UNIVERSITY OF CALIFORNIA,
IRVINE

On the Design and Evaluation of a New Order-Based Join Algorithm

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Guangqiang Li

Thesis Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Professor Sharad Mehrotra

2010

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express the deepest appreciation to my committee chair, Professor Michael J. Carey, who has motivated and guided me into the Database world. I have lost most of my passion to study for a long time, but since I started to work with Prof. Carey, the ages when I used to throw myself into work have come back. He is also patient even though I started slow. During the last few weeks of the spring quarter, we worked closely every day, sometimes even chatting at night so as to move my work forward. Without his guidance and persistent help this thesis would not have been possible.

I would like to thank my committee members, Professor Chen Li and Professor Sharad Mehrotra, who are actively involved in the research activities in our ISG group, giving lots of valuable suggestions for our research work.

I would like to thank Dr. Goetz Graefe of HP Laboratories, my “guest committee member”, who has given me such a new and exciting join algorithm to work on. His fruitful industrial experience has given us many insights into how to implement algorithms efficiently.

Another Very Important Person for me is Vinayak Borkar, who has always been willing to help me with all kinds of technical problems. Although he is truly a master of software/algorithm implementation, he is also very nice and easy-going. His generous help has greatly reduced my time to finish the implementation of this join algorithm.

In addition, I would like to thank my wife Yinai Sun, who has always supported me in life and study. I am very grateful that she decided to give up her decent job in

China to come to the US to accompany me and without complaining much about the difficulties we have come into in the US.

Lastly, I would like to thank my parents for flying over the Pacific to visit me. They have supported me by cooking delicious dinners for me, giving me the power to get through this exhausting final research period.

Abstract of the Thesis

On the Design and Evaluation of a New Order-Based Join Algorithm

By

Guangqiang Li

Master of Science in Computer Science

University of California, Irvine, 2010

Professor Michael J. Carey, Chair

The join operation is one of the most important operations in relational database systems. It has been studied for more than 30 years. The state of the art in join algorithms has been stable for the last decade, with the gold standards remaining essentially the same. Currently, there is an effort at HP Laboratories to develop a new join algorithm that aims to be as good as all other major join algorithms and to be general enough so that mistaken join algorithm choices during compile-time query optimization effectively become "impossible".

In this thesis, this new join algorithm is reviewed and studied experimentally. A design of the new join algorithm based on the UCI computational platform Hyrax is proposed and implemented. The algorithm is then evaluated by analyzing its performance, buffer pool behavior, and the behavioral characteristics. This first study of the new algorithm focuses on ad hoc joins of unordered tables, and the algorithm is compared to implementations of both Grace Hash Join and Hybrid Hash Join.

1. Introduction

Efficient join algorithms for processing large volumes of data are very important for database systems [1]. Join operations can be implemented using nested loops- or sort- or hash-based algorithms. In early relational database systems, only nested loop-based and sort-based algorithms were employed [2]. In the last two decades, hash-based algorithms [3] have gained acceptance and popularity, and they are now considered generally superior to sort-based algorithms for large unordered joins. Query processing today mainly relies on three alternative join algorithms: Index Nested Loops Join as it is able to exploit an index on inner input; Merge Join, which is able to exploit sorted inputs; and Hash Join, which is able to efficiently handle inputs of very different sizes. Cost-based query optimization chooses the most appropriate algorithm(s) for each query.

There is an effort at HP Laboratories to design a new join algorithm [4] to substitute and replace all three of the current standard algorithms so that mistaken choices during compile-time query optimization essentially become impossible. The design goal of the new join algorithm is thus to match the performance of the best traditional algorithm in all situations. If the new join method knows that one or both inputs are sorted, it can further make use of the sorted property to skip its run generation step(s); in that case, it can outperform hash-based join methods and match the performance of merge join in the pre-sorted case. In this thesis, the design of this new join algorithm is reviewed; problems and their solutions for this new join algorithm are also discussed. A series of experiments are done to evaluate the new join algorithm and to compare it with Grace Hash Join and Hybrid Hash Join for performing ad hoc joins of unordered data. The emphasis of this thesis is therefore to evaluate this new join algorithm in detail for unordered joins.

