentially from disk, and write the blocks of the runs sequentially to disk. Both the source and the destination can be one or more disks.

3.1.1 Processor Time

Two types of tuple processing occur during heap based run-forming. The first type of processing, called *filling*, occurs when the heap is full, and tuples are continually being entered and removed. The second type of processing, called *emptying*, occurs when the supply of input tuples is exhausted, and the heap has to be flushed of the remaining tuples within it. The filling and emptying operations differ in two respects. First, filling is more work, since a tuple has to be copied into the tuple buffer before the element is entered in the heap. During emptying, this copying into the heap does not take place. Second, the filling operations have a large granularity of parallelism, consisting of entire blocks of tuples as they are received from the I/O routines. In contrast, emptying has a small granularity of parallelism, consisting of single tuples. This small granularity of parallelism makes good load balancing possible, and not surprisingly, emptying parallelizes well, whereas filling parallelizes quite poorly.

Filling Costs

Figure 3.1 shows the average time taken per tuple when filling. The times are shown over a range of processor counts (P) and a range of heap heights (H). Note that this operation parallelizes rather poorly; for H=12, the parallel efficiency [1] at P=8 is only 42%.

A number of models were fitted to this data, but the best was

$fill(P, H) = -.00727 * P * H + .0290 * H + .173 * W + .0279$

The constants for this model were arrived at by least-squares analysis in Maple [3] using $35 = 5 * 7$ points; the included points had $7 \leq H \leq 11$ and $1 \leq P \leq 7$. An examination of the timings in the edge cases showed the costs to be erratic and therefore unreliable.

Figure 3.2 shows the ratios of the model predictions to the data from figure 3.1. Except at the edges, the maximum relative error of this prediction function is 4%. The error is larger at the edges, but this is acceptable since the true performance graphs behave badly at the edges (as the heap gets very large, and as the number of

[1] The definition of *parallel efficiency* as used in the chapter is consistent with the accepted definition in the field: $E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)}$.

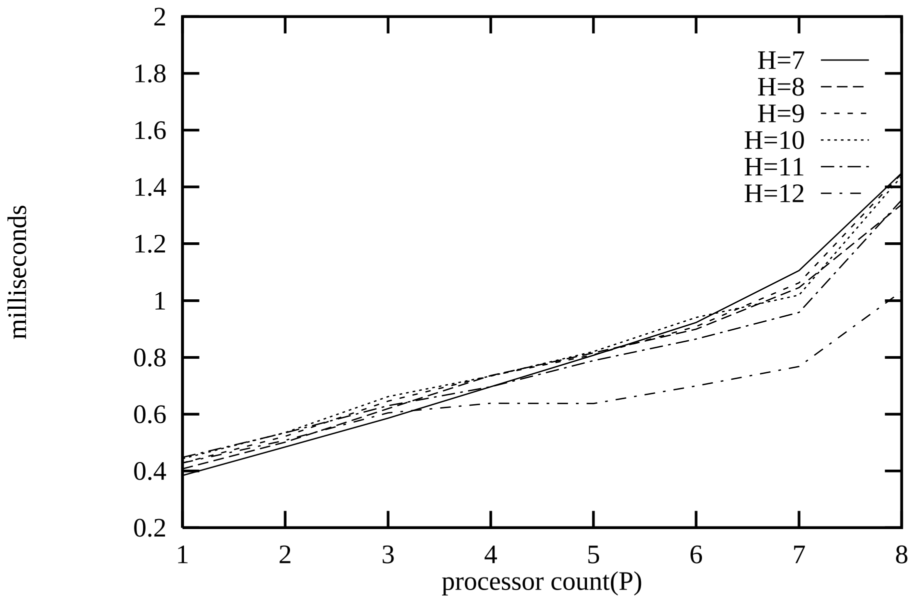


Figure 3.1: Average Filling Cost Per Tuple Per Processor

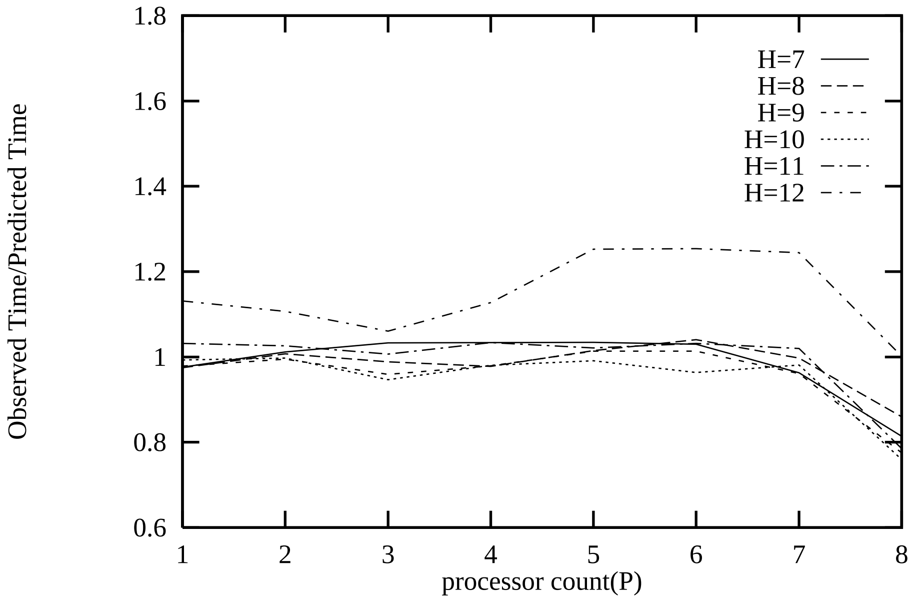


Figure 3.2: Prediction of *fill* Relative to Average Filling Cost

processors approaches the maximum for the machine).

Emptying Cost

Figure 3.3 shows the average emptying costs per tuple per processor. Notice, again, that the operation parallelizes quite poorly. The parallel efficiency for H=12 and P=8 is only 32%.

A number of models for this data were constructed. The one which best models the average emptying cost per tuple is

$empty(P, H) = -.00172 * P * H + .0308 * H + .113 * P - .0515$

The constants for this model were arrived at by least-squares analysis using 30 points; the included points had $7 \leq H \leq 12$ and $1 \leq P \leq 5$. An analysis of graphs of the true costs for the excluded range edge cases showed the costs there to be erratic and therefore unreliable.

Figure 3.4 shows the accuracy of the *empty* model. Within the range for which the curve was fitted, this curve predicts the performance to within 3%. The error for the higher values of P are more severe, with a maximum error of 28% in one location. The set of points shown in figure 3.3 is actually very complex. For the low-P cases, a plane is a good approximation, since the costs rise nearly linearly in both axes. The behaviour for high-P cases is sharply different; suddenly, the cost becomes insensitive to H, and rises sharply as P varies. The following curves were fitted to the whole or part of the data:

$aP^3 + bP^2 + cP + d + e/P$

$aP^3 + bP^2 + cP + d$

$(aP + b)(cH + 1)$

$aH + bP + cH/P + d$

None of the curves fitted to the data were able to account for the H-dependence in low-P cases, while also accounting for the H-independence in high-P cases.

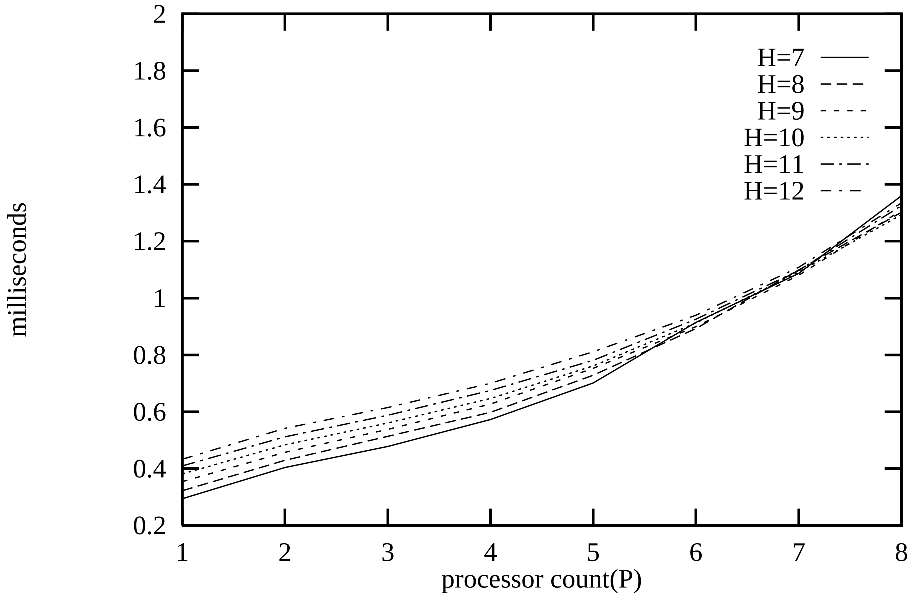


Figure 3.3: Average Emptying Cost Per Tuple Per Processor

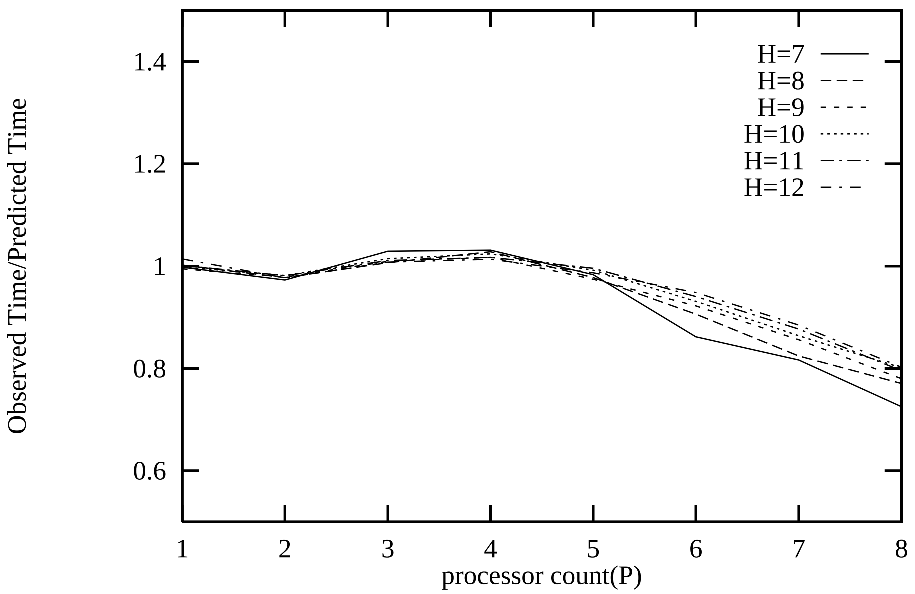


Figure 3.4: Prediction of *empty* Relative to Average Filling Cost

3.1.2 I/O requirements

The heap based run-forming module performs input and output sequentially. Figure 3.5 shows the performance of the I/O routines used. Also shown is the performance of pure stripped-down I/O routines, under the same parameters. Results are shown for varying block sizes (B, in tuples), and varying numbers of worker processes per disk. Note that the performance for the pure and actual routines was not collected in quite the same way, which accounts for the fact that the actual routines' performance sometimes exceeds the performance of the pure routines. In any case, the results establish that the secondary record-keeping performed by the actual routines does not hamper performance. For the experiments that will be described in chapter 4, I/O was performed using two readers per disk, and a block size of 256. Note that essentially full performance can be obtained for blocks as small as B=64 tuples, but performance suffers at B=32.

3.2 In-Memory Run-Forming

The in-memory run-forming module (described in section 2.4) has the interesting characteristic that I/O-time and processor-time are not only divided into separate modules, but are also separated in time. The processor-intensive code is inactive when the I/O intensive code is active, and vice versa. The processor-intensive work is performed in the Sort and Permute modules, and the I/O in the Flush module. Thus, the modeling considers only processor time for the sorting and permutation; and only I/O costs for the Flush module. Appendix A discusses the processor costs of flushing, which are minor.

	Reading		Writing	
	Pure	Actual	Pure	Actual
1 worker				
B=256	0.940	0.940	0.941	0.947
B=128	0.626	0.626	0.626	0.631
B=64	0.360	0.375	0.375	0.378
B=32	0.208	0.208	0.208	0.210
2 workers				
B=256	1.88	1.89	1.88	1.90
B=128	1.87	1.88	1.87	1.89
B=64	1.83	1.84	1.80	1.80
B=32	0.385	0.383	0.380	0.385
3 workers				
B=256	1.88	1.88	1.88	1.90
B=128	1.88	1.89	1.88	1.89
B=64	1.88	1.87	1.86	1.89
B=32	1.79	1.78	1.76	1.78

Figure 3.5: Sequential I/O Routines' Performance (MB/s)

3.2.1 Processor Time

The Sorting Module

The sorting module has strange performance characteristics, as can be seen from figure 3.6 [2]. Note that performance depends strongly on both the number of processors available, and on the number of blocks in memory(M). Notice also that for P=8, performance increases as M increases, whereas for P=1, performance decreases as M increases.

The reason for this puzzling behaviour is not clear. The behaviour might be due to two effects of cache use: co-location and cache contention. Co-location refers to the fact that needed data can often be found in the cache, since it has been drawn in by over-sized cache lines, accompanying nearby data. Co-location is probably the central issue in low-P cases. As the size of the array increases and memory references spread farther apart, co-location is reduced, and performance decreases. In contrast, for high-P cases, performance *increases* as M increases. This could be accounted for by cache contention. Cache contention occurs when one processor has a cache line that is needed by another; performance suffers when this happens. As the memory size increases, the chance of conflicting memory accesses decreases, which would account for the increases in performance with increasing M in high-P cases.

Since the sorting module is not a focus of this thesis, no further results are reported here. For modeling purposes, the values of figure 3.6 can be interpolated.

[2] Note that the time shown is the execution time of the algorithm divided by the number of tuples, not the processor time per tuple. It is therefore to be expected that the times indicated will decline with increasing P, as long as parallelism is not actually counterproductive.

The Permutation Module

As can be seen from figure 3.7 [3], the performance of the permutation module is essentially independent of M, the number of blocks of tuples in the array being permuted; the time taken per tuple does not vary significantly with M. The graph shows the execution time of the Permute module per tuple for various processor counts and block counts, with blocks of 256 tuples. The performance is, however, dependent on the number of processors available. The algorithm does not manage to make full use of all the available processors; the parallel efficiency drops to about 56% when 8 processors are used, as can be seen in figure 3.8. The efficiency results in figure 3.8 are based on the M=320 line in figure 3.7.

The values from figure 3.8 can be used directly for modeling the execution time of the Permute module, with interpolation as necessary.

3.2.2 I/O Requirements

I/O during in-memory run-forming is handled exclusively by the Flush module, which overlaps the writing out of one run with the reading in of the next sequence of blocks of tuples to be sorted. The Flush module is described in section 2.4.3.

The performance of the Flush module is related to four parameters: P, the number of processors; W, the number of worker processes; B, the number of tuples per block; and M, the number of blocks in memory. Of these, only W is completely internal to the Flush module, with no direct impact on the rest of the program. There is some liberty in choosing values for B and M, although Flush cannot choose these values, since their values affect the performance of later stages of processing. However, since they can be specified by the program as a whole, the program can take into account the extent to which the Flush module's performance is dependent upon B and W when selecting appropriate values. Given the number of parameters, and the time

[3] As in figure 3.6, this is the processing time per tuple, not the processing time per tuple *per processor*.

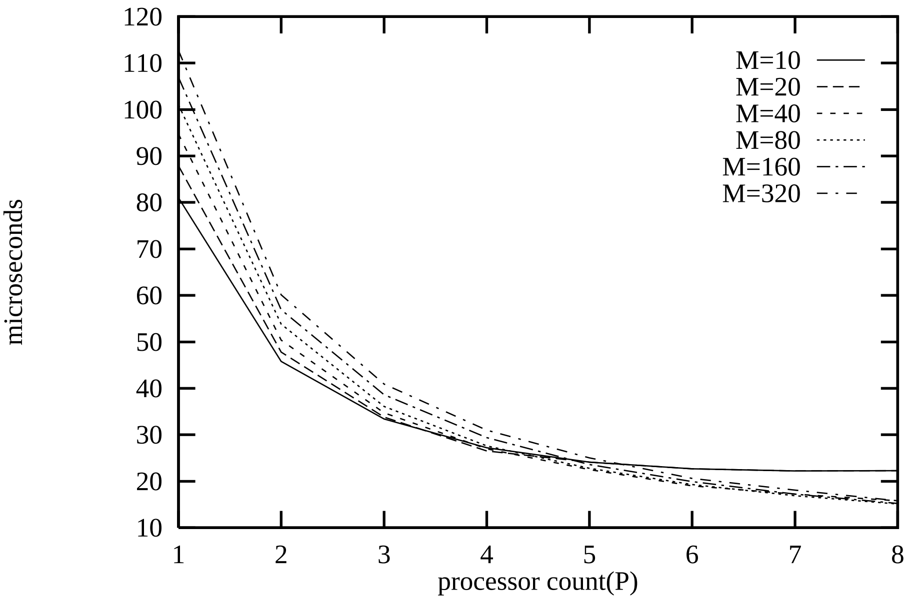


Figure 3.6: Execution Time for the Sort Module Per Tuple

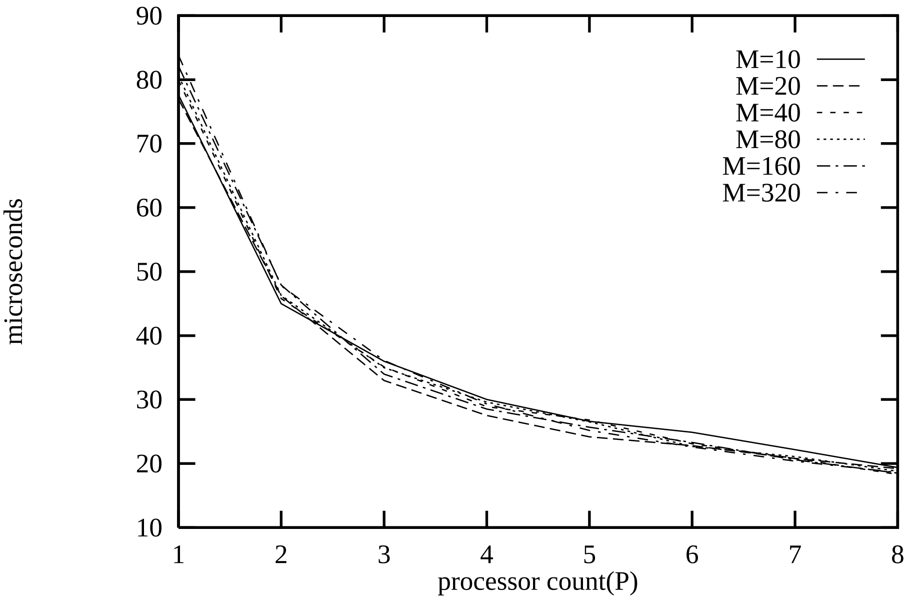


Figure 3.7: Execution Time for the Permute Module Per Tuple

P	Efficiency(%)
1	100
2	87.7
3	77.4
4	71.5
5	66.5
6	61.7
7	58.7
8	56.3

Figure 3.8: Permutation Efficiency (P=256, M=320)

involved in performing experiments, the results of this section are necessarily based on a limited search of the performance-space.

Selecting W

The first problem is selecting an appropriate value of W. This value might be dependent on any or all of the other parameters, but turns out to be dependent only on B. Figure 3.9 establishes that the desired W is independent of P; W is optimal at 4 in each case. Figure 3.10 establishes that the desired W is independent of M; W is again optimal at 4 in each case. The desired value of W is dependent on B. The results of a series of experiments to determine the best W-values are given in figure 3.11.

Evaluating Performance

Figure 3.12 establishes that performance is essentially independent of P. This is particularly true for higher M-values, which are the general domain of use for the algorithm. Consequently, performance can be usefully evaluated without reference to the P parameter.

It is useful to distinguish among three types of Flush cycles: read, read/write, and

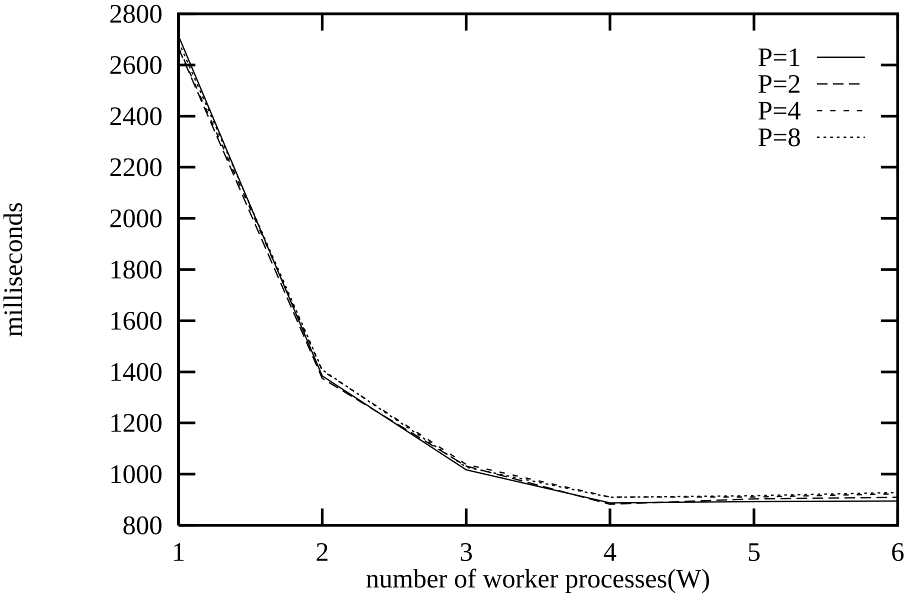


Figure 3.9: Time Per Flush (B=256, M=50, size=4M)

write. The first use of the Flush module is always a read cycle, since there is no data to write out; similarly, the last use of the Flush module is always a write cycle, since all the data have already been read in. All the intervening uses of the module are read/write cycles. Figure 3.13 shows the performance of the Flush module under a range of B and M values. A few trends are evident from this graph. First, performance generally increases as the memory size processed increases. Second, performance

M	W=1	W=2	W=3	W=3	W=5	W=6	W=7	W=8	W=9	W=10
100	0.686	1.241	1.721	1.818	1.813	1.812	1.811	1.801	1.801	1.805
200	0.747	1.475	1.677	1.737	1.841	1.831	1.833	1.843	1.838	1.840
300	0.744	1.479	1.733	1.856	1.851	1.855	1.852	1.850	1.849	1.851
400	0.766	1.470	1.732	1.852	1.849	1.856	1.857	1.811	1.856	1.850
500	0.753	1.479	1.745	1.863	1.869	1.867	1.865	1.866	1.863	1.863

Figure 3.10: Flush Processing Speed (B=256, P=4, relation size = 2M)

B	W
256	4
128	6
64	12
32	19

Figure 3.11: Flush Module Optimal W Values

is comparable for B=256 and B=128, but significantly lower for B=64 and B=32. Third, the Read/Write phases are the slowest type of Flush, presumably since they have to wait for both reads and writes to complete. The model of performance is the interpolation between these points, hence the extent of data included in the table.

3.3 Joining Phase

The join phase is described in section 2.6. The multiple join processors of the join phase use random I/O for input, but write the output sequentially. The processor costs during this phase are incurred merging runs and manipulating pointers when searching in tables.

3.3.1 Processor Time

The performance of the join module was tested without any I/O being performed. The test included 128 cases, with each case being run 21 times; the first run was excluded in each case due to memory effects, and the mean of the other twenty was calculated. The cases varied over parameters w, the number of tuples written; r, the number of tuples read; and P, the number of processors. The cases can be characterized as

$$\{w = 4000, 2000, 1000, 200\} * \{r = 12000, 16000, 20000, 24000\} * \{P = 1, 2, 3, 4, 5, 6, 7, 8\}$$

A total of 8 different models were tested, each formed using this set and the method of least squares. The best model was

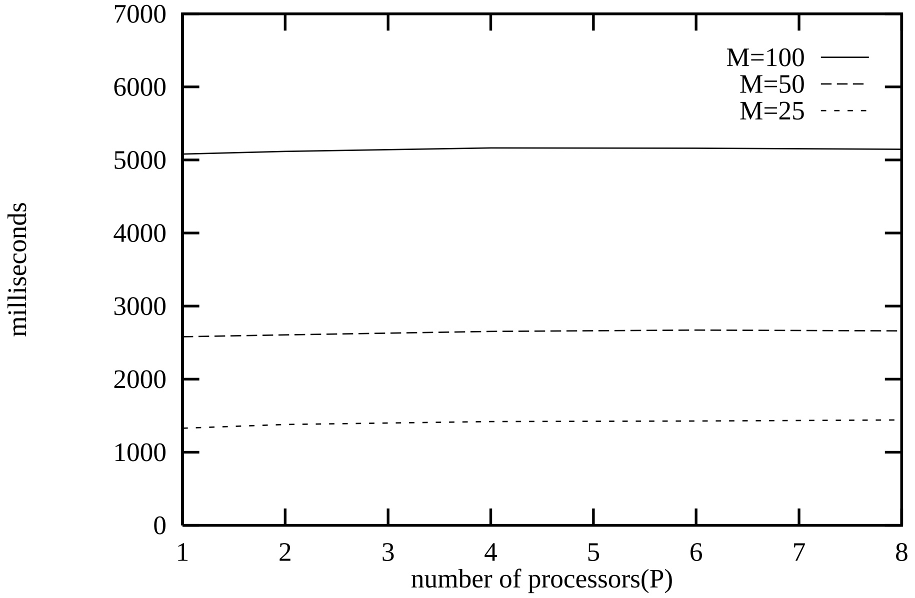


Figure 3.12: Flush Time

$A * r/P + B * w/P + C * r$

with constants

$A = .0558; B = .0818; C = .00607$

Of the model's estimates,

- 73/128 are within 5% of the actual,
- 100/128 are within 10% of the actual, and
- 117/128 are within 15% of the actual.

This is acceptable performance for a model.

The constants A and B are particularly interesting. The model gives the execution time in milliseconds, and the constants A and B represent the time to process a tuple read and a tuple write, respectively. The values of 55.8 usec and 81.8 usec for these operations are very close to the numbers derived by direct investigation of the per-tuple processing costs.

	M=20	M=40	M=60	M=80	M=100	M=200	M=300	M=400	M=500
	B=256, W=4								
R	1.669	1.768	1.792	1.827	1.834	1.854	1.865	1.872	1.874
R/W	1.548	1.688	1.733	1.765	1.786	1.825	1.827	1.836	1.849
W	1.682	1.785	1.820	1.822	1.842	1.860	1.867	1.878	1.879
	B=128, W=6								
R	1.547	1.676	1.746	1.771	1.797	1.833	1.838	1.860	1.867
R/W	1.309	1.523	1.623	1.672	1.690	1.757	1.791	1.813	1.809
W	1.498	1.676	1.743	1.773	1.799	1.839	1.856	1.852	1.860
	B=64, W=12								
R	0.978	1.347	1.425	1.656	1.733	1.791	1.812	1.841	1.842
R/W	0.935	1.143	1.285	1.513	1.601	1.673	1.698	1.733	1.748
W	1.319	1.554	1.643	1.708	1.733	1.808	1.814	1.821	1.841
	B=32, W=18								
R	0.995	1.243	1.368	1.576	1.560	1.696	1.762	1.785	1.809
R/W	0.785	1.050	1.020	1.154	1.172	1.205	1.212	1.252	1.283
W	1.028	1.403	1.499	1.594	1.647	1.692	1.790	1.812	1.827

Figure 3.13: Flush Speed(MB/s) (P=4, relation size=2M)

3.3.2 I/O Requirements

The read traffic of block requests to each disk is unfortunately random, rather than sequential. The runs follow one another on the logical disks, and the blocks of each run are contiguous. The requests are random because each join process is at any moment reading tuples from all of the runs, and can at any moment request the next block from any run. Thus, the disk ends up seeking from run to run continually.

The reading is done with two reader processes per disk. Figure 3.14 shows the reading speed achieved with various block sizes (B, in tuples), reader counts, and reading fields (N, in blocks.) The data are shown at length because of their use in interpolation modeling. Figure 3.15 displays graphically the reading speed achieved in the configuration in which the experiments to be described in chapter 4 were run, i.e., B=256 and 2 readers. Here, the system is reading blocks at random from a field of N blocks. Naturally, the performance decreases as the field increases in size, and the disk has to seek over a larger area. According to these results, no benefit would be gained from using more reader processes.

The write output traffic from the join module is sequential. It uses the same I/O routines as the heap-based run-forming module. The performance of these routines is discussed in section 3.1.

3.4 Integration of the Models

The previous sections of this chapter have discussed the performance of the various phases of the Domain-Partitioned Sort-Merge Join algorithm. They have also provided either explicit performance models, or sufficient data to interpolate performance values for the various phases and modules of processing. From these, it is possible to integrate a comprehensive model of the algorithm's performance. The model divides neatly into the Sort Phase and the Join Phase. (The Calculation Phase is too short to require modeling.) For each phase, the cost is the greater of the I/O and the

	N=1	N=2	N=5	N=10	N=50	N=100	N=500	N=1000
1 reader								
B=256	0.935	0.936	0.935	0.935	0.958	0.950	0.916	0.899
B=128	0.922	0.748	0.755	0.747	0.748	0.698	0.634	0.609
B=64	0.467	0.465	0.453	0.454	0.466	0.462	0.396	0.379
B=32	0.233	0.233	0.246	0.259	0.266	0.265	0.237	0.220
2 readers								
B=256	1.87	1.87	1.48	1.36	1.15	1.15	1.06	1.02
B=128	0.936	1.24	1.09	1.01	0.935	0.853	0.759	0.734
B=64	0.468	0.621	0.705	0.666	0.634	0.624	0.499	0.479
B=32	0.233	0.120	0.172	0.114	0.134	0.132	0.314	0.294
3 readers								
B=256	1.87	1.87	1.48	1.36	1.15	1.15	1.06	1.02
B=128	0.936	1.24	1.10	1.01	0.935	0.855	0.759	0.733
B=64	0.468	0.621	0.733	0.678	0.635	0.623	0.501	0.478
B=32	0.234	0.311	0.390	0.399	0.371	0.366	0.315	0.294

Figure 3.14: Random Reading Speeds(MB/s)

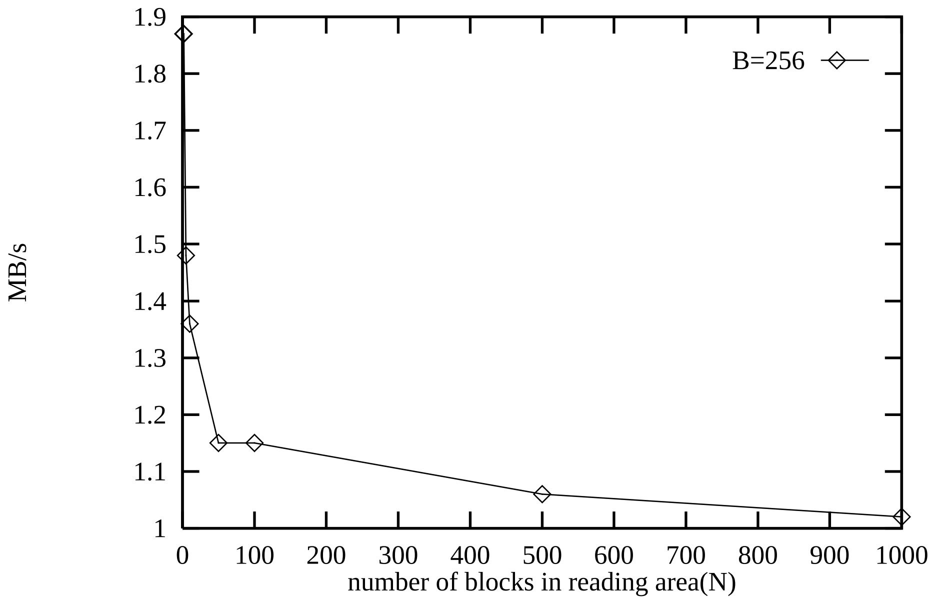


Figure 3.15: Random Reading Speed for Experiments

processor cost. Thus, the comprehensive model assumes completely overlapped I/O- and processor-times.

3.4.1 Join Phase Modeling

In the Join Phase, the time to model is either the time needed for the I/O or the processor time, whichever is greater. (In I/O-less cases only the processor time estimate is used.) The processor time for the Join Phase is modeled in section 3.3.1. A Join Phase I/O time estimate can be derived by calculating the number of blocks on disk (from the run forming) and using this with figure 3.15 to interpolate a reading speed. This speed can then be multiplied by the amount of data to be read in. The number of blocks to be read in is all the blocks, plus $2p-1$, assuming $2p$ partitions (as suggested in section 2.5.2).

3.4.2 Sort Phase Modeling

The integrated model also needs to include the Sort Phase, and one of the two possible run-forming methods of the Sort Phase. The times to process both of the joining relations must be included in the estimate. If heap-based run-forming is used, the time estimate for each relation is the maximum of the processor and I/O costs (if any). The processor cost is estimated by using the *fill* and *empty* models of section 3.1.1, which give the per-tuple filling and emptying costs respectively. These must be multiplied by the number of tuples filled into the heap (the total size of the relation) and the number of tuples emptied out of the heap (the total size of the heap). The I/O cost is the maximum of the costs to read and write the entire relation (or the runs formed, which end up being only fractionally larger.) Figure 3.5 shows the I/O speeds.

If in-memory run-forming is used, the estimation process is quite different. First, the number of rounds of processing required for each relation must be calculated by dividing the relation size by the amount of memory available. Then, estimates must be derived for each of the three processing modules for each cycle: Sort, Permute, and Flush. The estimate for the Sort Module is generated by interpolation, based on figure 3.6. Similarly, the estimate for the Permute module is generated by interpolation based on figure 3.7. Both of these figures give the per-tuple processing times; they must be multiplied by the actual number of tuples being processed. The performance of the Flush module can be estimated by interpolation of the values of figure 3.13.

Chapter 4

Experiments

Ten series of experiments were conducted using the Domain-Partitioned Sort-Merge Join algorithm, to explore the performance of the algorithm under a variety of conditions, and to determine the accuracy of the performance models in chapter 3. The experiments themselves are described in section 4.1. The performance of the algorithm is discussed in section 4.2, and the accuracy of the models is discussed in section 4.3,

4.1 Description of the Experiments

The experiments can basically be divided into two categories: real-I/O and simulated-I/O. In the real-I/O experiments, the program performs actual I/O using a small number of disks, reflecting situations where I/O is likely to be a system bottleneck. In the simulated-I/O experiments, the program performs no actual I/O. In this case, all the relations, temporary results, and the final products are stored in memory rather than on disk. These experiments simulate cases where I/O is very fast, making the processors the likely bottlenecks. For shared-memory parallel processors, this is a realistic possibility, since the number of processors is quite limited, but the number of disks can be substantial.

series	base	mult	p
ProcessorVaryRealIO	150	3	1-8
SizeVaryRealIO	50-250	3	4
RatioVaryRealIO	150	1-5	4

Figure 4.1: Experiments with Real I/O

The experiments were conducted on a Sequent Symmetry multiprocessor with ten Intel 80386 processors and 48 megabytes of memory[19]. In all of the experiments, each tuple was 128 bytes in length, including a four-byte integer key. Processing and I/O(if applicable) was performed in blocks of 256 tuples (32 K). In the experiments with real I/O, five disks were used: one for each of the relations (two, total), one for the temporary results for each of the relations (two, total), and one for the final results. Except for the memory size, this experimental platform is exactly the one used by Martin et al. for their experiments.

4.1.1 Experiments with Real I/O

The experiments with real I/O were carried out in three series (ProcessorVaryRealIO, SizeVaryRealIO, and RatioVaryRealIO) using both heap-based run-forming and in-memory run-forming. The three series of experiments for real I/O processing can be described by the three-tuple (*base*, *mult*, p). Here, *base* is the number of blocks in S, the smaller of the two relations. The parameter *mult* is the ratio of the sizes of S to the size of R, the larger relation. Finally, p is the number of processors available. The parameter values used in each series of experiments are summarized in figure 4.1. Note that the three series intersect: experiment (*base* = 150, *mult* = 3, $p = 4$) is a member of all three.

The memory size for all the real-I/O experiments was fixed at 128 blocks. For the heap-based run-forming, this memory was used for the heap's tuple buffer and for I/O buffers. For the in-memory run-forming module, the entire set of 128 blocks was

Object	tuples	blocks
R relation	38400	150
S relation	19200	75
Output	3840	30

Figure 4.2: Relations for Experiments with Simulated I/O

series	processors	memory size
ProcessorVaryNoIO	1-8	25
MemoryVaryNoIO	4	10-50

Figure 4.3: Parameters for Experiments with Simulated I/O

used for the central tuple array. The number of blocks allocated to the I/O system varied with p, ranging from 8 at p=1 to 24 at p=8, but this was a small fraction of the total number of memory blocks. Since the heap-based run-forming algorithm works faster the smaller the heap, the heap in each experiment was adjusted to be as small as possible.

4.1.2 Experiments with Simulated I/O

The experiments with simulated I/O were carried out in two series: ProcessorVaryNoIO and MemoryVaryNoIO. Both series were carried out using both heap-based run-forming and in-memory run-forming. Since the entire relations and all results had to be stored in memory, the relations and in-memory block counts were necessarily smaller than they were for the real-I/O series of experiments.

Both the ProcessorVaryNoIO and MemoryVaryNoIO series of experiments were carried out using the same relations, as per figure 4.2. Figure 4.3 shows the parameters for the experiments.

$bytes_R$	128	size of tuple of R relation, in bytes
$bytes_S$	128	size of tuple of S relation, in bytes
bs	32768	block size, in bytes
t_{probe}	20	time to probe a hash table entry, in microseconds
t_{buf}	40	time to acquire a buffer position, in microseconds
t_{hash}	20	time to hash a key, in microseconds
t_{copy}	0.6	copying time per byte, in microseconds
t_{insert}	20	time to inset an entry into a hash table, in microseconds
$READ_{seq}$	16622	time to read a block, sequentially
$WRITE_{seq}$	16622	time to write a block, sequentially
$WRITE_{rand}$	27412	time to write a block, randomly

Figure 4.4: Parameters for Martin Models

4.2 Performance of the Algorithms

4.2.1 Experiments with Real I/O

The aggregate performance of the two versions of sort-merge join, as well as the three types of hash-based joins, is shown in figures 4.5, 4.6, and 4.7. The sort-merge results are actual performance measurements; the hash-based results are estimates, based on the Martin et al. models [13]; relevant simulation parameters are as shown in figure 4.4. The three hash-based algorithms, as described in section 1.1.2, are Hashed Loops, GRACE, and Hybrid. The division of time within the algorithm can be seen in figures 4.8 and 4.9. Figure 4.8 breaks down the costs for the run forming phase into Sort, Permute and Flush, since it deals with cases where in-memory run-forming was used. Figure 4.9 breaks the costs down only into the run-forming and joining phases since it deals with cases where heap-based run-forming was used. Further proportion analysis results appear in appendix C.

At the settings investigated in the experiments, the in-memory run forming equals

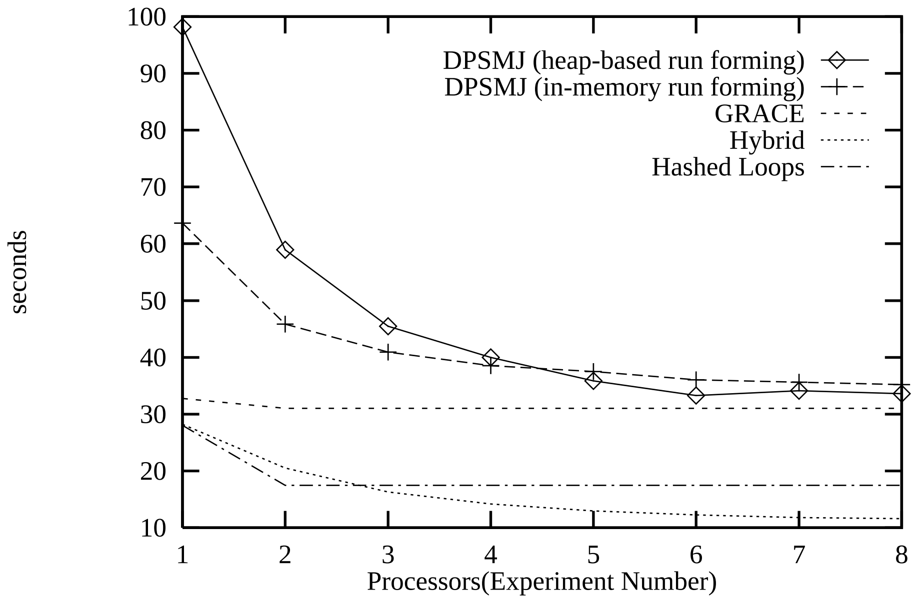


Figure 4.5: Total Time for ProcessorVaryRealIO Experiments

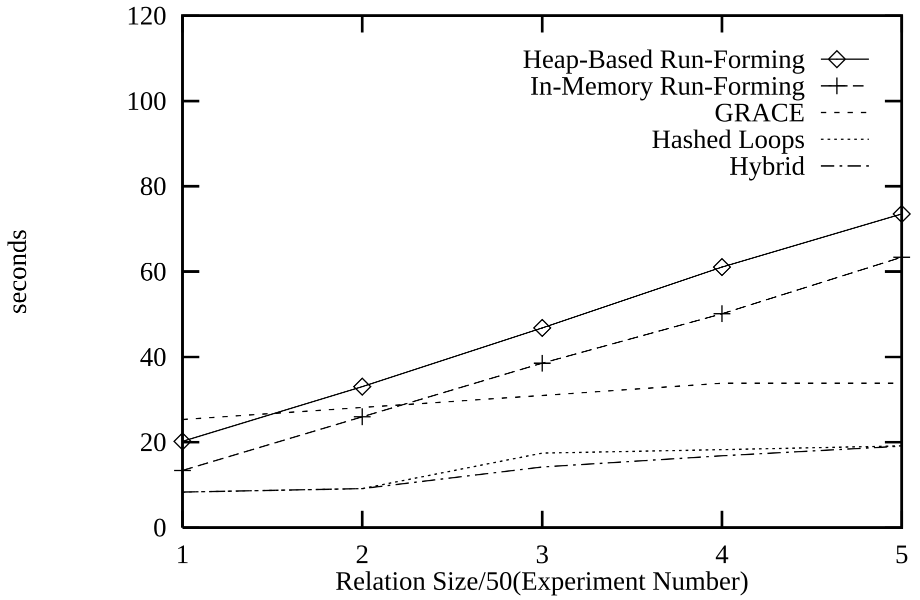


Figure 4.6: Total Time for SizeVaryRealIO Experiments

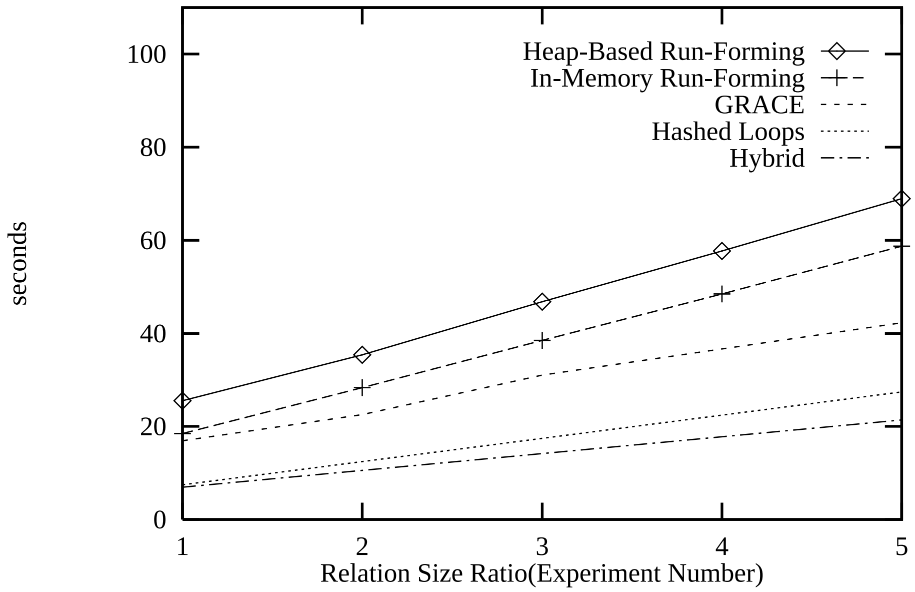


Figure 4.7: Total Time for RatioVaryRealIO Experiments

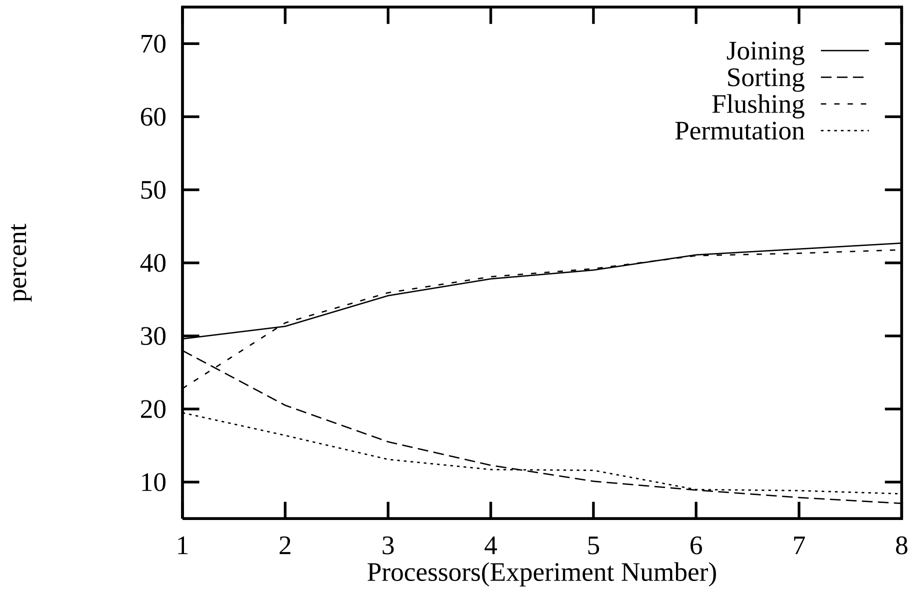


Figure 4.8: Division of Running Time of ProcessorVaryRealIO Experiments with In-Memory Run-Forming