The remainder of this thesis is organized as follows. Chapter 2 gives a brief review of previous work on join algorithms. The idea of the new join algorithm is described in Chapter 3, followed by its design and Hyrax implementation in Chapter 4. In Chapter 5, experimental results are presented and discussed; experiences during its design and implementation are shared as well. Finally, our preliminary conclusions and future work are covered in Chapter 6.

2. Prior Work on Joins

In this chapter, we give a brief review of previous work on join algorithms. They are Nested Loops Join, Sort-Merge Join, and Hash Join.

2.1 Nested Loops Join

The simplest and most direct algorithm for binary joins is the Nested Loops Join: for each item in one input (called the outer input), scan the entire other input (called the inner input) and find matches. The main advantage of this algorithm is its simplicity and its ability to compute a join with any arbitrary two-relation comparison predicate. The disadvantage is that the inner input is scanned repeatedly, which can be disastrous for performance [5].

Index Nested Loops join exploits a permanent or temporary index on the inner input's join attribute to replace inner file scans by index lookups. Index Nested Loops Join can be the fastest join method if one of the inputs is so small and if the other indexed input is so large that the number of index and data page retrievals, i.e., the product of the index depth and the cardinality of the smaller input, is smaller than the number of pages in the large input [2] [6].

2.2 Sort-Merge Join

The second commonly used join method is Sort-Merge join. The basic idea behind this join algorithm is to sort both inputs on the join attribute and then look for qualifying tuples by essentially merging the two relations [1] [2]. If one or both inputs are already sorted, the Sort-Merge join algorithm typically can make use of the existing ordering and jump to the merge phase. An input may be sorted because a database file was stored in sorted order, an ordered index was used, an input was sorted explicitly for the join, or the input came from a previous operation that produced sorted output [2] [5].

It is possible to improve the performance of the Sort-Merge join method by combining some or all of the merging phases of sorting with the merging phase of the join. First, sorted runs of size M (M is memory size) are produced for both relations R and S. Then, simultaneously, the runs of R are merged, the runs of S are merged, and the resulting R and S streams are merged as they are generated. The join condition is applied as the R and S streams are generated, and tuples in the cross-product that do not meet the join condition are discarded. Moreover, by using replacement sort, sorted runs of size approximately 2*M for both R and S can be produced. Consequently, if M >$\sqrt{L}$, there will be fewer than $\frac{\sqrt{L}}{2}$ runs of each relation, where L is the size of the larger relation. Thus, the total number of runs will be less than $\sqrt{L}$, i.e., less than M, and the R and S merging phases can be combined with the join merge with no need for additional buffers [1] [3].

2.3 Hash Join

The goal of Hash Join is to compute the equijoin of two relations R and S. Hash join algorithms are based on the idea of building an in-memory hash table on the smaller input and then probing this hash table using items from the other input. This classic algorithm works best

when the hash table for the smaller input can fit into real (not virtual) memory. If a hash table for the smaller input cannot fit into memory, it behaves poorly. Grace Hash Join and Hybrid Hash Join [3] make use of the concept of partitioning the two inputs so that computing the join can then be done by just joining the corresponding subsets of the two inputs.

The Grace Hash Join algorithm executes in two phases. The first phase begins by partitioning R into $\sqrt{F|R|}$ subsets (In Phase 1 of Figure 2.1, N = $\sqrt{F|R|}$, where F is a fudge factor used for conservatively sizing the hash table). This partitioning is done using a hash function h chosen such that R is partitioned into sets of approximately equal size. The same hash function h is then used to partition S into $\sqrt{F|R|}$ subsets as well. During Phase 2, joining is done by hashing for each pair of corresponding subsets (R_i, S_i). Grace Hash Join works considerably well when there is relatively little memory available, as it avoids repeatedly scanning R and S. However, when most of R fits into memory, Grace Hash Join does somewhat poorly since it scans both R and S twice. When memory size is less than $\sqrt{F|R|}$, it is necessary to recursively apply the hash partitioning technique to the join of each subset R_i with the corresponding subset S_i [1] [3].