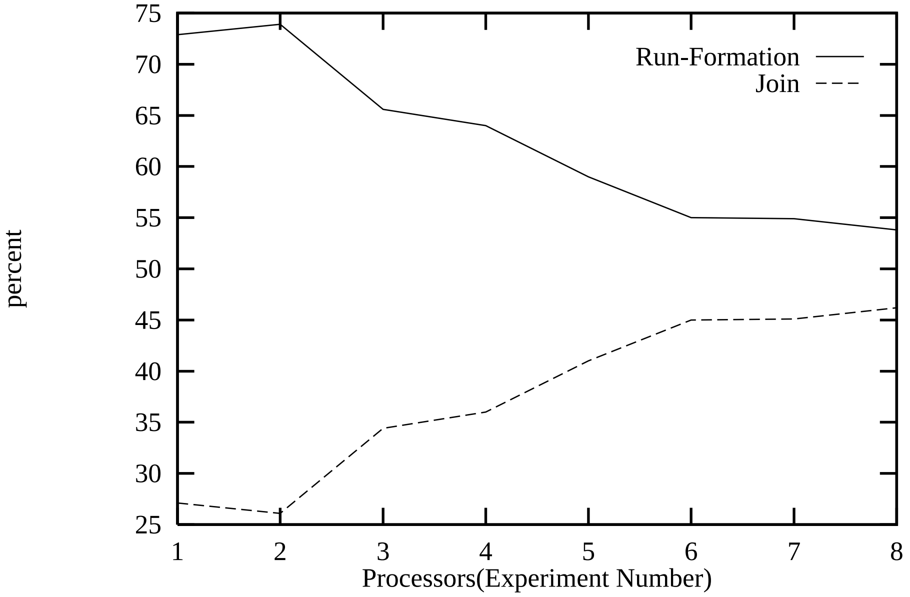


Figure 4.9: Division of Running Time of ProcessorVaryRealIO Experiments with Heap-Based Run-Forming

or betters heap-based run-forming. This superiority is particularly evident in the low-P cases, where the expense of pushing tuples through the heap costs the heap-based algorithm dearly; the run forming accounts for some 70% of the full join cost at P=4. The superiority narrows as the processor count increases, until the performance of the two algorithms is essentially identical at P=8. Here, I/O costs (which are almost identical for the two algorithms) dominate; at P=8 in figure 4.8, non-I/O costs account for a mere 15% of the total cost. It is notable that the superiority of the two approaches is decided by the run-forming phase exclusively, since join phase costs are similar. Nonetheless, when I/O is actually performed, the in-memory run-forming methods perform better or at least no worse than the heap-based method.

As can be seen from each of the figures 4.5, 4.6, and 4.7, both versions of the sort-merge join algorithm perform poorly compared to the hash-based join algorithms. For these experiments, the reasons why depend strongly on the nature and amount of I/O performed by various algorithms. The sort-merge algorithms' I/O pattern resembles that of GRACE most, since they read and write the full relations during run-forming, and then read the whole relations again during the join phase. Accordingly, the performance of GRACE is similar to the sort-merge performance, particularly in high-P cases, where processor costs of the algorithm are relatively insignificant. But GRACE still retains an advantage, and a significant one, because of its lower processing costs in the initial partitioning phase. Here, hashing is far cheaper than the full sorting performed in the sort-merge join. The difference turns out to be even clearer in section 4.2.2, where I/O costs are absent.

If the sort-merge algorithms perform poorly compared to GRACE, then their performances compared to Hashed Loops and Hybrid are still worse, as is seen most clearly in figure 4.5. Both Hashed Loops and Hybrid manage to excel by retaining memory between execution phases. In in-memory and heap-based sort-merge join, as in GRACE, no portions of the relations are retained between the run-forming and joining phases. Hybrid is essentially GRACE with one hash bucket kept in memory, and its performance is accordingly superior to GRACE when relations are relatively small. It decisively defeats sort-merge join. Even the Hashed Loops algorithm manages to defeat sort-merge join, mainly because it performs no random I/O. The fact that the memory is relatively large compared to the relations is significant here, since it means that the algorithm only has to iterate over one of the relations a few times. Two significant weaknesses of the sort-merge join method are that it retains nothing in memory between the run-forming and partitioning phases, and that it needs to use random I/O to retrieve blocks for the join phase.

4.2.2 Experiments with Simulated I/O

The ProcessorVaryRealIO and MemoryVaryRealIO experiments are I/O-less. The performance of the sort-merge algorithms is shown in figures 4.10 and 4.11, as are the predicted performances of the hash-based join algorithms, based on the Martin et al. models. The division of costs between phases of processing is shown is figures 4.12 and 4.13. Further proportion analysis results appear in appendix C. These results reflect a system where disks are plentiful enough that I/O is *not* a bottleneck, a reasonable situation in shared-memory multiprocessors.

As can be seen from figure 4.10, in-memory run-forming is decisively superior to heap-based run-forming. In-memory run-forming is about twice as fast as heap-based run-forming. Pushing tuples through the heap is just too expensive; it accounts for more than 90% of the processor time.

A few other noteworthy details are apparent from figure 4.13. First, the Flush module accounts for a miniscule portion of the costs in the absence of I/O, as could be expected. Second, the sort phase parallelizes better than the permutation phase; at P=1, it accounts for a greater share of the execution time than the permutation phase, but at P=8, it accounts for less. Third, both forms of sort-merge join parallelize well. The efficiency of the in-memory run-forming method at P=8 is 80.5%; the efficiency of the heap-based method at P=8 is 69.7%. Both are respectable results.

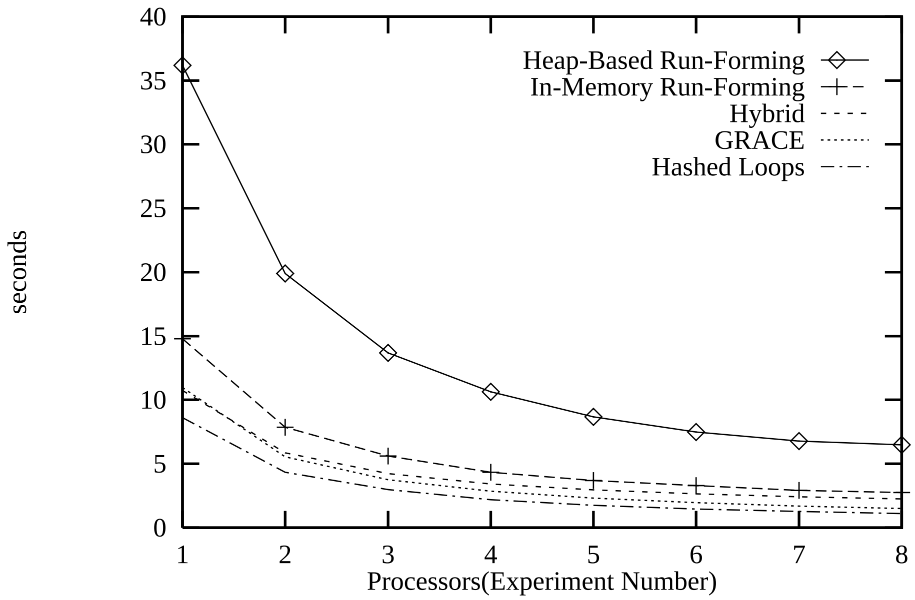


Figure 4.10: Total Time for ProcessorVaryNoIO Experiments

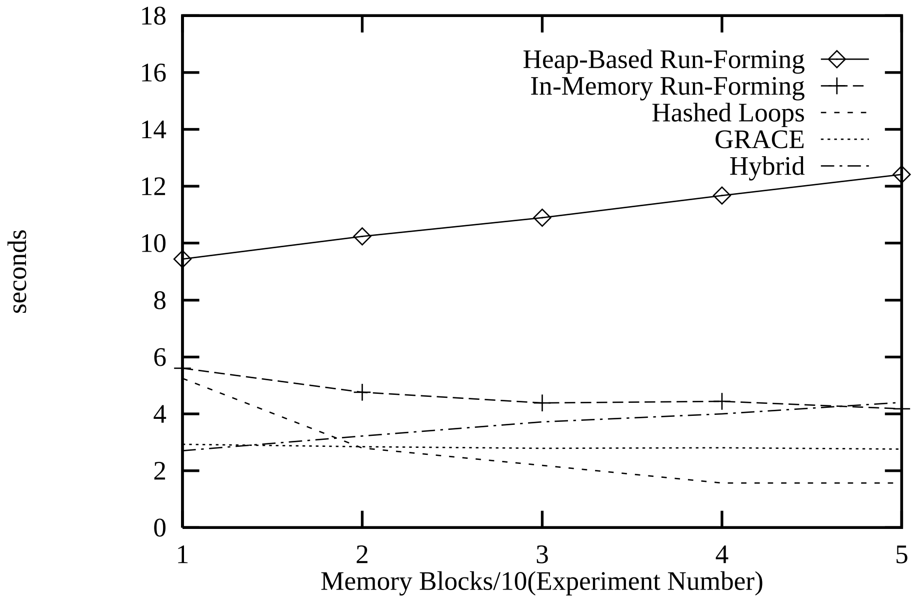


Figure 4.11: Total Time for MemoryVaryNoIO Experiments

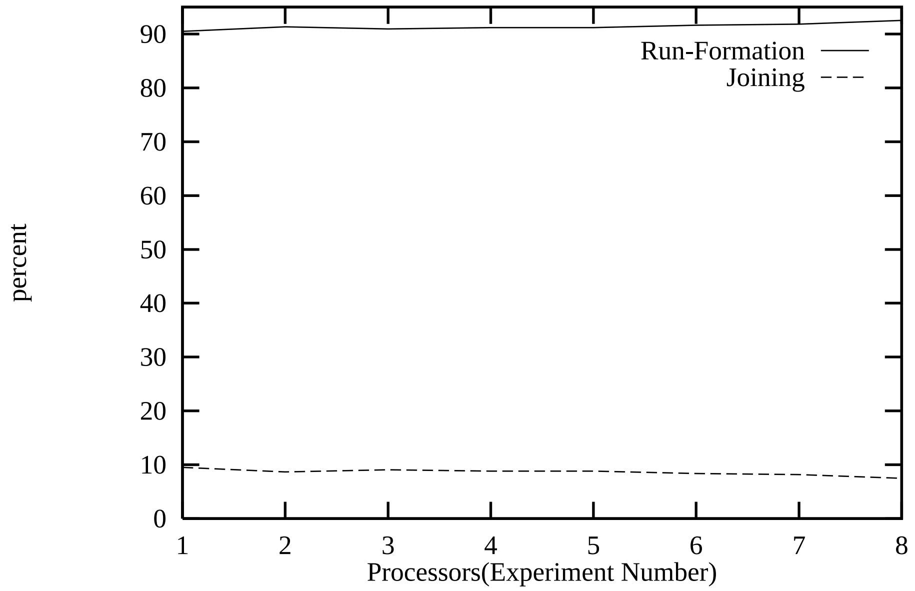


Figure 4.12: Division of Running Time of ProcessorVaryNoIO Experiments with Heap-Based Run-Forming

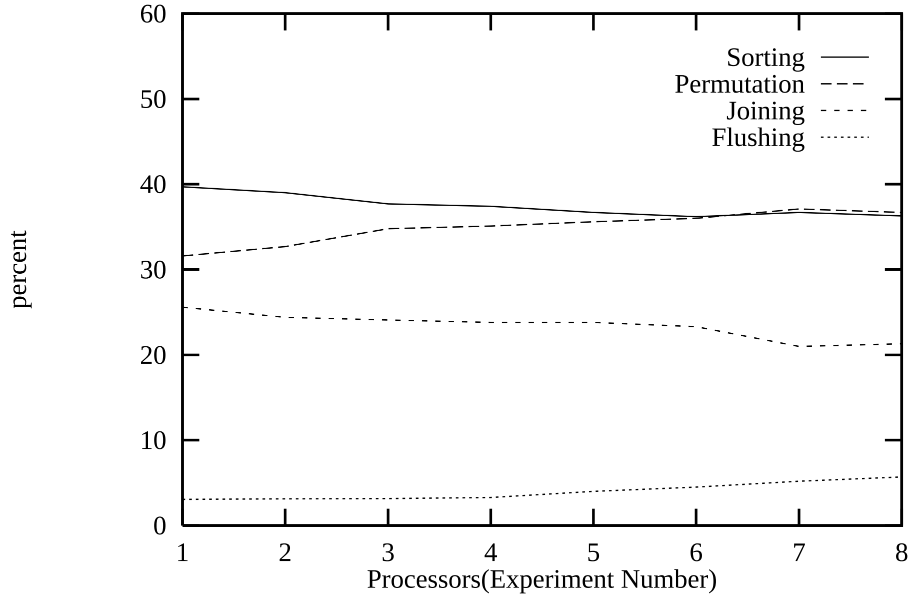


Figure 4.13: Division of Running Time of ProcessorVaryNoIO Experiments with In-Memory Run-Forming

Both sort-merge join algorithms perform quite poorly compared to the hash-based algorithms, as can be seen in figures 4.10 and 4.11. Hashed Loops is the clear winner, which is not surprising, since the more complicated processing methods of the other algorithms exist mainly to minimize I/O costs. In the absence of I/O, the simpler routines of Hashed Loops are better. However, even the more expensive hash-based algorithms are cheaper than sort-based joins. The reason why is clear. Hashing is quite simply far cheaper than sorting. The sort-merge join algorithms do too much work in forming fully sorted partitions, and this effort is not rewarded by a corresponding decrease in the joining costs.

4.2.3 Summary

In both the I/O and I/O-less experiments, in-memory run-forming proves superior to heap-based run-forming. Pushing tuples through the many layers of the heap is simply too expensive. Similarly, in all the experiments, the hash-based algorithms defeat the sort-merge algorithms. Since the experiments include both processor-bound and I/O-bound cases, there is reason to believe that the cases tested are representative of a wide range of actual environments.

The experiments demonstrate three main failings of sort-merge join algorithms:

1. Sort-merge join performs too much work during partitioning; sorting is much more expensive than hashing.

2. The sort-merge join algorithms do not retain in memory any portion of the temporary results between the partitioning and joining phases. Instead, they write out everything to disk, unlike Hybrid.

3. The sort-merge join algorithms use random I/O to read blocks for the joining. This is in contrast to the Hashed Loops algorithm, which uses purely sequential I/O.

4.3 Accuracy of Performance Models

This section evaluates the accuracy of the performance models developed in chapter 3 by comparing the predictions against actual execution results. Section 4.3.1 discusses the aggregate estimates for the entire Domain-Partitioned Sort-Merge Join algorithm. The remaining sections discuss the estimates of the various phases and sub-phases of processing.

4.3.1 Aggregate Estimates

The accuracy of the aggregate performance models is shown for each of the experiment series in figures 4.14, 4.15, 4.16, 4.17, and 4.18. The metric used in this whole section to determine the accuracy of the models is Observed Time/Predicted Time. This quantity should ideally be 1, and any difference from 1 is termed the "relative error" in this document.

The overall results are quite clear. The integrated performance models perform poorly in only one case: heap-based run-forming without real I/O. As can be seen from figures 4.17 and 4.18, the accuracy plummets as the processor count increases, reaching an error of 35% at P=8. As discussed in section 4.2.2, for the experiments, most of the time is spent on heap-processing. The performance models for the parallel heap processing perform poorly, as is explained in the section 4.3.2.

In all other experiment series, the performance models' predictions are typically within 10% of the actual value. There are a few outliers, particularly in cases with extreme processor counts, or extreme relation sizes. But even these outliers have errors less than 20%. Interestingly, the relatively poor Join Phase predictions in I/O-less cases do not show up in the aggregate results. They are swamped by the run-forming modules, which use far more processor time.