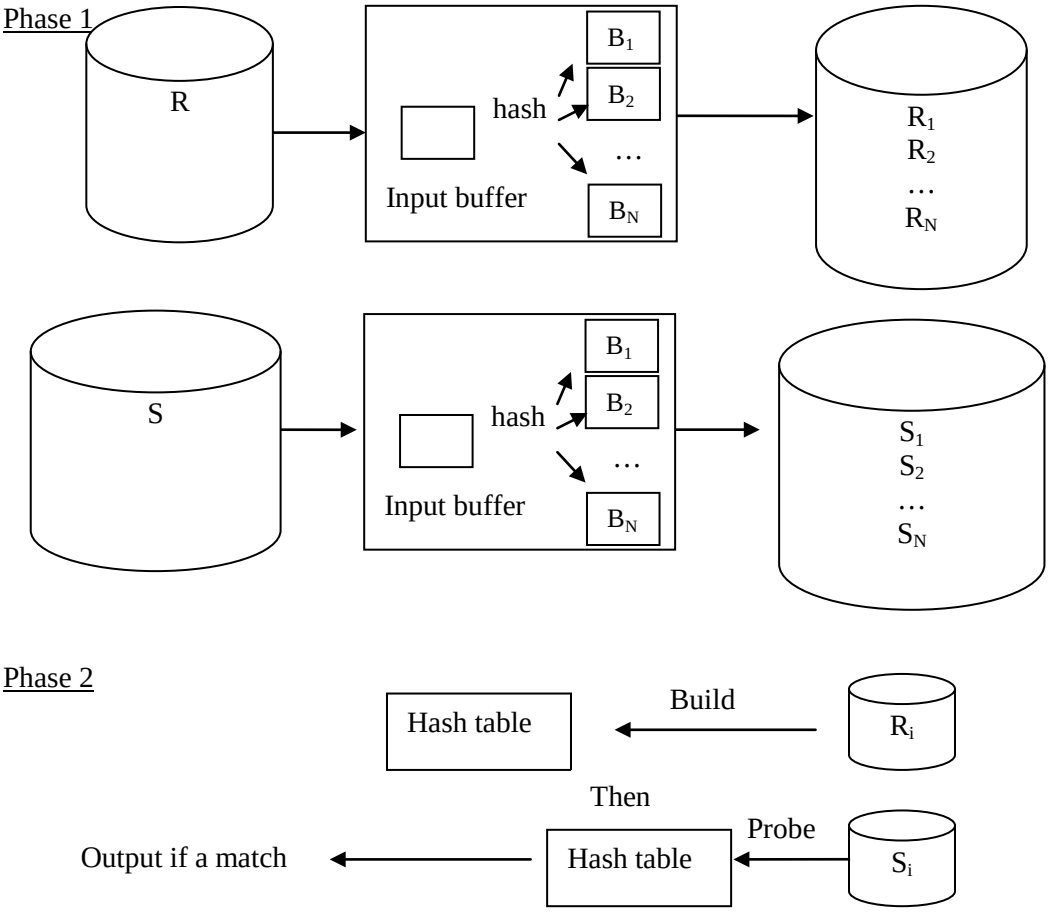


Figure 2.1 Grace Hash Join

Hybrid Hash Join improves upon Grace Hash Join by doing both partitioning as well as some hash-joining in the first phase over both relations. During the first phase, instead of using memory solely as a partitioning buffer as is done in the Grace Hash Join algorithm, only as many as B pages ($B = \left\lceil \frac{|R| * F - M}{M - 1} \right\rceil$) of memory are used to partition R into subsets that can subsequently fit in memory. The remaining memory (M - B pages) is used to build a hash table for R_0 that is created and utilized at the same time that R and S are being partitioned during Phase 1. The hash table is built for R_0 and then probed with tuples that lie in S_0, with join tuples being output immediately if there is a match. The second phase is identical to Grace Hash Join. Figure 2.2

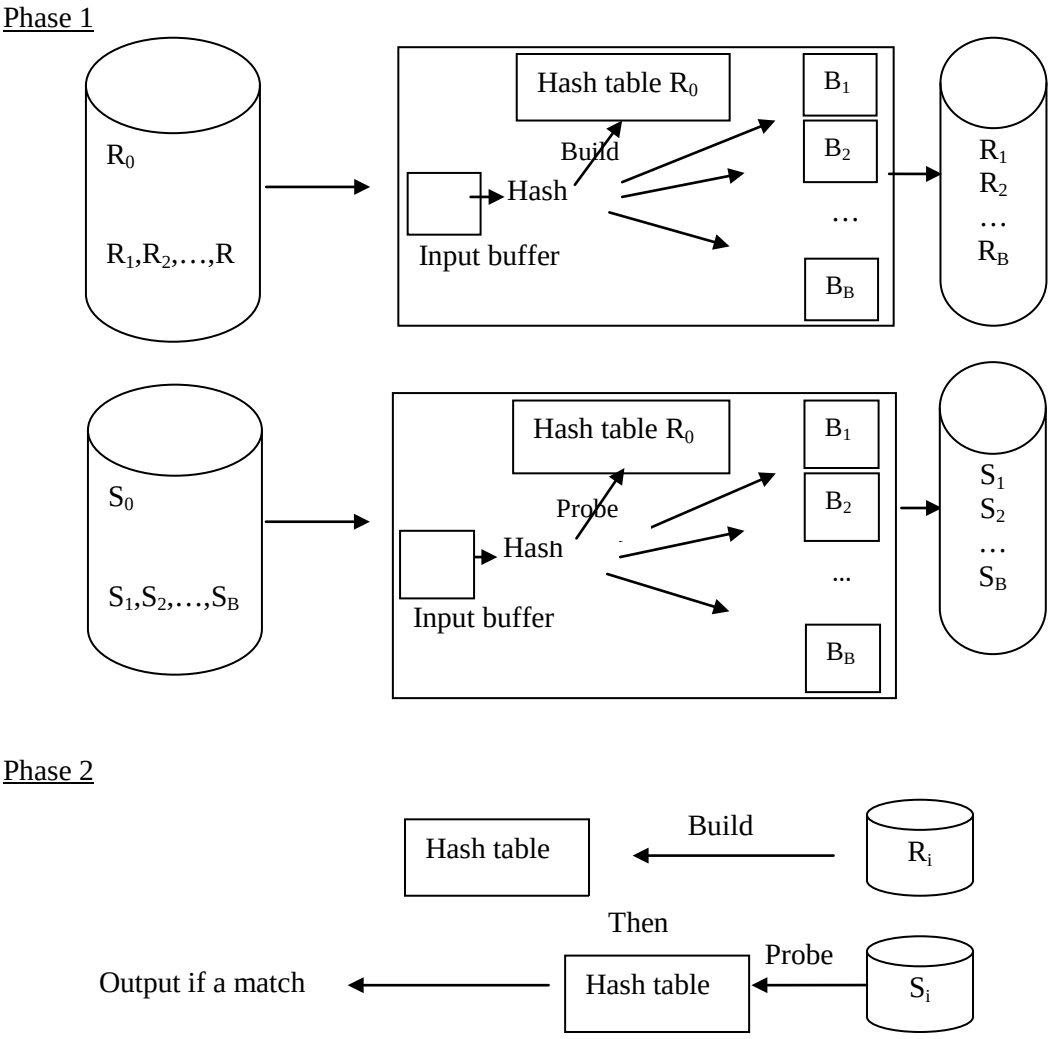


Figure 2.2 Hybrid Hash Join

shows Hybrid Hash Join. The advantage it has over Grace Hash Join is that tuples that fill in R_0 and S_0 are read only once. This is especially good when R is just a bit bigger than memory.