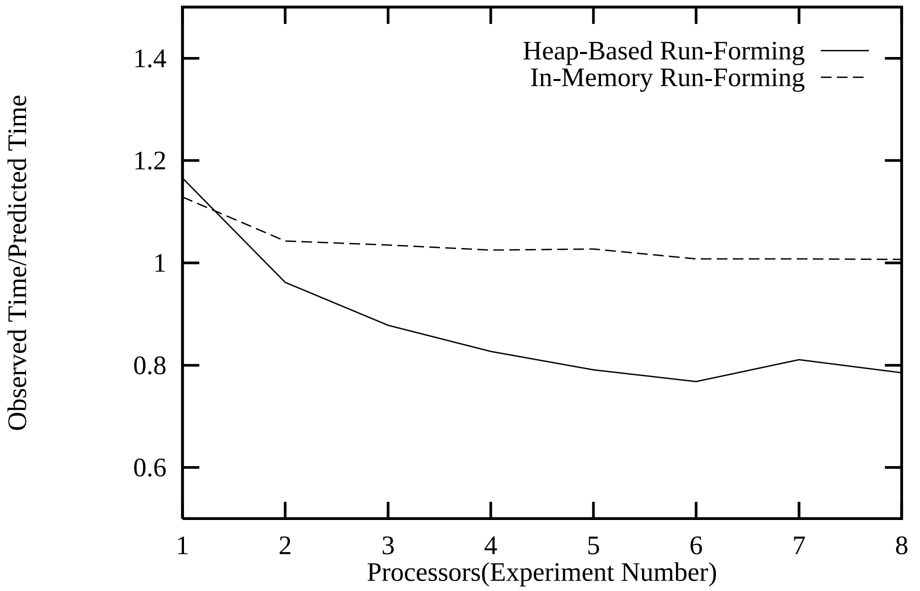


Figure 4.14: Aggregate Prediction Accuracy for ProcessorVaryRealIO

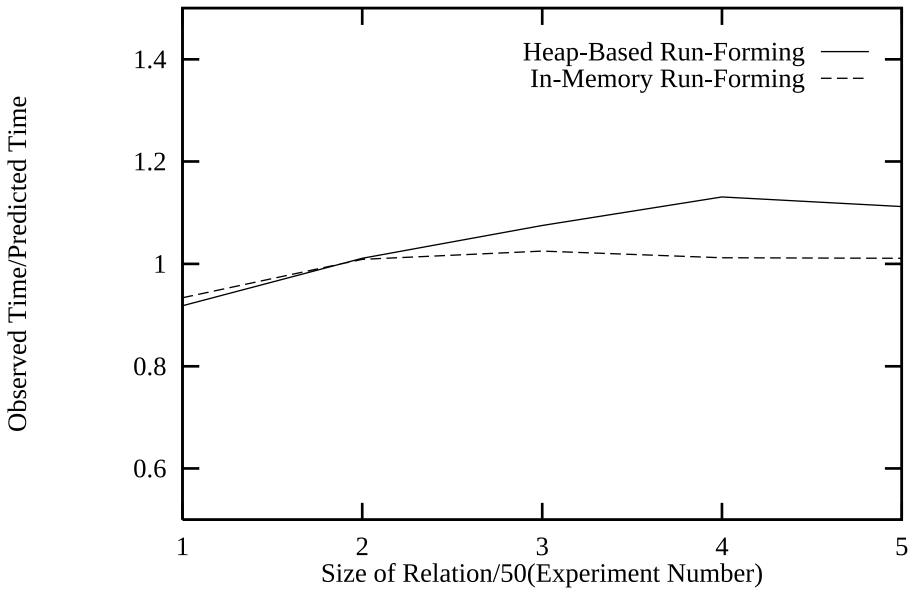


Figure 4.15: Aggregate Prediction Accuracy for SizeVaryRealIO

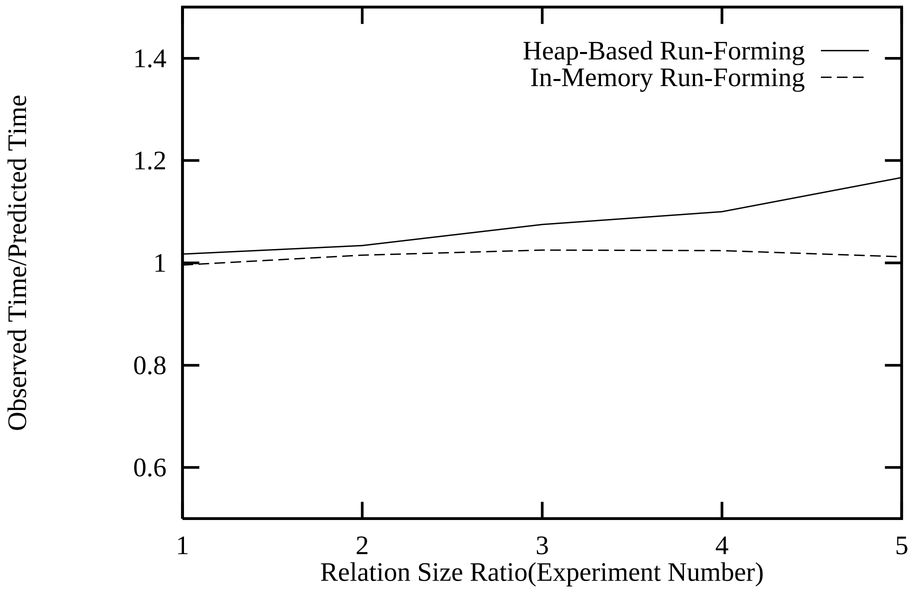


Figure 4.16: Aggregate Prediction Accuracy for RatioVaryRealIO

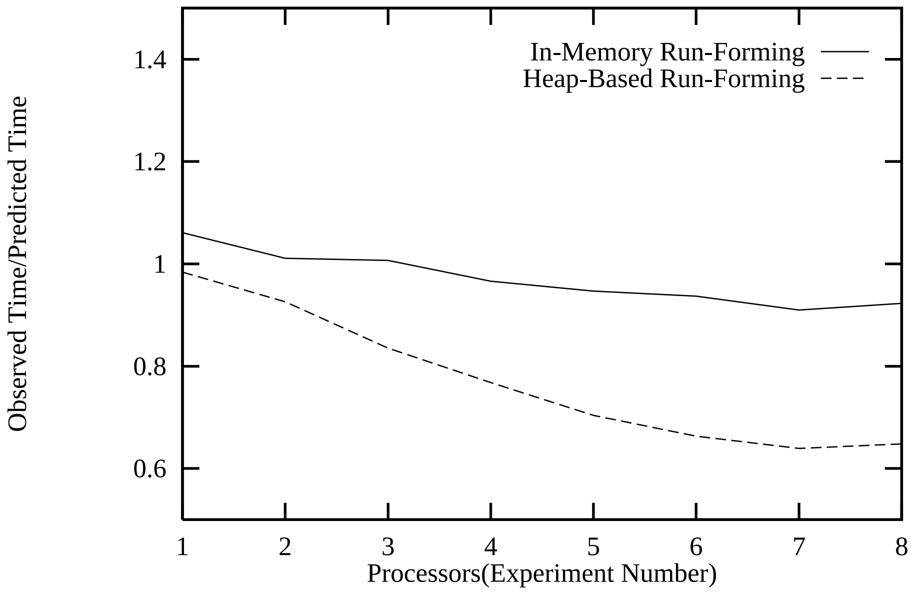


Figure 4.17: Aggregate Prediction Accuracy for ProcessorVaryNoIO

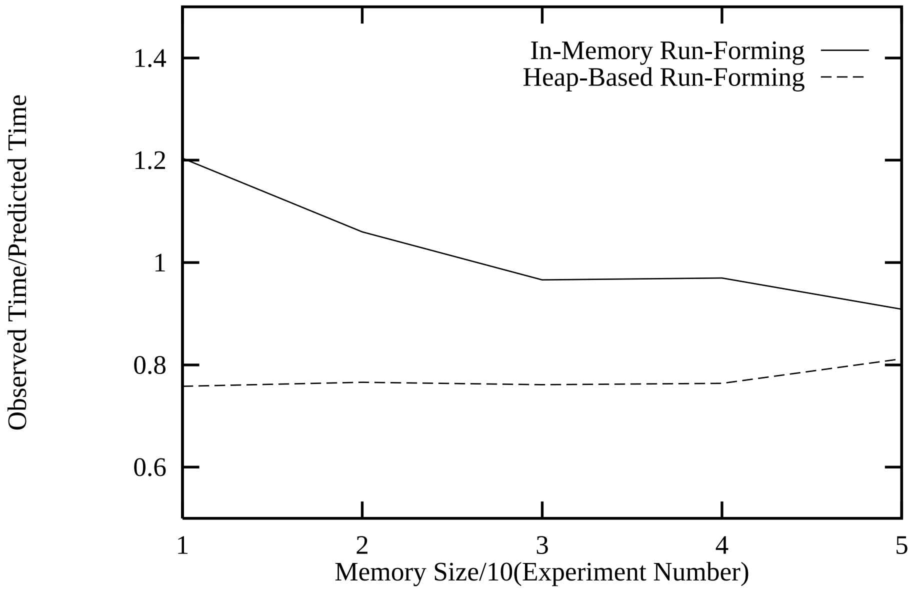


Figure 4.18: Aggregate Prediction Accuracy for MemoryVaryNoIO

4.3.2 Heap-Based Run-Forming

The predictions for the heap-based run-forming subphase are evaluated in figures 4.19, and 4.20. Both of these deal with the experiments with real I/O. Here, the predictions are acceptable, but not good. They are generally overly optimistic by some 15-20% for the SizeVaryRealIO experiments, but overly pessimistic by as much as 30% in the case of the ProcessorVaryRealIO experiments. The high error is bad enough; the fact that the estimates are not consistent in their direction of error is even worse.

The heap-based run-forming predictions for the I/O-less experiments can be found in figures 4.21 and 4.22. Here, the predictions are at least consistently pessimistic. The pessimism increases sharply as the number of processors increases in figure 4.21. The degree of error is very high; it reaches 35% for high-P cases in the ProcessorVaryNoIO experiments, and hovers around 25% even in the MemoryVaryNoIO experiments, where P=4.

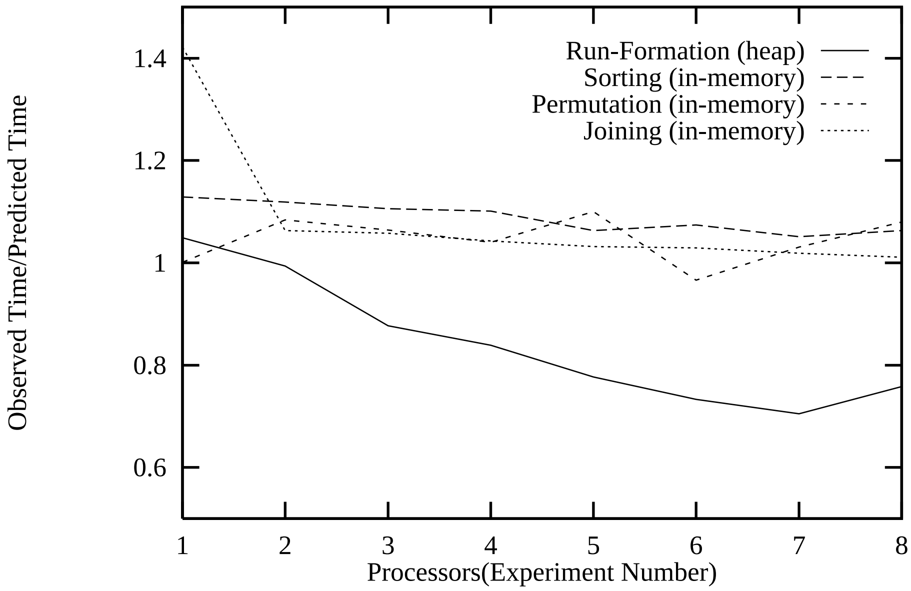


Figure 4.19: Prediction Accuracy for ProcessorVaryRealIO Experiments, Run-Forming Phases

Thus, the heap-based run-forming predictions are at best marginally acceptable, particularly since they are not consistent in the direction of their error.

4.3.3 Joining

The performance of the prediction algorithms in the experiments with real I/O is shown in figures 4.23 and 4.24. In these cases, the join phase is I/O bound, and thus the estimates are essentially based on computing the number of blocks to be read. As can be seen, the estimates are superb. They are typically well within 5% of the actual performance. The only exception occurs when P=1, where the estimates can be grossly optimistic. This may be due to the peculiar nature of the algorithms when P=1. The estimates are based on the assumption that processor and I/O performance overlap seamlessly. This may not be true when P=1, since there are no other processors to do work when one processor is waiting for data.

In the I/O-less experiments, the performance of the predictions can be found in figures 4.25 and 4.26. These predictions are different from the cases where real I/O is performed, since the estimates are based on the processor cost of the joining, rather than the I/O cost. The predictions themselves are consistently pessimistic, and their error varies from about 10% for small processor counts and relatively small relations, to about 35% for high processor counts and relatively large relations. Although the consistency of error direction is good, the high absolute error makes these estimates only decent.

4.3.4 Sorting

The prediction algorithms for the sort costs perform well. Since I/O is never carried out concurrently with sorting, all experiments are essentially I/O-less for this subphase. The ProcessorVaryRealIO and SizeVaryRealIO sort performance predictions are to be found in figures 4.19 and 4.20. The ProcessorVaryNoIO and MemoryVaryNoIO predictions are in figures 4.21 and 4.22.

The predictions are good; they are generally optimistic, but typically within 10% of the actual performance. The accuracy generally increases with the processor count, but decreases slightly with increasing relation sizes. There is only one exception to these trends: MemoryVaryNoIO-1, where the prediction is 74% too optimistic. In this experiment, the runs sorted are very small; cache contention may well be the problem.

4.3.5 Permuting

The permutation subphase is very much like the sort subphase. It too is performed without any I/O, and its performance, too, is predicted accurately by the estimation functions. The predictions are generally slightly pessimistic, and the error is generally less than 10%. This is excellent performance by the estimation functions. The permutation predictions' accuracy is displayed in figures 4.19, 4.20, 4.21, and 4.22.

4.3.6 Flushing

The performance of the prediction functions for the Flush subphase depends strongly on whether or not I/O is actually performed. When I/O is performed, as in the figures 4.19, and 4.20, the estimates are very accurate, typically within 10% of the actual value. This is because these cases are I/O bound, and as was the case with the join costs in section 4.3.3, I/O costs can be estimated very accurately.

When I/O is not actually performed, as is the case in figures 4.21 and 4.22, the estimates are for the actual (small) processor costs of the flush operations. These estimates tend to be less accurate than the I/O-based estimates. The estimates themselves are reasonable; they tend to be optimistic, and their errors hover around the 15-20% mark. The error increases with the processor count. The performance in the MemoryVaryNoIO series is particularly odd; here, the accuracy fluctuates wildly from case to case.

4.3.7 Overview Of Prediction Accuracy By Phase

The overall performance of the estimation functions for the various phases and subphases is:

heap run-forming (marginal); inconsistent error direction; error around 25%.

joining (superb) when I/O bounded; error approximately 5%. (decent) without I/O; error increase to 35% at high P-count; consistently pessimistic.

sorting (good); 10% error bound; generally optimistic.

permutation (good) 10% error bound; generally pessimistic.

flushing (good) when I/O is real; 10% error bound; pessimistic. (decent) without I/O; 15-20% error; optimistic.

In general, I/O dependent costs are predicted very well. The join phase predictions are superb when the phase is I/O bound, but only decent in the I/O-less experiments. Similarly for the flush subphase, the predictions are good for the experiments with real I/O, but only decent in the I/O-less cases.

The processor-time predictions are of mixed quality. The sort and permutation subphases are estimated well; their prediction errors are around 10%. The only real disappointment among the prediction functions is the heap-based run-forming phase. The prediction function is of only marginal utility, since it is inconsistent in error direction, with an error around 25%.

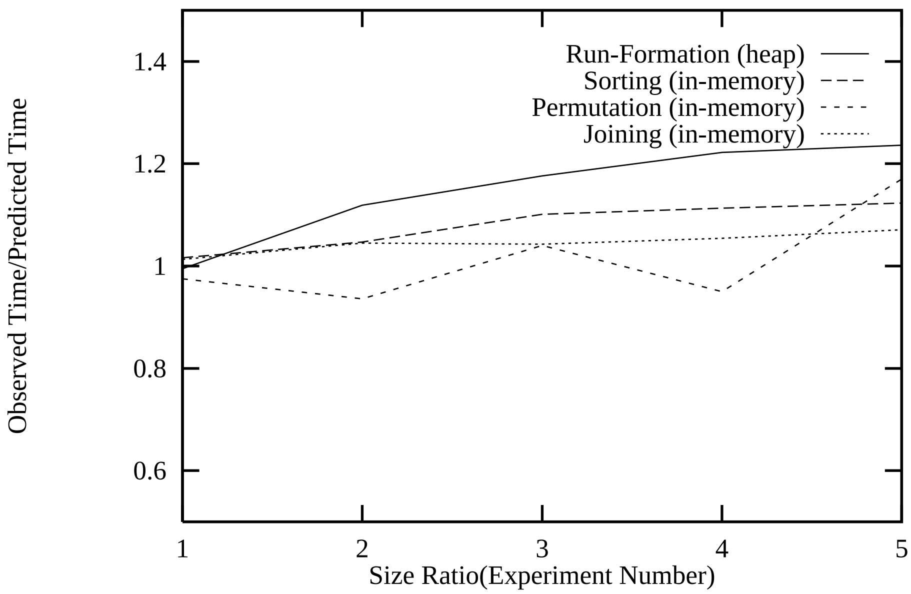


Figure 4.20: Prediction Accuracy for SizeVaryRealIO Experiments, Run-Forming Phases

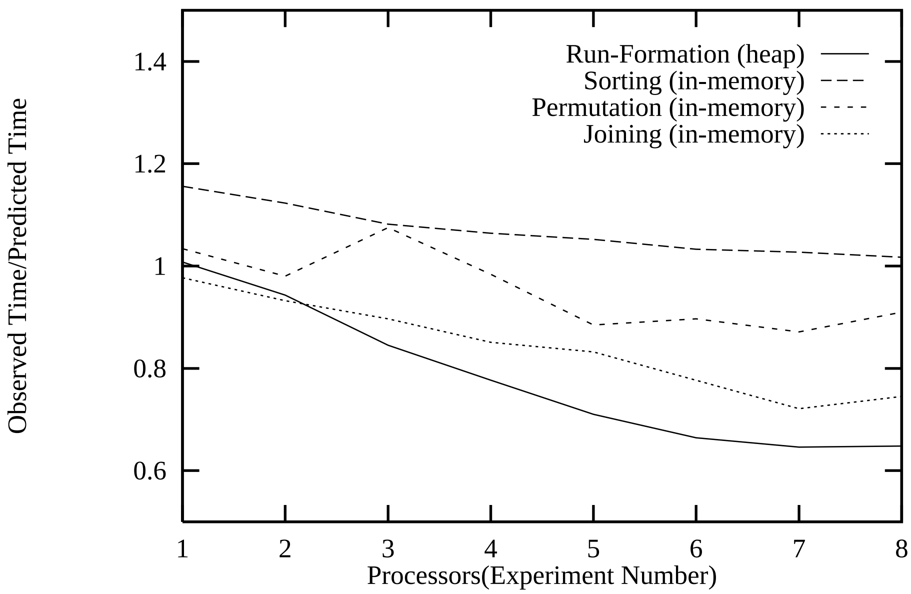


Figure 4.21: Prediction Accuracy for ProcessorVaryNoIO Experiments, Run-Forming Phases

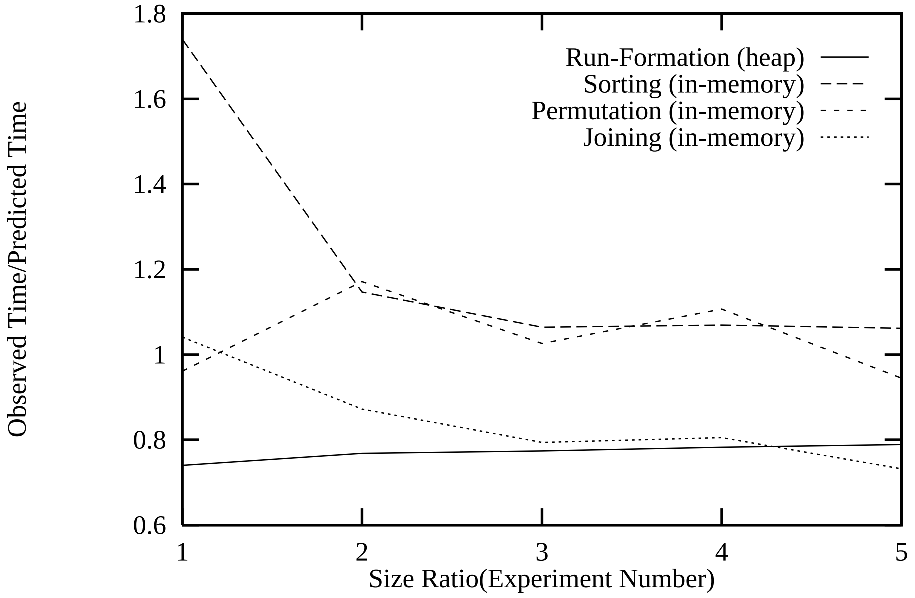


Figure 4.22: Prediction Accuracy for MemoryVaryNoIO Experiments, Run-Forming Phases

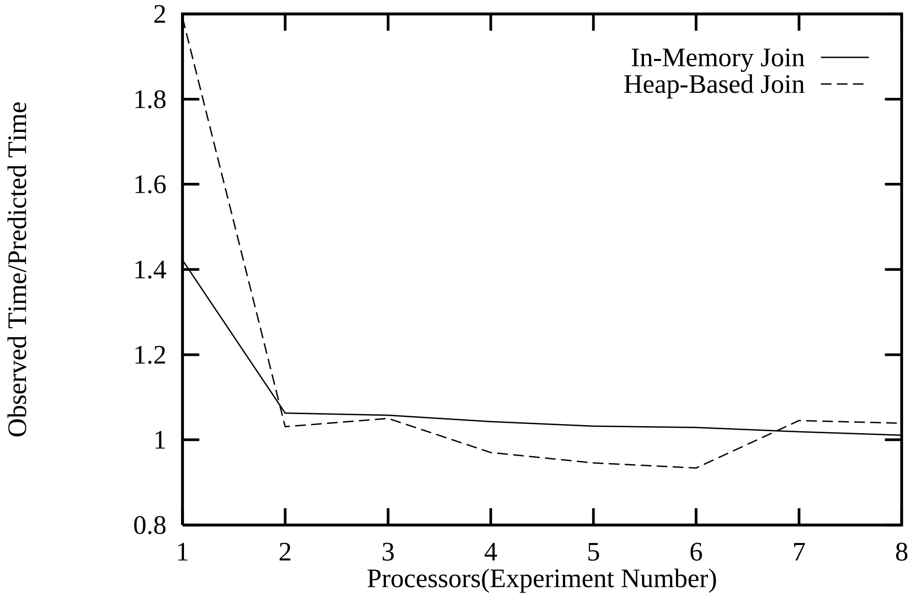


Figure 4.23: Prediction Accuracy for ProcessorVaryRealIO Experiments, Join Phase

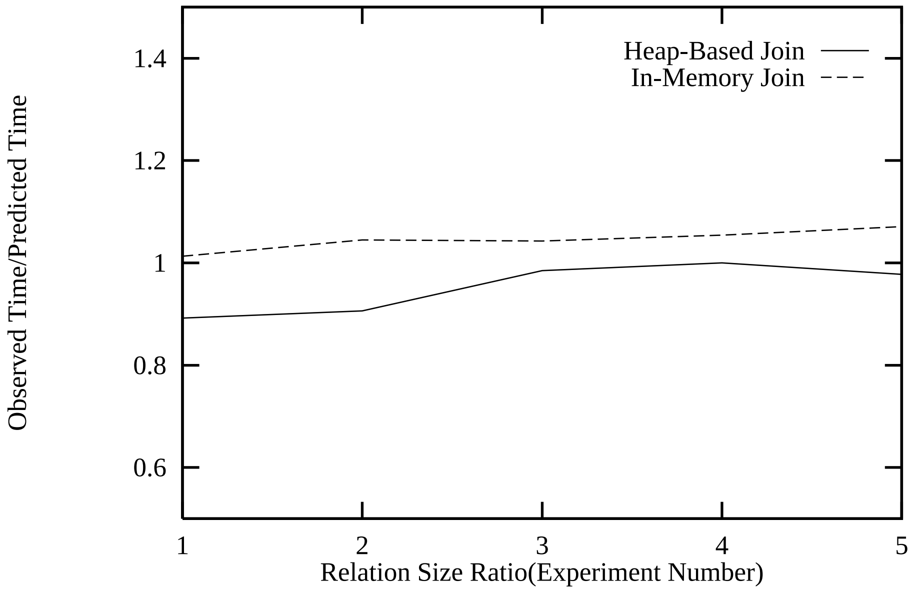


Figure 4.24: Prediction Accuracy for SizeVaryRealIO Experiments, Join Phase

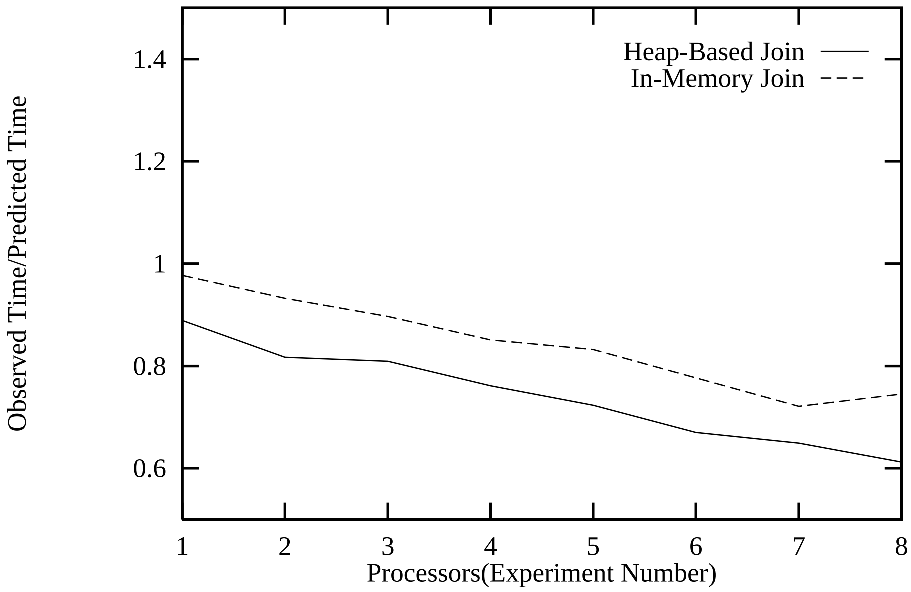


Figure 4.25: Prediction Accuracy of ProcessorVaryNoIO Experiments, Join Phase

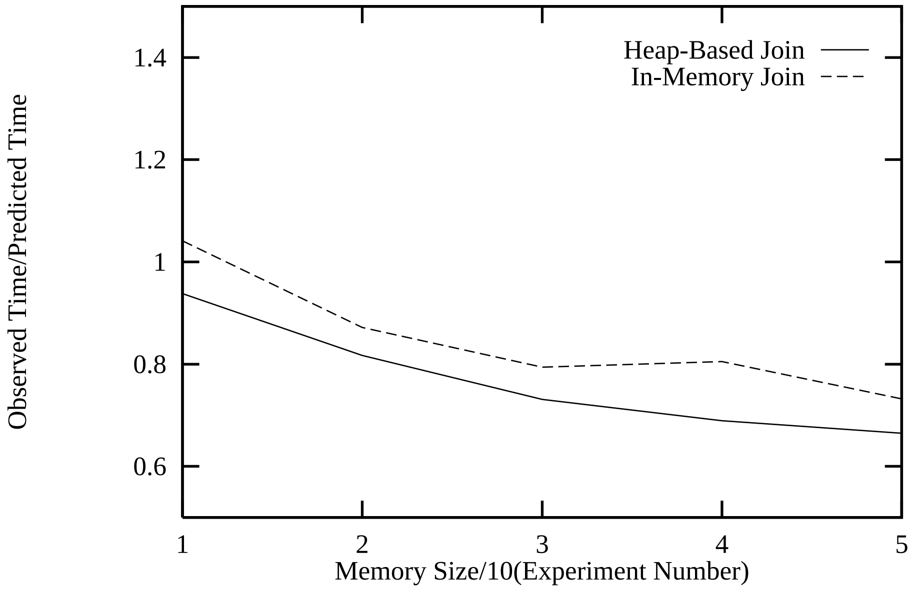


Figure 4.26: Prediction Accuracy for MemoryVaryNoIO Experiments, Join Phase

Chapter 5

Other Issues

This chapter briefly addresses a number of peripheral issues regarding the performance and use of the Domain-Partitioned Sort-Merge Join Algorithm. The treatment is not comprehensive; some issues are raised without having full solutions, but only suggestions for further research. The three sections of the chapter address

- the buffer use of the DPSMJ algorithm,
- the use of DPSMJ in joining with indices, and
- joining with pre-sorted relations.

5.1 Buffer Use

The GRACE and Domain-Partitioned Sort-Merge Join algorithms have a problem in common. For any given memory size and page size, these two algorithms have limits to the size of relations they can process. For the GRACE algorithm, the limit is due to the necessity of keeping each hash-bucket of the smaller relation smaller than memory, and by the need to have two buffers available for each hash-bucket during partitioning. For the DPSMJ algorithm, the limit occurs because of the (roughly) fixed size of the runs, and the fact that two buffers are needed for every run for every

Parameter	Meaning
R	number of pages of the smaller relation
S	number of pages of the larger relation
D_R	number of runs produced(in DPSMJ) for the smaller relation
D_S	number of runs produced(in DPSMJ) for the larger relation
M	number of pages of memory
p	number of processors

Figure 5.1: Buffer Use Analysis Parameters

Joiner during the Join Phase. Generally, the smaller the page size used, the larger the relations that can be processed. Of course, I/O transfer rates generally decrease as page sizes decline, and there is a practical minimum, namely the disk sector size. Hence, the number of buffers needed by an algorithm is an important characteristic, and measures should be taken to minimize the required buffer count. DPSMJ needs a lot of buffers to work effectively.

5.1.1 Buffer Use Limits

The derivation of the limit formulae for DPSMJ is as follows. Note that this is an approximate analysis, and hence some liberties are taken. In particular, remainders from divisions are ignored, and memory used for purposes other storing tuples is neglected. The parameters used in this analysis are shown in figure 5.1.

In the Join Phase, all p Joiners need two buffers (assuming double buffering) for every run for each of the two relations, hence

$M \geq p(D_R + D_S)2.$

From the Sort Phase, we know that

$D_R = R/M, D_S = S/M.$

Hence, $M \geq 2p(R/M + S/M)$

and $M^2 \geq 2p(R+S)$.

Finally,

$$M \geq \sqrt{2p(R+S)}. \tag{5.1}$$

The equivalent inequality for heap-based run-forming is

$$M \geq \sqrt{p(R+S)}. \tag{5.2}$$

The difference between formulae 5.1 and 5.2 is due to the double-length runs produced by heap-based run-forming.

According to Martin, Larson and Deshpande [13], the buffer use limit for GRACE is

$$M \geq \sqrt{2R}. \tag{5.3}$$

Notably, Martin, Larson, and Deshpande remark that this is a relatively severe restriction. In the formula, the factor of two reflects double buffering, which is used to write the pages of hash-buckets out to disk.

5.1.2 Buffer Use Problems of Sort-Merge Join

Formulae 5.1 and 5.2 indicate two fundamental problems with the DPSMJ algorithms, problems that are shared by all versions of the DPSMJ algorithm. First, the formulae depend on the sum of the sizes of the two relations to be joined, rather than the size of the smaller relation. This is significant, particularly if the joining relations have very different sizes. Second, the number of buffers increases with the (square root of the) number of processors. Thus, the memory size required quickly becomes prohibitive when large-order multiprocessors are used. The GRACE algorithm has neither of these two weaknesses.

There does not appear to be a solution for the first of these problems. The second one, however, can be managed in some circumstances. The factor of p in the formulae is the number of Joiners used, typically with one Joiner per processor. Thus, to join very large relations when memory is scarce, the algorithm can run fewer Joiner processes. This allows graceful degradation in critical situations. Actually, running fewer Joiners might not hamper performance at all, given the low processor-cost of the Join Phase, since the Join Phase is almost certainly I/O-bound anyway.

5.1.3 Eliminating Double Buffering

The factor of two in formulae 5.1 and 5.2 reflects double-buffering, the most straightforward way to assure good computing and I/O overlap[18]. The double-buffering is used in the Join Phase, when reading in blocks from the runs to be merged and joined. The factor of two can be reduced nearly to one using the technique of *forecasting* [10] [24] to optimize the merging of the runs.

Forecasting uses one buffer per run, plus one extra buffer. Its fundamental idea is that it is possible to determine, given a set of blocks being merged, which block will be exhausted first. Hence, a request to read (into the extra buffer) the next block in that run can be dispatched before the buffer is actually exhausted. Identifying which block will be exhausted first is easy: the block with the lowest final tuple value will be exhausted first.

A simple implementation of forecasting would have one buffer for every run of each of the relations within each Joiner, plus one buffer for each of the relations within each Joiner. Here, forecasting could be done separately for each relation. However, given the similarity of merge-joining and simple merging, the forecasting could be made over both of the relations at once. This would save one buffer for each Joiner, which might be significant in cases with high parallelism or limited memory.

Actually, since the secondary information collected as part of the run forming phase includes the join-field value of the last tuple of every block, it is possible to predict not only the next block to be replaced, but the entire sequence of block reads to be made by a Joiner in processing a partition. Note that this prediction is possible only for each individual Joiner. When multiple Joiners are in operation, the aggregate request stream can be any interleaving of the Joiners' request streams.

5.2 Sort-Merge Joins with Indexes

This thesis has considered Domain-Partitioned Sort-Merge Join as a method for joining base relations. In modern database systems with plentiful indices, base relation joins are relatively rare. This section considers the more common situation of joins with indices, and the applicability of the Domain-Partitioned Sort-Merge Join algorithm to such joins. The general case considered is the join of two relations A and B, with an index XB on relation B. The index XB is on the join attribute. In this section, it is assumed that the index is a B+-tree [5], a common but not universal index structure.

5.2.1 Conditions

When a join is to be performed, it is possible that an index should not be used at all, even if one is available and relevant to the join. In particular, the presence of an index is an issue only if the selectivity of the join is low (i.e., the portion of elements of the joined relations participating in the result is low.) If this is not the case, the wisest policy is to read all blocks of the base relations, and to save time by reading them sequentially, rather than trying to use the index. Use of the index would probably result in reading in most of the blocks anyway, and the reading would be done with random rather than sequential I/O. Thus, in looking for cases where merge-join can be useful when joining with an index, it is safe to assume that it will be in cases where the join has low selectivity.

It is also safe to ignore cases where the memory available is large compared to the index. If essentially all of the index fits into memory, the best algorithm is a nested loop join. In this case, XB can be loaded into memory, and A scanned sequentially. As each tuple of A is brought into memory, it is checked for a match in the index.

In those few cases where a match results, a random I/O can be issued to bring the joining tuple of B into memory. This scheme works perfectly if all of the index is in memory, but should be workable even if not quite the whole index can stay in memory at once. If necessary, a few iterations can be performed, loading in part of XB, and using it for a scan over A, and then loading in the rest of XB for another scan over A. This only becomes prohibitive when XB is large relative to memory, necessitating too many scans over A. When this is the case, a more sophisticated join algorithm must be used, essentially performing a join of A and XB, which is then filled out with tuples from B itself.

The merge-join algorithm generally becomes competitive in situations like this where both of the relations (or relation A and index XB) are significantly larger than memory. Then, it becomes practical to read in, sort, and write out the relations before they can be used, so that they can then be joined in a single scan over the two sorted relations. In the particular case of joining A and XB, one of the relations is already sorted: the leaf level of the index is in sorted order, by the join attribute. Thus, only A has to be sorted into join-attribute order. Given the size of the final relations in a situation like this, since the relations are large compared to memory, it is important to retrieve the joining tuples of B from disk using a method more efficient than random reads on demand. One-by-one random retrievals would result from doing a merge-join between XB and A, with B consulted only when a join tuple is found.

5.2.2 Division of Index-Hybrid Join

Merge-Joining A and XB, and then filling out the tuples with single random retrievals from B becomes inefficient when the result size is large. The alternative is to do what is called a Hybrid join [4] in IBM's DB2 product; to distinguish it from the other hash-based Hybrid algorithm, IBM's join method will be designated the Index-Hybrid algorithm. Using this method, A is first sorted into join-attribute order, and then A and XB are merge-joined. (XB is already in sorted order for this join.) The output of this join is a temporary table IT. IT is then sorted by the logical disk location of the corresponding tuples of B. IT and B are then accessed in this sorted order to retrieve the required tuples of B. Since the I/O accesses are in order by logical disk location, the operation requires only a single sweep of the arm across the disk. It is hence very fast.

The Index-Hybrid Join can be decomposed into two back-to-back merge-joins. The first merge-join joins A with XB to form the temporary result IT, joining on the original join attribute. The second merge-join joins IT to B on the disk location of the B-tuples. Disk location is a real rather than relational attribute of a tuple attribute of every tuple. Both merge-joins can be parallelized into Domain-Partitioned Sort-Merge Joins. In the first join, the issue is formatting and partitioning the upper levels of the B+-tree index to permit the join to occur. In the second join, the issue is partitioning B, since it is already (trivially) in sorted order by disk location. This case is not worth parallelizing if only one disk is used for B, since a single scan is be optimal in that case.

5.2.3 The First Merge-Join: A and XB

The first of the two sort-merge join operations joins A and XB. Relation A has to be processed normally: sorted into runs, sampled and partitioned. The index XB does not need to be sorted. Assuming it is organized into a B+-tree, it is already in the required sorted order. However, it does need to be partitioned for use with the Domain-Partitioned Sort-Merge Join algorithm.

Basic Structure of a B+-tree

A B+-tree index is a tree consisting of two distinct types of page-sized nodes: interior nodes and leaf nodes (see figure 5.2.) The leaf nodes contain the actual index entries, full Record Identifiers(RIDs) referring to the indexed relation. These RIDs appear in

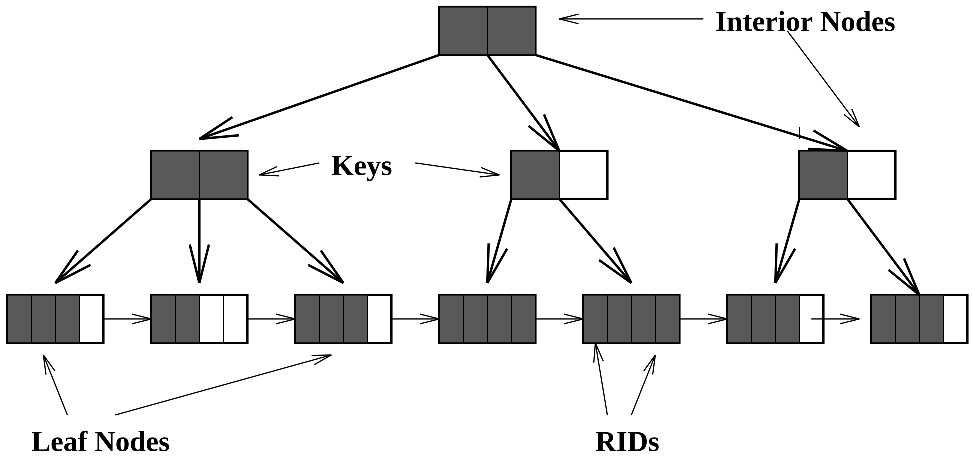


Figure 5.2: B+-tree, Whole Structure.

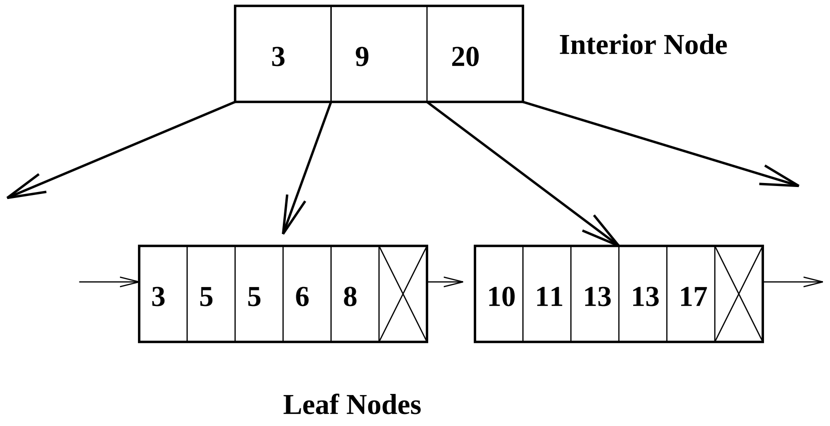


Figure 5.3: B+-tree, Interior and Exterior Nodes.

sorted order in the leaf nodes, and the leaf nodes themselves are linked together to permit easy sequential access of the index entries in sorted order. The upper levels of the tree consist of interior nodes, which are used for random access of the index. They are used to direct a query from the root of the tree to the appropriate leaf node. Interior nodes do not contain actual RIDs. Instead, they contain key values and pointers to descendant nodes in the tree. Each node i contains some number k_i of key values and $k_i + 1$ pointers (see figure 5.3). These key values are used in the descent from the root to select the appropriate node to access next.

Partitioning Using the B+-tree

The index XB must be partitioned for use with the DPSMJ algorithm. In the standard algorithm, the key value of the last tuple in every block of the sorted runs is recorded in the BlockIndex array. This is a sampling of the join-attribute values of the tuples, with a density of one sample per block. It is used to partition the relation. To partition XB, a similar sampling of the RIDs of XB is needed. This sampling could be gathered by scanning over the leaf nodes and thereby gathering a full sampling of the index values, but this direct approach uses too much I/O, and should be avoided. Instead, it is possible to derive a set of sample data from the interior nodes. This method uses less I/O, since there are fewer interior nodes than leaf nodes.

An in-order traversal of the interior nodes of the tree produces a sequence of alternating key values and pointers: $p_0 k_1 p_1 k_2 \ldots p_{n-1} k_n p_n$, with pointer values p_j and key values k_j. This sequence can be augmented with a final $k_{n+1} = +\infty$, and obeys the property that all RIDs in the leaf nodes accessed through p_i have key values $t \leq k_{i+1}$. This relationship between the k_i values and the leaf nodes is equivalent to that between the BlockIndex key value samples and the data blocks in the standard DPSMJ algorithm. Thus, the same partitioning algorithm that is used in the standard algorithm (see section 2.5.1) can be used for the Calculation phase of this special case. (Note that there is one key value per block of RID entries using this approach, a sampling density equal to that of the standard algorithm.)

Forming the actual sequence is easy. Since nodes are large and therefore of high order, B+-trees are seldom more than four levels deep. The inorder traversal only requires one node-sized buffer per level. This is a small number, even if a few extra buffers are added for pre-fetching of the interior nodes. This prefetching can be done efficiently since the traversal order of the tree is known.

5.2.4 The Second Merge-Join: IT and B

The first merge-join joined A and XB to produce IT. The second joins IT with B. Each tuple of IT is a tuple of A, augmented with the RID of its joining tuple in B. IT is initially in the order of the original join. The join between IT and B is on the disk location of the tuples of B. On the assumption that the system is I/O-bound, the goal is to avoid random I/O on the disks by accessing B in disk order with a single scan across each of the disks on which it resides. For the join, IT first has to be sorted into the order of the tuples of B on disk. This is essentially a standard Sort Phase, albeit with a rather odd sorting function. The partitioning, however, is not standard.

First, if B is a single disk, then no partitioning should take place. In this case, the best policy is a single scan over the disk, reading in each required block when sweeping across the disk. If there are more disks, the best policy is disk-wise partitioning, with the partitions consisting of the portion of B on each disk, and IT partitioned accordingly. Thus, the processing of each partition involves a single scan over a single disk, with a single Joiner reading in blocks of B in that order, and joining each required tuple of B to a tuple in IT.

It is not entirely clear what to do when the number of disks exceeds the number of processors. To maximize I/O performance, the ideal situation is to read from all of the disks at once. This cannot be done with each partition corresponding to one disk, and with each processor running one Joiner processing one partition. The question would require further research, and is beyond the scope of this thesis. There are two intuitive solutions. One solution is to run multiple Joiner processes on each processor. In an I/O bound system, there is time to multitask among the multiple Joiner processes while waiting for I/O, particularly if lightweight threads are available. The other solution is to devise an ordering function for the locations of the tuples of B that interleaves the tuples that are resident on a set of disks, so that they are contiguous in the access sequence, when sorted. A single Joiner could then be assigned a partition corresponding to the interleaved access of several disks, and would access the tuples on each of them in parallel, sweeping across all the disks in the set simultaneously.

5.3 Pre-Sorted Relations

When performing a join, it is possible that the relations to be joined will appear in sorted order on the join attributes, either as base relations, or as temporary results when executing a query plan. This sorted order may either have been deliberately maintained, or occurred as a fortunate (and exploitable) side-effect of a previous operation. Given that much of the work of performing a Domain-Partitioned Sort-Merge Join (DPSMJ) is the deriving of a sorted order (at least in the form of sorted runs), it seems that the algorithm would perform well in cases where the inputs are already sorted. This turns out to be true in two specific cases.

Simply knowing that the tuples of a relation are sorted is not enough for DPSMJ. Note that for DPSMJ to work, it needs not only the relations in sorted order, but also the sampling data of the BlockIndex array. This data consists of the join-attribute entry of the last real tuple in every block of the sorted relations. If this data is not available, then DPSMJ cannot partition the joining work, and becomes classic Sort-Merge Join without the initial sorting of the relations.

It is reasonable to assume that sampling data is available for any relation that is in sorted order. Sorted order does not happen by itself. A relation either has to be deliberately maintained in sorted order (at great expense) if it is a base relation, or it deliberately appears in sorted order as a temporary result of a larger query plan. In either case, assuming that the database system encorporates DPSMJ as one of its principal join methods, it is reasonable to expect that in addition to maintaining a relation in sorted order, or knowingly producing a temporary result in sorted order, the system will also collect block-wise sampling data. This is reasonable, since the

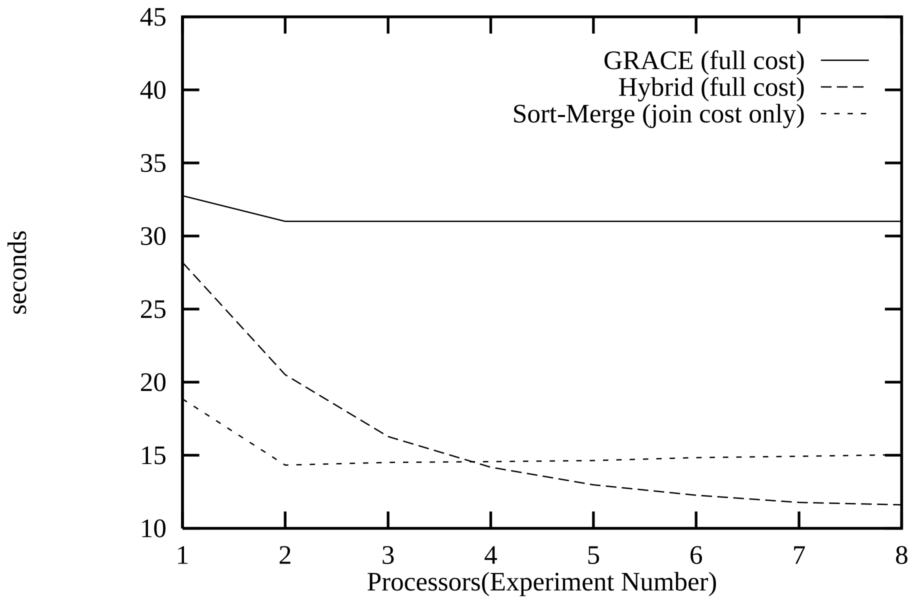


Figure 5.4: ProcessorVaryRealIO Series Pre-Sorted Times

sampling data is compact and easy to collect. Similarly, when producing a temporary result in sorted order to be joined, the system would collect the sampling data as the relation is produced.