3. A New Join Algorithm: G-Join

The goal of the new join algorithm [4] – which will be referred to here as generalized join (G-Join) – is to be as good as all existing join algorithms so that one join algorithm can be used in all cases and mistaken choices of join algorithms during compile-time query optimization will

become impossible. We review this new (as yet unpublished) algorithm here. On one hand, the new join algorithm can exploit sorted inputs, behaving similarly to merge join. On the other hand, for unsorted inputs, it employs run generation very similar to traditional Sort-Merge Join – however, it avoids many of the merge steps in a merge sort. The behavior and cost function of recursive and Hybrid Hash Join have guided the algorithm design for unsorted inputs – but with the basic approach being based on merge sort.

The main idea is as follows [4]: For unsorted inputs, run generation produces a set of initial sorted runs. Assuming a fixed memory allocation, all run sizes will be roughly equal – any difference in the sizes of the two join inputs will be reflected in the count of runs for each input, not in the sizes of the runs. With sufficient memory and a small number of runs from the small input, join processing then roughly follows the sort order of join key values. A buffer pool holds pages from runs of the small input, with pages with higher keys successively replacing pages with lower keys. A single buffer page holds pages from all runs of the large input while such pages are being joined with the small input's pages in the buffer pool. In other words, the buffer pool is reminiscent of the in-memory hash table in a hash join, but its contents slowly turn over in the order of join key values and pages from the large input are joined during that process.

3.1 The G-Join Data Structures

Three priority queues are used in the new join algorithm. Priority queues A and B are for the small input (R), while priority queue C is for the large input (S). The detailed use of these priority queues is as follows [4]:

A. Priority queue A guides how to grow the buffer pool with respect to R. Its top entry indicates the next page of R to load into memory. Every run from the small input R has one entry in this priority queue at all times. The sort key for the queue is the high key of

the newest (highest) resident page in the buffer pool for each run, and the top of priority queue A is the run with the lowest of these values.

B. Priority queue B guides how to shrink the buffer pool with respect to R. Its top entry indicates the next R page to drop from memory. Each run from input R has one entry in this priority queue at all times. The queue's sort key is the high key of the oldest (lowest) page in the buffer pool for each run. The top of this priority queue is the run with the lowest of these values.

C. Priority queue C guides how to shrink the buffer pool for R based on join progress through the runs of S. Its top entry indicates the next page to join from the large input S. Each run from input S has an entry in this priority queue. The top of this priority queue is the run with the lowest low key value in the next page on disk. (We will see shortly how this information can be obtained or approximated.)

3.2 The G-Join Algorithm

The G-Join algorithm can be considered to be a two-level hybrid algorithm. If input R is less than memory size (M pages), the algorithm switches to in-memory hash join. This is detected when it goes to generate R's first run and discovers that it fits entirely in memory. If input R is larger than memory size, however, the new algorithm uses the G-Join method. In phase 1(see Figure 3.1), memory management borrows the idea of Hybrid Hash Join, but instead of using B pages as partition output buffers, G-Join uses the same B pages of memory for run generation of input R. The rest of memory (M – B pages) is used to build a hash table for a partition R_0. For input S, the G-Join algorithm probes the in-memory hash table with part of input S (S_0) while generating runs for the rest of S using B pages. A cut point is used to partition the input so that tuples with join keys less than the cut point are used to build/probe the hash table for R_0 with S_0, while tuples with join keys greater than the cut point go for run generation. Assuming the estimated size of R as well as the join key range (MIN, MAX) of the entire input R are known,

the high key of records for R_0 should be $\frac{M-B}{|R|}$ * (MAX –MIN) + MIN. Thus, this high key can be used as the cut point to decide which records go to hash table for R_0 and which ones go to run generation. The cut point is an approximate value, whose accurateness does not affect the correctness of the algorithm but the actual memory usage during the join process.