5.3.1 Pre-Sorted Join versus Standard Hash-Based Join

This section compares the performance of DPSMJ for sorted and sampled relations against the standard versions of the hash-based join algorithms (see section 1.1.2). Note that when fully sorted and sampled relations are available, DPSMJ can omit the Sort Phase entirely, leaving only the Calculation and Join Phases, of which only the latter takes a significant amount of time. The results presented are based on the timings for the Join Phase of DPSMJ versus the full execution time estimates for the hash-based join algorithms. Since pre-sorted and pre-sampled relations are assumed to be available, the Sort and Compute Phases of DPSMJ are omitted. The results are from the experiments described in chapter 4.

I/O-Bound Experiments

The results for the cases where real I/O was used are shown in figures 5.4 [1] (ProcessorVaryRealIO experiments) and 5.5 (RatioVaryRealIO experiments.) The figures show the performance of two types of hash-based join methods. Hybrid is included because it performs best, particularly when relations are small compared to memory. When relations are large, Hybrid's performance approaches that of GRACE, and GRACE is therefore included.

In the case of large relations, the access to pre-sorted relations is decisive. DPSMJ does much better than GRACE, since it manages to avoid reading in raw relations and writing out temporary results. Since both methods have to do similar amounts of random I/O (GRACE, to write the temporary results, and DPSMJ, to read the temporary results), but DPSMJ manages to avoid all sequential I/O. DPSMJ is decisively superior in cases with large relations which are pre-sorted and pre-sampled. In the case of relatively small relations, DPSMJ manages to do well in all but the small-relation cases, but in the other (more realistic and more interesting) cases, Hybrid manages to beat DPSMJ, despite the latter's access to pre-sorted relations.

The superiority of Hybrid in cases with small relations even when DPSMJ has the great advantage of pre-sorted relations is due to two factors: Hybrid's use of memory to store part of the relations in memory between the partitioning and joining phases, and DPSMJ's use of random I/O to read in blocks during the Join Phase. The two factors are related, and synergistic. First, in an I/O-bound system, much of the I/O cost of DPSMJ is located in the Join Phase. The DPSMJ algorithm ends up reading the full joining relations during the Join Phase, and this reading is done using random

[1] The execution time curve for the Sort-Merge Join has a strange but explicable shape. The initial decline from P=1 to P=2 is due to improved overlap of I/O and processor-time in going from one to two processors. The curve then slowly rises. This rise is due to slightly increased I/O demands in the I/O-bound system. As P increases, the number of partitions increases, and hence the number of blocks that have to be read in twice increases slightly.

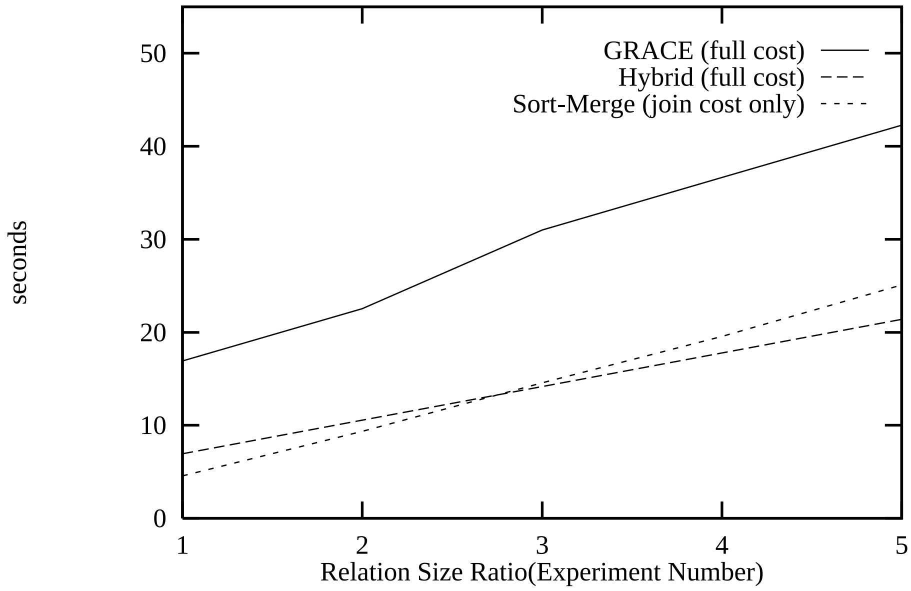


Figure 5.5: RatioVaryRealIO Series Pre-Sorted Times

I/O. (The full relations also have to be read and written during the Sort Phase, but here the I/O is sequential, and hence approximately twice as fast as random I/O.) Thus, omitting the Sort Phase does not result in tremendous time-saving for DPSMJ.

Second, with relatively small relations, the Hybrid algorithm managed to avoid a considerable amount of I/O by storing one hash-bucket in memory, rather than writing it out as a temporary result, only to read it in later. The DPSMJ algorithm has no similar feature; all of the temporary results (i.e. the sorted runs) are written out to disk during the Sort Phase. This reduction in I/O-activity is particularly significant, since the I/O saved is costly random I/O, rather than cheaper sequential I/O. The comparative I/O costs by phase are shown in figure 5.6.

When relations are large and pre-sorted, DPSMJ is better than the hash-based algorithms, since it uses less I/O. When relations are small, DPSMJ's reading of the whole relations at random turns out to be more expensive than Hybrid's reading of the whole relations sequentially, plus the random writing and sequential reading

Phase	GRACE	DPSMJ
Sort/Partition	whole relation sequential read	whole relation sequential read
	partial relation random write	whole relation sequential write
Join	partial relation sequential read	whole relation random read

Figure 5.6: I/O Activity, by Phase

of the partial relations. Thus, Hybrid manages to beat DPSMJ, when relations are small, despite the availability of sorted and sampled runs.

Processor-Bound Experiments

Figures 5.7 and 5.8 compare the performance of the DPSMJ algorithm, assuming the presence of sorted and sampled runs, against the performance of the best hash-based join algorithm in cases where the system is processor-bound. For the I/O-less experiments ProcessorVaryNoIO and MemoryVaryNoIO, Hashed Loops turned out to be the best join algorithm, as was discussed in section 4.2.2.

In an processor-bound system where sorted and sampled relations are available, using DPSMJ is far superior to using even the best hash-based join-algorithm. In such circumstances, the only cost for the DPSMJ algorithm is the actual joining cost for scanning down the sorted relations. By comparison, the hash-based algorithms have to do all the work of hashing, inserting tuples, and probing hash-tables. Since all the work of sorting has already been done for it, DPSMJ manages to finish far more quickly than the hash-based algorithm.

5.3.2 Pre-Sorting versus Pre-Hashing

The comparisons made in section 5.3.1 are not really fair, since they assume the presence of a special ordering (i.e. sorted order) that can be used by one algorithm (DPSMJ), but not the others. Although sorted order has intuitive appeal, it is an artificial ordering, which must be maintained (in the case of base relations) or produced

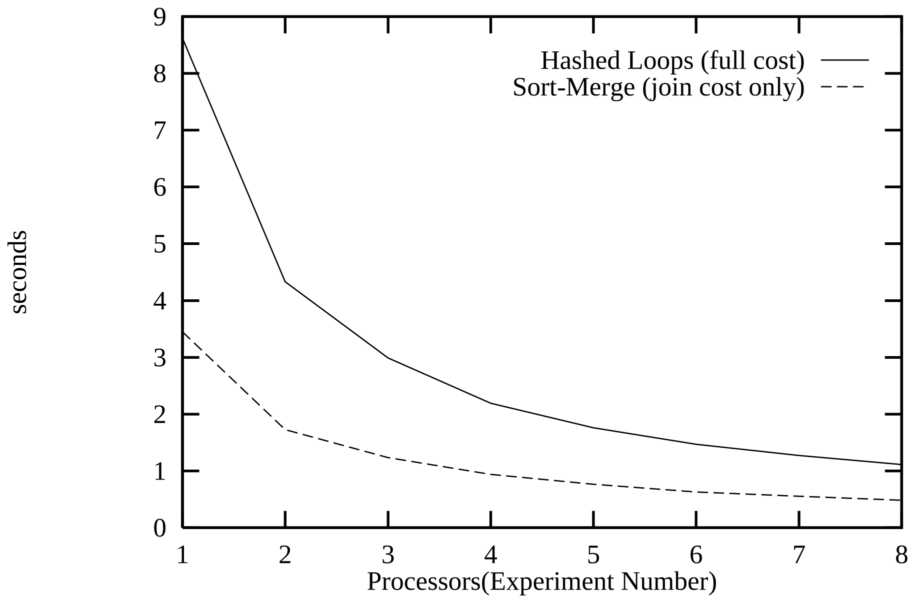


Figure 5.7: ProcessorVaryNoIO Series Pre-Sorted Times

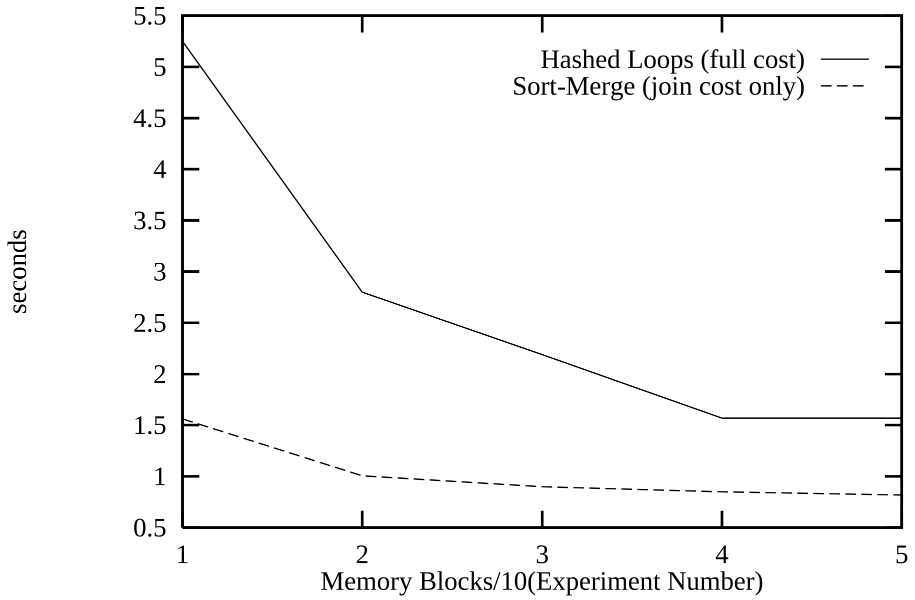


Figure 5.8: MemoryVaryNoIO Series Pre-Sorted Times

(in the case of intermediate results) at significant cost. This cost of maintenance is high because sorted order is very precise and restrictive; a looser ordering (such as hashing) would no doubt be cheaper to maintain, since there is more choice in how to arrange and place each tuple. Thus, if it makes sense to consider pre-sorted relations, it makes sense to consider pre-hashed relations. Providing hash-based algorithms with pre-hashed relations is equivalent to providing DPSMJ with pre-sorted and pre-sampled relations. This is a fairer basis for comparison than that used in section 5.3.1.

Of the three hash-based join algorithms discussed in section 1.1.2, only GRACE can really be modified to take advantage of hash-based pre-partitioning. The algorithm Hashed Loops makes no use use of partitioning, and in the case of pre-partitioned relation, Hybrid has no opportunity to exploit its one difference from GRACE, namely the fact that one hash-bucket is stored in memory rather than being written to disk. Since all the pre-formed hash buckets are initially on disk, only the GRACE algorithm can be performed, and only its Join phase needs to be done.

I/O-Bound Experiments

Figures 5.9 and 5.10 illustrate join costs in I/O-bound cases where pre-sorted and pre-hashed relations are available. The DPSMJ results are from the actual experiments in chapter 4; the GRACE results are based on the Martin et al. models for the GRACE Join Phase.

The figures are again quite clear. Hash-based joins with pre-hashed relations are far cheaper than DPSMJ with pre-sorted relations. This may be attributed to the GRACE algorithm's use of sequential I/O during the join phases, whereas the DPSMJ algorithm uses random I/O. In fact, the only cases where the sort-based algorithm has an advantage (RatioVaryRealIO-1 and RatioVaryRealIO-2) feature tiny relations. In such cases, the difference between sequential and random I/O is modest, since disk arm movement is limited even in the case of random I/O.

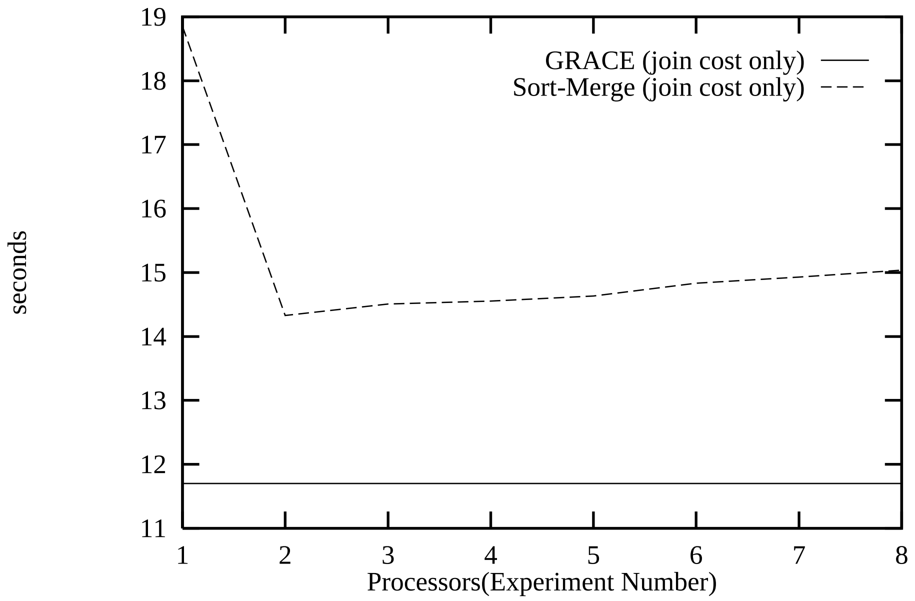


Figure 5.9: ProcessorVaryRealIO Series Pre-Sorted and Pre-Hashed Times

Processor-Bound Experiments

Figures 5.11 and 5.12 illustrate the performance of DPSMJ and GRACE when pre-sorted and pre-hashed relations are available. Again, these results are based on the actual measured times for the Join Phase, in the case of DPSMJ, and the estimated Processor cost for the Join Phase, based on the Martin et al. models, in the case of GRACE.

The results show that using hash-based joins is cheaper in all cases. This is actually somewhat surprising, since the forming and probing of hash-tables that occurs during the join phase of GRACE is quite complicated compared to the table-scanning that occurs in the Join Phase of DPSMJ. Of course, in DPSMJ the cost may be incurred in merging together the runs comprising each relation, and this may be expensive compared to the hashing and probing needed for GRACE. In any case, although the results are comparable, GRACE is superior.

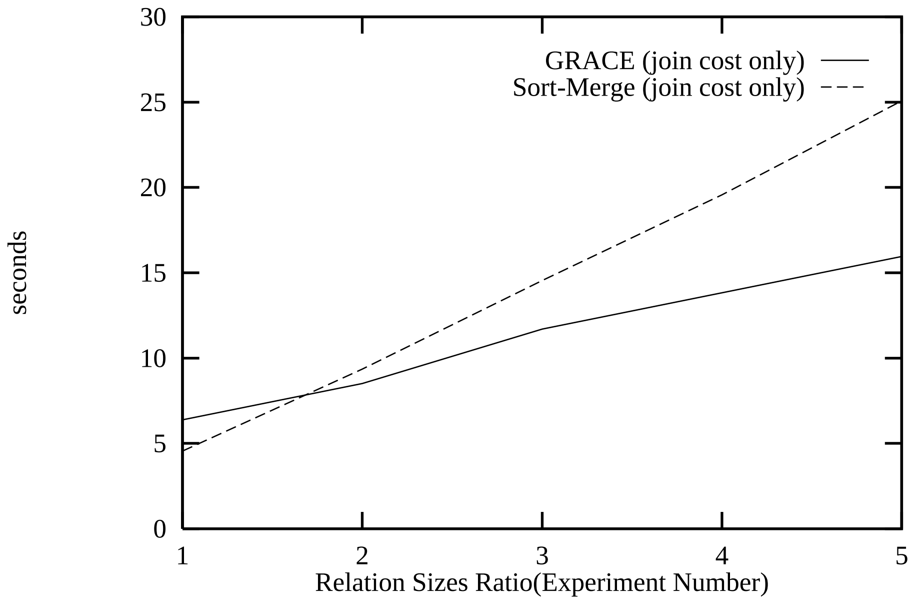


Figure 5.10: RatioVaryRealIO Series Pre-Sorted and Pre-Hashed Times

5.3.3 Conclusions Regarding Pre-Sorting

This section has considered the issue of pre-sorted relations, and how much DPSMJ is aided by their availability. The equivalent for hash-based join methods, pre-hashing, has also been considered. The performance of these cases has been compared for both processor- and I/O-bound systems. The results were favourable to DPSMJ in two situations. In both cases, the relations were pre-sorted but not pre-hashed. First, DPSMJ proved superior in I/O-bound experiments, with large relations. Second, DPSMJ proved superior in processor-bound experiments, with relations of any size. In all other cases, hash-based join algorithms proved superior. DPSMJ suffers principally from its use of random I/O in the join phase.

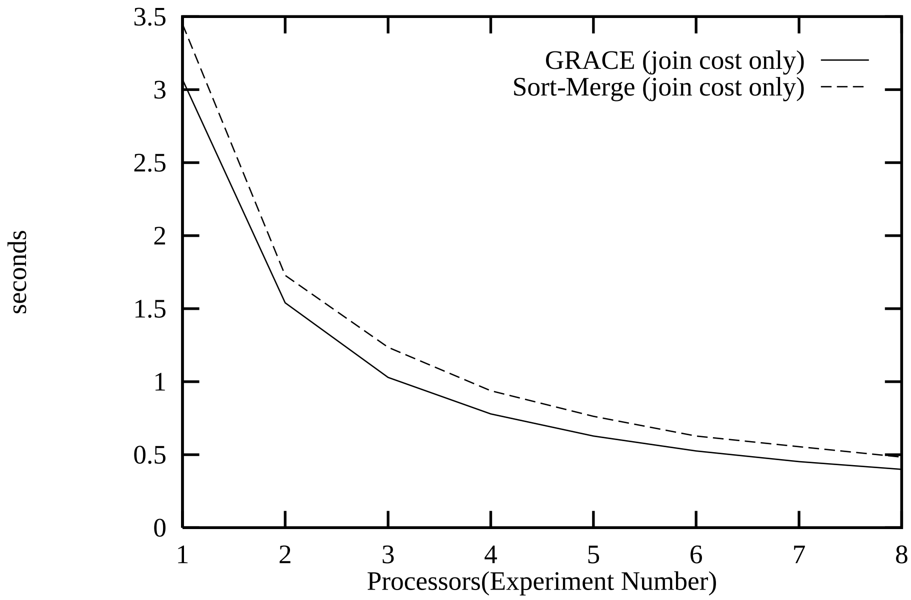


Figure 5.11: ProcessorVaryNoIO Series Pre-Sorted and Pre-Hashed Times

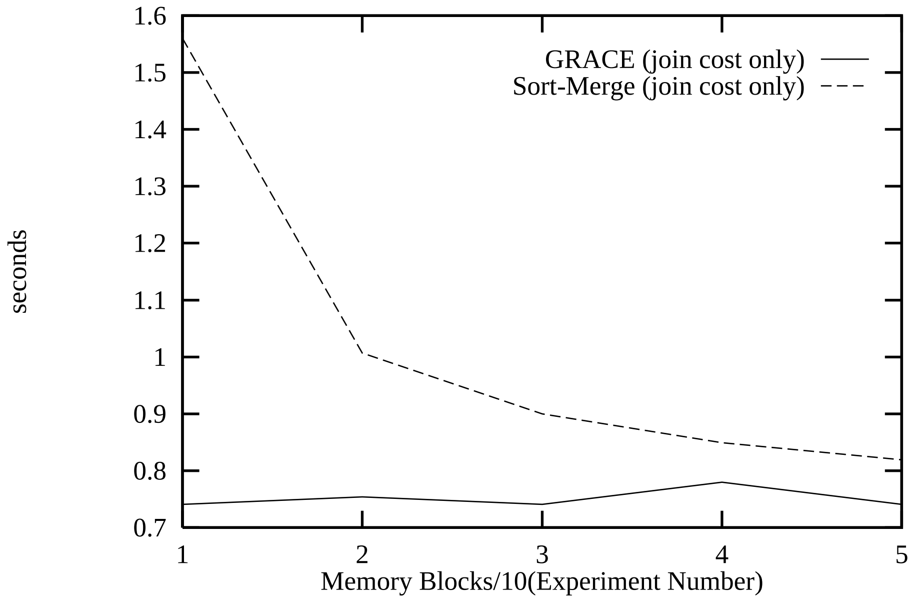


Figure 5.12: MemoryVaryNoIO Series Pre-Sorted and Pre-Hashed Times

Chapter 6

Conclusions

6.1 The Algorithm

This thesis has considered the Domain-Partitioned Sort-Merge Join algorithm (DPSMJ), a parallel version of the classic sort-merge join algorithm for relational equi-joins. The algorithm has been discussed in terms of a shared-memory model of parallelism, with uniform memory access times. Full classic sort-merge join consists of two phases: first, the relations are sorted into runs, and second, the runs are merged and joined. The DPSMJ algorithm parallelizes both the run-forming and the joining phases.

This thesis has considered two versions of parallel run-forming. The first, heap-based run-forming, is a parallel version of replacement selection. All processors access a single global heap, entering and removing tuples. The second, in-memory run-forming, reads in a memory load of tuples, sorts them internally, and writes them out as a sorted run.

In both versions, sampling data regarding the distribution of join-attribute values in the runs is gathered during run-forming. This data is used to partition the workload of the joining phase. The partitioning is based on join-attribute values; each partition corresponds to a range of such values. Each partition is processed by one processor, which simultaneously reads blocks from all the runs, merging the blocks from each of the runs into the comprehensive sorted order. As this order is generated, the processor joins the two relations, producing the tuples of the result for the partition range in question.

6.2 Experiments Performed

Five series of experiments were performed to determine the actual performance of the implementations of the algorithm. The five series varied a wide range of parameters, including processor counts, memory size, and relation sizes. All experiments were performed using both in-memory and heap-based run-forming. The models for three hash-based join algorithms were used to estimate the performance of the algorithms in the experiment cases. These estimates were the basis for comparisons between the performance of DPSMJ and the hash-based algorithms.

The first three series(ProcessorVaryRealIO, SizeVaryRealIO, and RatioVaryRealIO) were carried out with real I/O from disk to disk. These experiments were representative of execution environments when the algorithm would be I/O-bound. The last two experiment series (ProcessorVaryNoIO, and MemoryVaryNoIO) were done with I/O simulated in memory. These experiments simulated processor-bound environments. The main weakness of the experiments was that the relations and memory sizes used were rather small. This was particularly true in the I/O-less experiments, but there it was unavoidable, since the entire relations had to be stored in memory.

6.3 The Models and their Performance

The performance of the various phases, versions, and modules of DPSMJ was studied in this thesis. Explicit models were developed for some of the more important modules, and data was gathered for implicit interpolation models of the other components. The implicit and explicit model were integrated into a single performance prediction model, which was applied in all the experiment cases described above.

The aggregate model generally performed well; the results were very poor in only one case: I/O-less experiments with heap-based run forming. In all other cases, the performance prediction was generally accurate to within 10%, with a few outliers in extreme cases, all with errors less than 20%. I/O costs were predicted well in all cases. The predictions for the processor costs of the various modules were of mixed quality, but only those for the heap-based run-forming and the join processing were truly poor. The poor prediction of the processor-costs of heap-based run-forming was responsible for the poor performance of the aggregate model in the case of the I/O-less experiments when heap-based run-forming was used. In these cases, the processor cost of the heap-based run-forming was the dominant factor. Although the processor costs of the join were also predicted poorly, the costs were too low to influence the aggregate model significantly.

6.4 How to Form Runs

This thesis has considered two separate methods of forming sorted runs for use in the Join Phase. The first was a parallel version of replacement selection with a global heap. The heap was implemented with a top-down substitute operation, and used a spin lock on every element to maintain heap coherence. To minimize tuple movement, the actual tuples were not moved about in the heap; they were kept in a separate tuple buffer. Heap-based run-forming had full overlap of I/O and processor times, and produced runs on average twice as long as the memory used. These double-length runs were predicted by the analysis of Knuth [10], and confirmed in experiments.

The second version of run-forming was in-memory run-forming. Here, a memory load of tuples was first read into memory. They were then sorted, and written out as a memory-sized sorted run. To minimize tuple movement, the sorting and movement of tuples were separated. First, the sorted order was derived using an array of pointers that were sorted in parallel. Then, by considering the permutation that was needed to put the tuples in the correct order, the tuples could be moved into their correct positions with approximately one movement each. The permutation algorithm was parallel, and operated by factoring the permutation into its cycles. In-memory run-forming had partial I/O-overlap: reading and writing occurred simultaneously, but the I/O system was inactive during sorting and permuting.

From the experiments, it is clear that heap-based run-forming uses far more processor time than in-memory run-forming, and also has lower speedup as the processor count increases. The superiority of in-memory run-forming is particularly clear in processor-bound cases. In I/O-bound cases with high processor counts, heap-based run-forming can equal , or perhaps even outperform, in-memory run-forming because of its full I/O-processor time overlap. It is also possible to contrive cases where heap-based run-forming's ability to produce runs on average twice as long as memory is decisive. Nonetheless, the high processor cost of heap-processing makes in-memory run forming the superior choice in most situations.

The high processor cost of heap-based run-forming is due to a number of factors. The burden of locking every element in the heap to maintain consistency is 15-30% of the processing time. The rest is roughly evenly divided between the tuple copying and the actual heap adjustment. Adjusting the heap after insertion is very complex, and takes a great deal of time. A high cost cost of tuple copying is incurred by copying every processed tuple twice. In run-forming by in-memory sorting, by contrast, every tuple is copied roughly once.

Both versions of run-forming included measures to minimize tuple movement in memory. Heap-based run-forming used a tuple buffer to keep actual tuples out of the heap. In-memory run-forming used a separate permutation sub-phase to keep from having to move tuples about more than necessary.

6.5 The Verdict on the Algorithm

The DPSMJ algorithm never managed to perform joins more quickly than the hash-based join algorithm in the experiments. Based on the experiments and the models, it is safe to conclude that DPSMJ will always be outperformed by the hash-based algorithms when joining unordered relations on disk. There are, however, cases where DPSMJ can do well, when the relations to be joined are in sorted order [1] on the join attribute.

When the system is processor-bound, DPSMJ with pre-sorted relations is always faster than the hash-based algorithms. The cost to do the hashing is far too high to have any chance against the quick joining routines of DPSMJ when the processors are the bottleneck. When the system is I/O-bound, DPSMJ with pre-sorted relations is faster than the hash-based algorithms when relations are large. Interestingly, the Hybrid hash-based algorithm manages to defeat DPSMJ when relations are relatively small, even when DPSMJ has access to pre-sorted relations. Hybrid does this by retaining a substantial portion of the relations in memory, thereby reducing its I/O demand, and its random I/O demand in particular. DPSMJ, on the other hand, does random I/O of the full relations during the joining. The advantage of Hybrid vanishes as relations get larger, as the portion of the relations stored in memory becomes insignificant.

6.5.1 Four Weaknesses of the Algorithm

The investigation of Domain-Partitioned Sort-Merge Join in this thesis has uncovered four basic reasons for DPSMJ's poor performance compared to hash-based join algorithms:

(1.) DPSMJ uses too many buffers. The fact that every processor active during

[1] As noted in the text, the relations must be not only pre-sorted, but also pre-sampled. Maintaining proper pre-sampling information can be done very easily when maintaining sorted order.

the join phase needs at least one (and possibly two) buffers for every sorted run of the relations restricts the range of application of the algorithm.

(2.) DPSMJ does not retain any portion of the relations in memory between the sorting phase and the joining phase. Instead, all the temporary results are written to disk. In contrast, the Hybrid hash-based join algorithm stores part of the relations in memory during the partitioning, thus avoiding some costly I/O. This characteristic is particularly significant when relations are relatively small and the system is I/O-bound. There is no clear way to extend the algorithm to store part of its sorted runs in memory.

(3.) DPSMJ does more work during the initial partitioning (sorting) phase than the hashing join algorithms, and this effort is not rewarded by higher performance in the joining phase. This is because sorting is far more expensive than hashing. The difference in costs becomes particularly significant in processor-bound systems.

(4.) DPSMJ uses random I/O to read in blocks during the joining phase. Random I/O is more expensive than sequential I/O (roughly twice as expensive, in the experiments). This problem is significant when relations are relatively small, and nested loops-algorithms can do well by doing only sequential I/O.

All four of these problems are inherent in the algorithm and they do not have any clear remedies. They are not merely implementation defects.

6.5.2 Agreement with Past Research

Most of the past work cited in section 1.1.1 is consistent with the results of this thesis. However, the basic conclusion regarding the relative performance of sort-merge parallel join and hash-based parallel join differs from the conclusion of Valduriez and Gardarin [21]. They found sort-merge join to be the superior algorithm for large relations. This thesis never found sort-merge join to be superior for unordered relations. This difference is due to the different ranges of memory and relation sizes considered in this thesis and in the Valduriez and Gardarin article.

This thesis considered only relations that are small enough to be processed with a single merge stage in the sort-merge algorithm, and which did not cause bucket overflow in the GRACE algorithm. The relation-size restrictions are discussed in section 5.1.1. This permitted linear processing costs for the two algorithms. Given modern memory sizes, this is not unduly restrictive: assuming a memory of 128 MB, and a block size of 8 K, the largest outer relation that can be processed by GRACE is 1 Terabyte, according to equation 5.3. This is sufficient for most situations.

The memory sizes considered by Valduriez and Gardarin were much smaller, however, necessitating multi-level merges for the parallel sort-merge algorithm. The sort-merge algorithm dealt with this well, and the costs increased as $n \log n$ for relations of size n. In the same situations, the hash-join algorithm they considered managed poorly. It had a fixed-size hash-table that eventually saturated, causing long searches through collision-resolution chains. Thus, the costs for large relations varied as n^2 for relations of size n. This accounted for the superiority of parallel sort-merge join for large relations.

It would, of course, be possible to extend the DPSMJ algorithm to include multiple-level merges. This would allow it to join very large relations faster than the published GRACE algorithm. This is because the GRACE algorithm's cost becomes quadratic when hash-buckets exceed the size of memory. The quadratic cost occurs because GRACE processes over-sized buckets using Hashed Loops (i.e. a quadratic join method). However, GRACE could also be modified to perform multi-level hashing by analogy with multi-level merging. This modification might allow it to join faster than DPSMJ. Given the size of modern main memories, this would be relevant only for very large relations (of multiple terabytes or more), and it was therefore not studied in this thesis.

6.6 Applicability of This Research

6.6.1 Range of Applicability of the Algorithm

In considering the range of applicability of the results of this thesis, two issues must be resolved. First, one must determine the range of machines over which the DPSMJ algorithm can be used with any degree of efficiency. Second, one must consider whether the architectural differences within that range of machines are sufficient to change or invalidate the conclusions of this thesis.

DPSMJ is suitable for shared-memory or, in a pinch, shared-disks architectures; it is not suitable for shared-nothing architectures. This constraint is due to the nature of the Join Phase. Because every Joiner needs to access every run, there does not seem to be any locality of reference to the stream of block requests from the Joiners. Thus, the runs must be equally accessible to all processors, necessitating at least shared disks. The Joiners do not share any significant amount of memory, beyond minor coordination traffic which can be handled with explicit messages. Thus, the Join Phase does not require shared memory. The Calculation Phase is too brief and its products too small to affect the choice of architectures for the algorithm. For the Sort Phase, both of the run-forming methods in this thesis really require shared memory, but they could be adapted for use on shared-disks architectures. Without shared memory, each processor would work independently forming runs in its own memory, rather than working together with all the other processors on a large run in the shared memory. Accordingly, runs would be much shorter without shared memory, limiting the utility of the algorithm. Because of the wide-ranging memory accesses of each of the processors during the joining phase with shared memory, any significant non-uniform memory access times (NUMA) would require restricting the run-forming methods to function as if shared memory were not available (i.e. by having each processor work on run forming in its local memory only). Thus, DPSMJ is best suited for symmetrical shared-memory multiprocessors, but can be used on shared-disks platforms to some extents.

Having established the range of architectures for which DPSMJ is suitable, we must next consider what changes within that range could change the conclusions of this thesis. One of the principal results of this thesis is the inferiority of DPSMJ to hash-based algorithms when relations are unsorted. In section 6.5.1, four weaknesses were discussed, which accounted for the inferiority of DPSMJ. Only one of these (number 3) might possibly change with changes in architectures. The weaknesses of excessive buffer use, lack of retention of temporary results in memory, and the use of random I/O would remain for any architecture on which DPSMJ might be used. The only weakness that might disappear with changes in processor-, cache-, or memory-architecture is the third weakness, regarding the expense of sorting rather than hashing. It might be possible to find a machine on which sorting could compete in speed with hashing. However, given the inherent simplicity of sorting, this is a rather remote possibility.

The issue of heap-based run forming compared to in-memory run-forming is somewhat similar. Here, the issues are again processor- and memory-architecture. On the tested system, in-memory run-forming was found to be better largely due to its lower processing costs. As with the relative performance of hashing and sorting, it might be possible to find machines where heap-processing would be faster than in-memory sorting. However, given the inherent complexity of heap-processing, this possibility must be considered remote.

The experiments and analysis of this thesis considered both I/O-bound and processor-bound systems, and the conclusions were generally consistent for both of these extremes. The analysis also includes four weaknesses of the algorithm which seem to be constant across all operating environments. This section has determined that the algorithm is suitable for only a fairly narrow set of hardware platforms, which are very much like the system used for testing. Based on all of this, it seems safe to conclude that the results of this thesis apply universally.

6.6.2 Applicability to Larger Relations

Most of the experiments in this thesis concerned cases where the relations were only a small multiple larger than memory. Larger relations were really considered only in section 5.3, in addressing joins with pre-sorted relations. However, there are realistic cases where unordered relations to be processed are very much larger than memory, and the applicability of the algorithm and the validity of the results of this thesis in such situations should be considered. To begin with, an extreme difference between the size of memory and the sizes of the relations is not possible for the DPSMJ algorithm. This is due to the bound on relation sizes presented in section 5.1.1. Second, the results comparing heap-based and in-memory run forming do not change with even a dramatic increase in relation sizes relative to memory.

A more complicated issue is the relative performance of DPSMJ and the hash-based algorithms with larger relation sizes. When relations are large compared to memory, there is essentially only one hash-based algorithm, namely GRACE. In such situations, Hashed Loops is hopelessly inadequate due to its quadratic behaviour. Also, Hybrid's one difference from GRACE, the storing of one hash-bucket in memory, becomes insignificant when that single hash-bucket is not a significant portion of the relations being joined. Thus, only GRACE remains.

Unfortunately, DPSMJ performs quite poorly compared to GRACE. DPSMJ and GRACE have similar I/O-behaviours: every tuple moves three times between disk and memory. In each algorithm, one of these transitions uses random I/O and the others use sequential I/O. Of course, since DPSMJ would be competing against Hybrid rather than true GRACE, the hash-based algorithm would do slightly less I/O. Thus, in I/O-bound cases with very large relations compared to memory, the performance of DPSMJ would be comparable to, but slightly worse, than the hash-based algorithm. In processor-bound cases with very large relations, DPSMJ would be decisively inferior to the hash-based algorithms. Here, processing costs are the only issue, and the results in the thesis show DPSMJ's processing costs to be far higher than those of GRACE.

Thus, for relations that are very large compared to memory, DPSMJ is at best almost as good as the hash-based join methods, and at worst dramatically slower. This is essentially equivalent to the results from the experiments conducted with relations that were small relative to the memory used. Thus, the conclusions of this thesis may be considered valid for all memory and relation sizes.

6.7 Further Research

This thesis has raised three issues that should be investigated further:

- double-buffered in-memory run-forming,
- properties of the parallel permutation algorithm, and
- DPSMJ with indices.

The principal disadvantage of in-memory run-forming relative to heap-based run-forming is the fact that I/O and processor times do not overlap. This could be solved by dividing the memory into two tuple-arrays, with one being sorted and permuted, while the other is being read into and written out of. Since this would permit complete overlap of I/O and processor time, it is worth investigating, although it does have the disadvantage that the runs produced would be only a quarter the size of the runs produced by heap-based run-forming.

The parallel permutation algorithm used in the in-memory run-forming is original to this thesis. Further investigation of this algorithm would be worthwhile.

This thesis considered the applicability of DPSMJ to joins with indices, and found a case where it might be useful: Index-Hybrid joins can be decomposed into two DPSMJ operations. This thesis discussed the possibility, but included no concrete results. This topic is worthy of further study.

6.8 Hindsight

The basic result of this thesis is negative. The Domain-Partitioned Sort-Merge Join algorithm is not competitive in most join situations; in fact, it is useful in only fairly narrowly defined cases. This being the case, after the amount of effort that has been devoted to this study, it is worthwhile to consider whether the basic result could have been arrived at more economically.

This thesis has identified four major weaknesses of DPSMJ:

- excessive buffer use,
- lack of memory retention of relations,
- use of random I/O, and
- excessive sorting cost hashing.

The first of these problems, excessive buffer use, became apparent early in the project. It showed up as soon as experiments were being planned, and was quantified in short order. It was always acknowledged as a problem. The problems of lack of memory retention and use of random I/O became apparent during the design of the algorithm. However, the extent to which they hampered performance was not clear until the experiments were run. This was largely because modeling was not begun until well into the implementation of the algorithm. The extent of these two problems could have been discovered by simple I/O modeling before implementation. Essentially the entire cause of DPSMJ's weakness in I/O-bound systems could have been found without actual implementation.

Only the final problem, sorting cost, really required implementation. It was intuitively clear that sorting, whether by use of a heap or in an array was expensive, but it was not clear whether this expense would be compensated for in the joining phase. It was also not clear how much of the tuple movement cost could be avoided.

Thus, to quantify the performance of DPSMJ compared to the hash-based algorithms definitely required implementation.

Since I/O costs are easy to model, the performance of DPSMJ in I/O-bound cases would have been easy to discover without implementation. Processor costs are much harder to model. Thus, to determine the performance of the algorithm in processor-bound situations required actual implementation.

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*, pp. 138-145. Addison-Wesley, Reading, Massachusetts, 1983.

[2] P.A. Buhr and R.A. Stroobosscher, "The μSystem: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running Unix", *Software: Practice and Experience*, 20(9), pp. 929-963, September 1990.

[3] B.W. Char et al. *First Leaves for the Macintosh: A Tutorial Introduction to Maple*, Brooks/Cole Publishing Company, Pacific Grove, California, 1989.

[4] J. Cheng et al., "An Efficient Hybrid Join Algorithm: A DB2 Prototype" in *Proceedings of the IEEE Conference on Data Engineering*, p. 171, 1991.

[5] D. Comer, "The Ubiquitous B-Tree", *ACM Computing Surveys*, 11(2), pp. 121-137, June 1979.

[6] D. J. DeWitt and R. Gerber, "Multiprocessor Hash-Based Algorithms", *Proc. VLDB 85*, Stockholm, Sweden, 1985, pp. 151-164.

[7] D. J. DeWitt, R. H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood, "Implementation Techniques for Main-Memory Database Systems", *Proc. SIGMOD 84*, pp. 1-8.

[8] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys*, 25(2), pp. 73-170, June 1993.

[9] G. Graefe and S.S. Thakkar, "Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor", *Software: Practice and Experience*, 22(7), pp. 495-517, July 1992.

[10] D.E. Knuth. *The Art of Computer Programming*, Vol. 3, pp. 251-262. Addison-Wesley, Reading, Massachusetts, 1973.

[11] H. Li and K.C. Sevcik, "Parallel Sorting by Over-Partitioning", *Proc. 6th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA '94)*, Cape May, New Jersey (June 1994), pp. 46-56.

[12] H. Lu, K.-L. Tan, M.-C. Shan, "Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory", *Proc. VLDB 90*, Brisbane, Australia, 1990, pp. 198-209.

[13] T.P. Martin, P.-Å. Larson, and V. Deshpande, "Parallel Hash-Based Join Algorithms for a Shared-Everything Environment", University of Waterloo Computer Science Research Report CS-91-18, May 1991.

[14] P. Mishra and M.H. Eich, "Join Processing in Relational Databases", *ACM Computing Surveys*, 24(1), pp. 63-113, March 1992.

[15] G. Z. Qadah and K. B. Irani, "The Join Algorithms on a Shared-Memory Multiprocessor Database Machine", *IEEE Trans. Soft. Eng.*, 14(11), pp. 1668-1683, Nov. 1988.

[16] N.N. Rao and V. Kumar, "Concurrent Access of Priority Queues", *IEEE Transactions on Computers*, 37(12), pp. 1657-1665, Dec. 1988.

[17] J. P. Richardson, H. Lu, and K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms", *Proc. SIGMOD 87*, San Francisco, California, pp. 399-409, May 1987.

[18] B. Salzberg, "Merging Sorted Runs Using Large Main Memory", *Acta Informatica*, 27, pp. 195-215, 1989.

[19] Sequent Computer Systems, *Sequent Technical Summary*, 1987.

[20] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM TODS*, 11(3), pp. 239-264, Sept. 1986.

[21] P. Valduriez and G. Gardarin, "Join and Semijoin Algorithms for a Multiporcessor Database Machine", *ACM TODS*, 9(1), pp. 133-161, March 1984.

[22] J. L. Wolf, D. M. Dias, and P.S. Yu, "An Effective Algorithm for Parallelizing Sort Merge Joins in the Presence of Data Skew", *Proc. Second Int. Symp. on Databased in Parallel and Distributed Systems*, Dublin, Ireland, 1990.

[23] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek, "An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew", *Proc. Seventh International Conference on Data Engineering*, Kobe, Japan, 1991.

[24] L.Q. Zheng and P.-Å. Larson, "Speeding Up External Mergesort", University of Waterloo Computer Science Research Report CS-92-40.

Appendix A

The Flush Module's Processing Requirements

As can be seen from figure A.1, the processing time required by the flush module when it is operating in the expected parameter range (B=256) is modest, typically only a few microseconds per tuple. Compared to the processing times for the sort and permute modules, the processing time for the flush module is small enough to ignore. Given the slight impact of this module, only a cursory exploration of its performance characteristics is included here.

A few features of the table bear commenting on. First, note that the write cycles are significantly cheaper than either read/write or read cycles. This is attributable to the index-filling done when tuples are read in; the index entry corresponding to each tuple is set by the flush module. Second, the time per tuple generally rises as the block size declines. This is to be expected, since the operation is block-wise. The overhead of processing each block is spread over fewer tuples as the block size decreases. Another reason for this trend might be memory contention, which increases as more processors compete to access a smaller amount of memory. Third, per-tuple costs decrease as the number of blocks in memory increases. This decline is attributable to decreasing contention among the processors, due to the increased work-space.

P=4	M=20	M=40	M=60	M=80	M=100	M=200	M=300	M=400	M=500
read									
B=256	2.099	2.050	2.008	1.979	1.965	1.906	1.887	1.880	1.879
B=128	2.412	2.295	2.286	2.247	2.242	2.223	2.189	2.186	2.175
B=64	4.456	3.456	3.128	3.005	2.902	2.734	2.620	2.569	2.540
B=32	9.996	6.698	5.627	5.111	4.794	4.131	3.906	3.808	3.723
read/write									
B=256	2.157	2.051	1.992	1.964	1.955	1.889	1.867	1.866	1.859
B=128	2.411	2.292	2.241	2.237	2.238	2.202	2.180	2.175	2.162
B=64	4.418	3.456	3.127	3.009	2.887	2.711	2.633	2.580	2.553
B=32	9.990	6.745	5.690	5.154	4.863	4.164	3.953	3.854	3.770
write									
B=256	0.564	0.506	0.467	0.445	0.434	0.407	0.399	0.396	0.399
B=128	1.296	1.040	0.950	0.908	0.882	0.829	0.813	0.805	0.794
B=64	3.647	2.622	2.276	2.086	1.977	1.751	1.696	1.664	1.636
B=32	9.740	6.529	5.372	4.817	4.516	3.798	3.572	3.477	3.387

Figure A.1: Flush Processing Time (usec/tuple)

Figure A.2 shows the variance of the flush processing time over B and P. For these experiments, the W values were set to the optimal values when I/O is actually performed.

The clearest trend is the read/write and read timing for B=256. In this case, there are four FlushWorker processes active, as per figure 3.11. In this case, the algorithm is able to make use of the available processors, until stagnation at P=4. After that, the processing times actually begin to increase slightly, probably due to cache effects. The case for B=128 (with W=6) is similar; again, the minimum time is at P=W, and the times begin to increase slightly after that. In these cases, the efficiencies at the minima are 93% and 81%, entirely respectable showings. The other two cases, where there are more worker processes than processors (W=12 for B=64, and W=19 for B=32), show continuously decreasing processing times as P increases, since P=W is never reached. The parallel efficiency is very poor, however.

The write cycle costs behave quite differently. Here, in each case, the best performance occurs for P=1. This is probably due to how little the FlushWorker tasks have to do, in the absence of real I/O. The time shown is essentially the time required to synchronize all the processes, and the greater the number of processors available, the more active the processes will be.

M=100	P=1	P=2	P=3	P=4	P=5	P=6	P=7	P=8
read								
B=256	7.309	3.789	2.601	1.957	1.981	1.980	1.988	1.994
B=128	7.068	3.804	2.778	2.235	1.849	1.450	1.459	1.469
B=64	6.935	4.336	3.446	2.893	2.549	2.236	2.291	2.351
B=32	7.777	6.044	5.464	4.778	4.775	4.902	5.071	5.416
read/write								
B=256	7.243	3.753	2.587	1.944	1.969	1.973	1.977	1.978
B=128	7.008	3.799	2.789	2.215	1.864	1.445	1.456	1.459
B=64	6.814	4.351	3.465	2.909	2.548	2.250	2.248	2.297
B=32	7.838	6.335	5.569	4.838	4.777	4.871	5.039	5.380
write								
B=256	0.143	0.335	0.414	0.435	0.443	0.472	0.516	0.559
B=128	0.303	0.698	0.858	0.884	0.898	0.973	1.039	1.160
B=64	0.679	1.495	1.920	1.980	2.024	2.103	2.309	2.577
B=32	1.683	3.899	4.420	4.496	4.610	4.804	5.168	5.809

Figure A.2: Flush Processing Time(microseconds)

Appendix B

The Permuter Process

This is the complete C code for the Permuter process; nothing has been omitted from it. In this code, locks are seized and released with `S_LOCK` and `S_UNLOCK`, respectively. The `uDie(NULL,0)` call terminates the process. The process is started with:

arr The array of tuples to be permuted.

index The array of index entries, `index[i].where = j` implies that `arr[j]` should be moved to `arr[i]`.

control Array of control information, one per array entry; two fields: `guard`(the lock), and `busy`(the boolean). The field `busy` indicates that the entry should *not* be accessed.

nr_elements The number of tuples in the array.

done Boolean indicating that all tuples are in their correct places (shared among all Permuter processes.)

nr_processed The number of correctly placed tuples (shared).

global_guard The (shared) lock guarding `done` and `nr_processed`.

initial_element The first entry to try, when looking for elements to start a cycle.

The internal variable, `start_index` is the earliest element of the chain of claimed elements, `cur_index` is the latest, and `prev_index` the one just before it. The variable `new_index` is the element being considered for inclusion in the chain, and length is the current number of elements in the chain. All are used during the process of finding a cycle. The variables `cur_source` and `cur_dest` are used during the actual shifting of the tuples of a cycle. The variable `temp_tuple` is used to store the first tuple in the chain.

```
void Permuter( Tuple *arr,
               Sorter *index,
               Busy *control,
               unsigned nr_elements,
               bool *done,
               unsigned *nr_processed,
               slock_t *global_guard,
               unsigned initial_element ) {
  unsigned cur_index, prev_index, start_index, new_index, length;
  unsigned cur_source, cur_dest, temp, i;
  Tuple temp_tuple;

  start_index = initial_element;

  /* outer loop */
  while(TRUE) {
    /* look for an unchained element */
    if (*done) {
      uDie(NULL,0);
    } /* end if */
```

```
/* note two-stage check of control[start_index].taken;
   first without seizing the guard lock, and then a second time
   using the lock, if the element appears to be unclaimed */
if (control[start_index].taken) {
  start_index = (start_index+1)% nr_elements;
  continue;
} /* end if taken */
S_LOCK(&(control[start_index].guard));
if (control[start_index].taken) {
  S_UNLOCK(&(control[start_index].guard));
  start_index = (start_index+1)% nr_elements;
  continue;
} /* end if taken */
control[start_index].taken = TRUE;
S_UNLOCK(&(control[start_index].guard));

/* we now have a start_index */
length = 1;
cur_index = start_index;
/* inner loop */
while(TRUE) {
  new_index = index[cur_index].where;
  if (new_index== start_index) {
    /* we have a cycle, possibly degenerate */
    break;
  } /* end if have cycle */
  S_LOCK( &(control[new_index].guard) );
  if ( control[new_index].taken ) {
```

```
S_UNLOCK( &(control[new_index].guard) );
/* we have a claim, and progress is blocked */
if ( length <= 3) {
  /* abandon the range */
  if (length == 3) {
    S_LOCK(&(control[prev_index].guard));
    control[prev_index].taken = FALSE;
    S_UNLOCK(&(control[prev_index].guard));
  } /* end if length == 3 */
  if (length >= 2) {
    S_LOCK(&(control[start_index].guard));
    control[start_index].taken = FALSE;
    S_UNLOCK(&(control[start_index].guard));
  } /* end if length >= 2 */
  S_LOCK(&(control[cur_index].guard));
  control[cur_index].taken = FALSE;
  S_UNLOCK(&(control[cur_index].guard));
  length = 0;
} else {
  /* clip cycle */
  new_index = index[start_index].where;
  /* exchange two array elements */
  temp_tuple = arr[cur_index];
  arr[cur_index] = arr[new_index];
  arr[new_index] = temp_tuple;
  length -= 2;
  /* release the first and last elements of the chain */
  /* exchange two array elements */
```

```
      temp = index[prev_index].where;
      index[prev_index].where = index[start_index].where;
      index[start_index].where = temp;
      S_LOCK(&(control[cur_index].guard));
      control[cur_index].taken = FALSE;
      S_UNLOCK(&(control[cur_index].guard));
      S_LOCK(&(control[start_index].guard));
      control[start_index].taken = FALSE;
      S_UNLOCK(&(control[start_index].guard));
      start_index = prev_index;
    } /* end if length */
    break;
  } else {
    /* expanded claim by one */
    control[new_index].taken = TRUE;
    S_UNLOCK(&(control[new_index].guard));
    prev_index = cur_index;
    cur_index = new_index;
    length++;
  } /* end if */
} /* end while inner */

/* we have a cycle */
if (length > 0 ) {
  S_LOCK(global_guard);
  *nr_processed += length;
  *done = (*nr_processed==nr_elements);
  S_UNLOCK(global_guard);
```

```
      if (length > 1) {
        cur_dest = start_index;
        temp_tuple = arr[start_index];
        while(TRUE){
          cur_source = index[cur_dest].where;
          if (cur_source == start_index) {
            arr[cur_dest] = temp_tuple;
            break;
          } else {
            arr[cur_dest] = arr[cur_source];
            cur_dest = cur_source;
          } /* end if */
        } /* end while */
      } /* end if */
      if (*done) {
        uDie(NULL,0);
      } /* end if */
    } /* end if length */
    start_index = (start_index+1)% nr_elements;
  } /* end while outer loop */
} /* end Permuter */
```

Appendix C

Experiment Results: Proportion Analysis

Proportion analysis has been carried out on all experiment series, to determine what portion of time is spent in each of the execution phases. Some of the results are included in the thesis, the rest appear here.

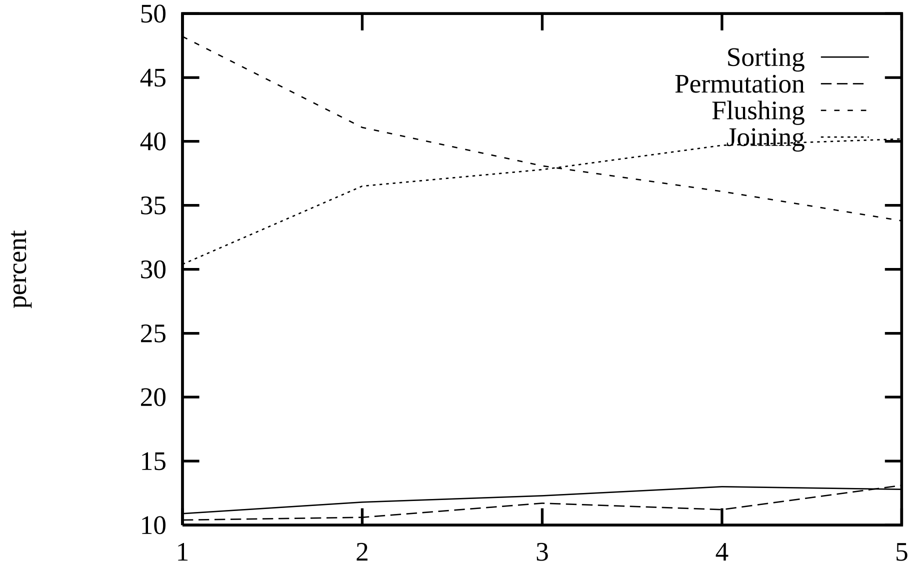


Figure C.1: Division of Time for SizeVaryRealIO Experiment with In-Memory Run-Forming

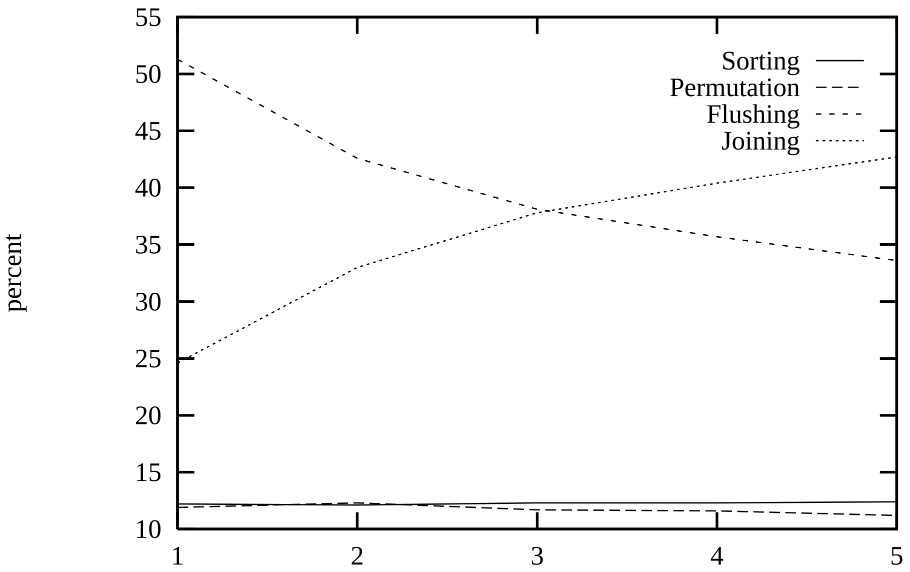


Figure C.2: Division of Time for RatioVaryRealIO Experiment with In-Memory Run-Forming

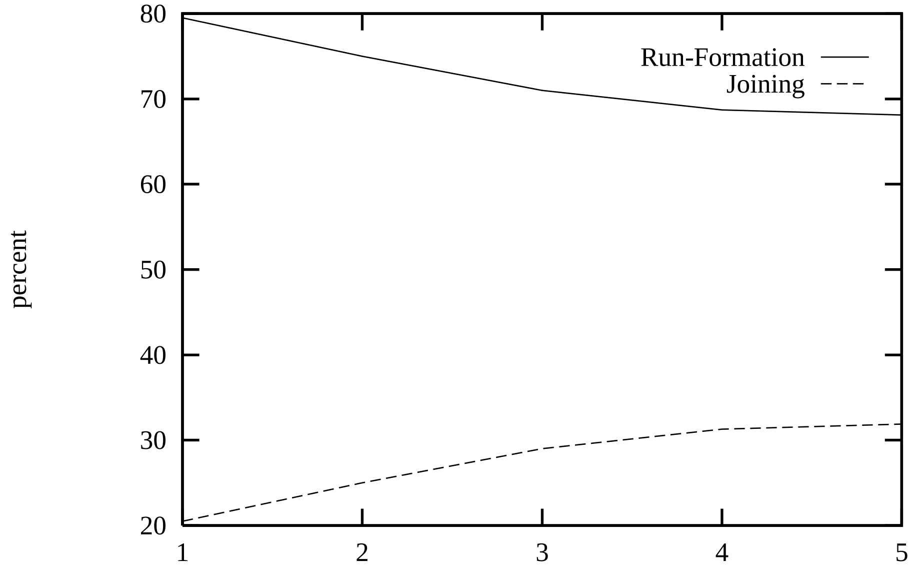


Figure C.3: Division of Time for SizeVaryRealIO Experiment with Heap-Based Run-Forming

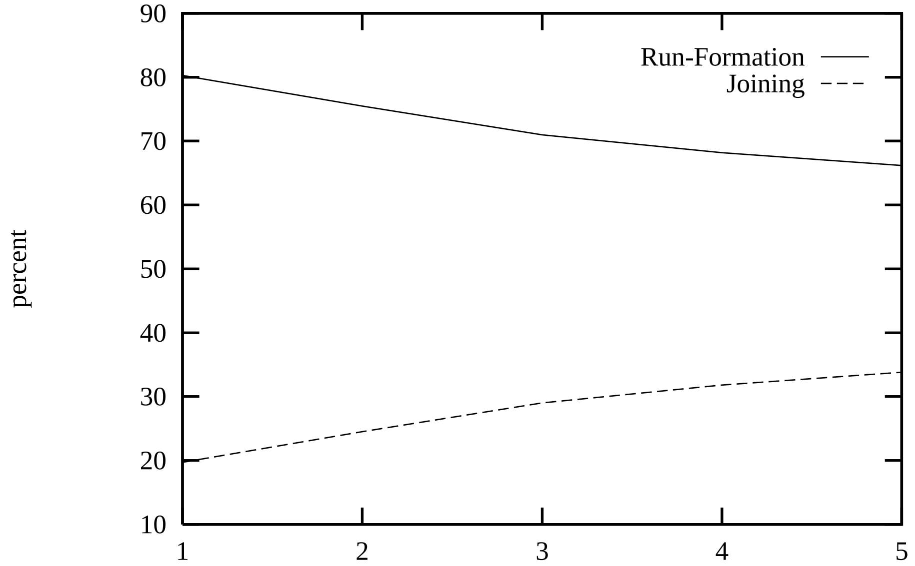


Figure C.4: Division of Time for Gamma Experiment with Heap-Based Run-Forming

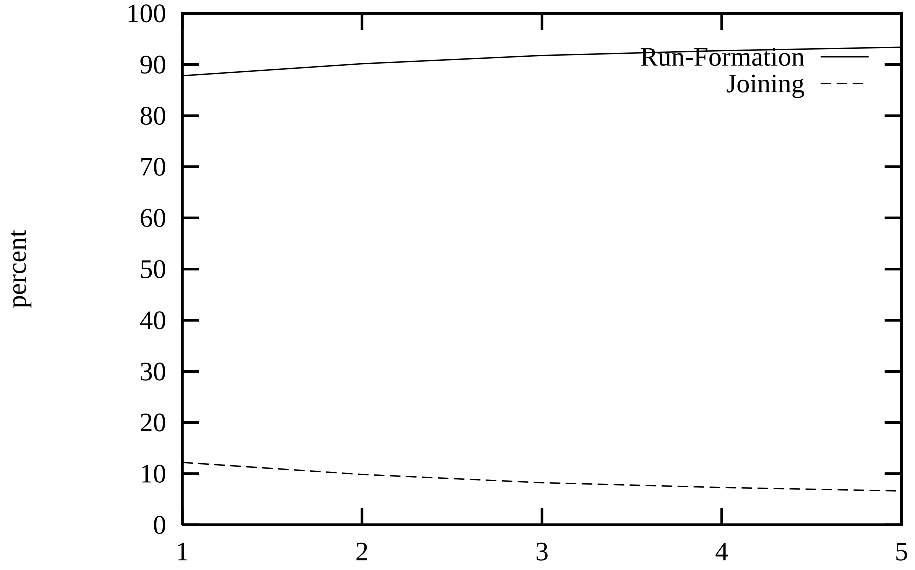


Figure C.5: Division of Time for MemoryVaryNoIO Experiment with Heap-Based Run-Forming

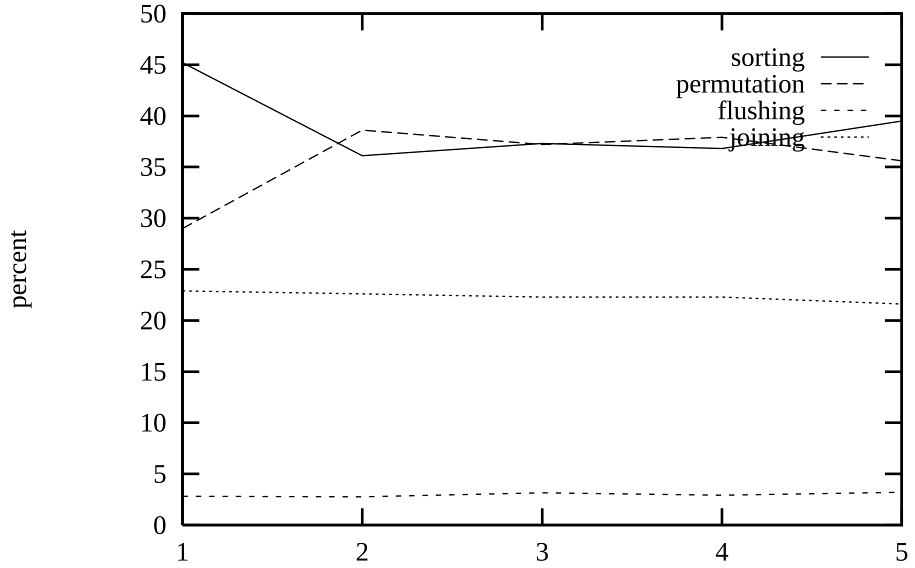


Figure C.6: Division of Time for MemoryVaryNoIO Experiment with In-Memory Run-Forming

Appendix D

Experiments Regarding Heap Processing Costs

Experiments have been conducted to find out whether locking in the heap during heap-based run forming is responsible for the extreme slowness of heap adjustment. The locking was found to account for 15-30% of the processing time. Thus, eliminating the concurrency control in the heap would not dramatically speed up heap-based run forming.

An artificial setup was used to study the costs of heap processing. A proper heap was used for the experiments, and elements were entered into it, with heap adjustment being performed. The degree of heap consistency maintained in the heap varied; in some experiments all of the elements had locks, and in others, only the locks in a few of the top levels of the heap were used. Thus, the heap could be inconsistent in the levels that went unlocked. The heap was initially full, and remained so throughout each experiment.

The copying of tuples into and out of the tuple buffer was simulated by performing two tuple copy operations at random in an array of tuples. The array had twice as many elements as the heap. The experiments were conducted with various H (head height) values, and various P (processor count) values. Each experiment consisted of

	P=1	P=2	P=3	P=4	P=5	P=6	P=7	P=8
H=8	3771	2217	1609	1496	1465	1549	1508	1506
H=10	4560	2574	1822	1523	1438	1424	1460	1477
H=12	5208	2908	2012	1607	1491	1461	1544	1521
H=14	5733	3279	2279	1812	1557	1471	1454	1467
H=16	6341	3589	2456	1926	1756	1797	1582	1779

Figure D.1: Heap Processing Costs(milliseconds), Full Locking (10,000 Tuples)

	P=1	P=2	P=3	P=4	P=5	P=6	P=7	P=8
H=8	3177	1956	1392	1215	1181	1110	1106	1112
H=10	3884	2205	1542	1237	1161	1154	1117	1110
H=12	4284	2416	1677	1308	1207	1127	1089	1094
H=14	4713	2644	1861	1485	1299	1196	1166	1144
H=16	5165	2885	2022	1652	1507	1460	1279	1291

Figure D.2: Heap Processing Costs(milliseconds), Locking Top Element Only (10,000 Tuples)

entering (substituting) 10,000 elements in the heap.

Figure D.1 shows the baseline case. Here, full consistency was maintained, using a lock for every heap element. Figure D.2 shows the other extreme, with only a single lock maintained, associated with the head of the heap. The rest of the heap was left potentially inconsistent. The lack of locking in the heap allowed the processing time to drop some 15-30%. This is a realistic assessment of the burden of locking during the heap adjustment process. The burden generally increased with increasing processor counts.

Figures D.3 and D.4 show the burden of heap adjustment and tuple copying in heap-based run-forming. The experiments in D.3 were run without heap adjustment, and the experiments in D.4 were run without tuple copying. Clearly, both the actual

	P=1	P=2	P=3	P=4	P=5	P=6	P=7	P=8
H=8	1855	1177	1151	1104	1057	1050	1056	1088
H=10	2129	1238	1164	1104	1035	1034	1045	1078
H=12	2241	1248	1171	1119	1025	1023	1034	1066
H=14	2331	1289	1186	1153	1035	1032	1040	1065
H=16	2312	1267	1211	1197	1067	1241	1137	1214

Figure D.3: Heap Processing Costs(milliseconds), No Heap Adjustment (10,000 Tuples)

	P=1	P=2	P=3	P=4	P=5	P=6	P=7	P=8
H=8	2498	1460	1374	1372	1389	1289	1304	1325
H=10	3022	1750	1313	1233	1295	1278	1280	1300
H=12	3527	2063	1454	1348	1365	1350	1293	1320
H=14	4070	2403	1639	1306	1248	1296	1284	1299
H=16	4629	2698	1825	1418	1333	1354	1321	1305

Figure D.4: Heap Processing Costs(milliseconds), No Tuple Copying (10,000 Tuples) heap adjustment and the tuple copying were significant impediments to performance, since both took significant time. At the heap- and tuple-sizes used, tuple copying took slightly less time than heap adjustment. Given these results, in any attempt to improve the performance of heap-based run-forming it would be wise to address both the issues of the complex heap adjustment process, and the required double copying of every tuple.