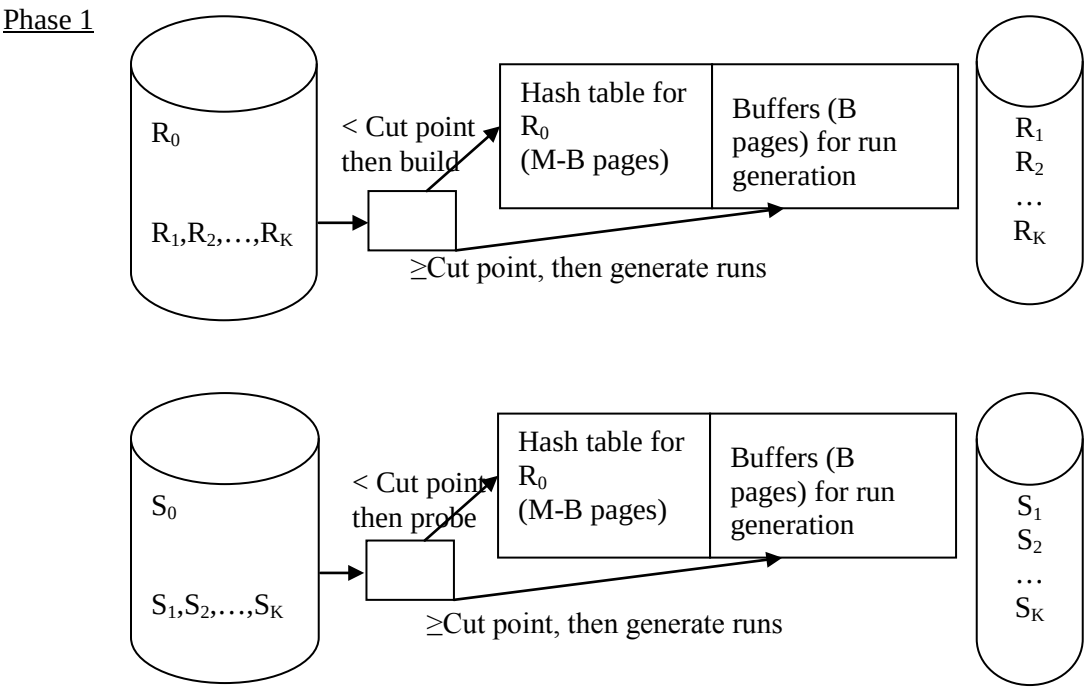


Figure 3.1 Phase 1 of G-Join

In Phase 2 (see Figure 3.2), the algorithm initializes the buffer pool and priority queues A and B with the first page of each run from input R. Priority queue C initially holds the information about the first page of each run from input S[1]. The algorithm then proceeds step-by-step until all matched pages from all runs of input S have been joined, i.e., until either priority queue B or C is empty. In each step a page is loaded from input S guided by the top of priority queue C. The high key of the current page from input S and the top entry of priority queue A are tested to see whether this page can be joined immediately with all R runs. If so, this page of S is joined with the pages of input R in the buffer pool. The information of the page of input S in priority queue C is then replaced with the information about the next page (i.e., the high key of the S page, which is the approximated low key of the next page) from the same run from input S. If the top entry of priority queue B is less than the top entry of priority queue C, the buffer pool is now able to drop some R page(s) guided by priority queue B, as these pages are no longer needed. Otherwise, if the current page of input S cannot be joined immediately with all R runs, the buffer pool loads additional pages from input R, guided by priority queue A, until the top entry in the priority queue A is less than the high key of the current page of input S – at which point the current S page can indeed be joined with all R runs (assuming there is enough memory for the additional R pages).

The overall complexity of the priority queue operations in G-Join is modest: each page in all runs from inputs R and S goes through 3 priority queues. Replace and pop operations are required in the priority queues. Tree-of-losers priority queues [7] [8] can implement these operations with a single leaf-to-root pass.

[1] Information about the first page of each run from input S is kept in memory during run generation and then passed to Phase 2.

Phase 2

LK: Low Key. For example, $LK_{1.1}$ represents the Low Key from Run#1 Page #1

HK: High Hey. For example, $HK_{2.0}$ represents the High Key from Run#2 Page#0

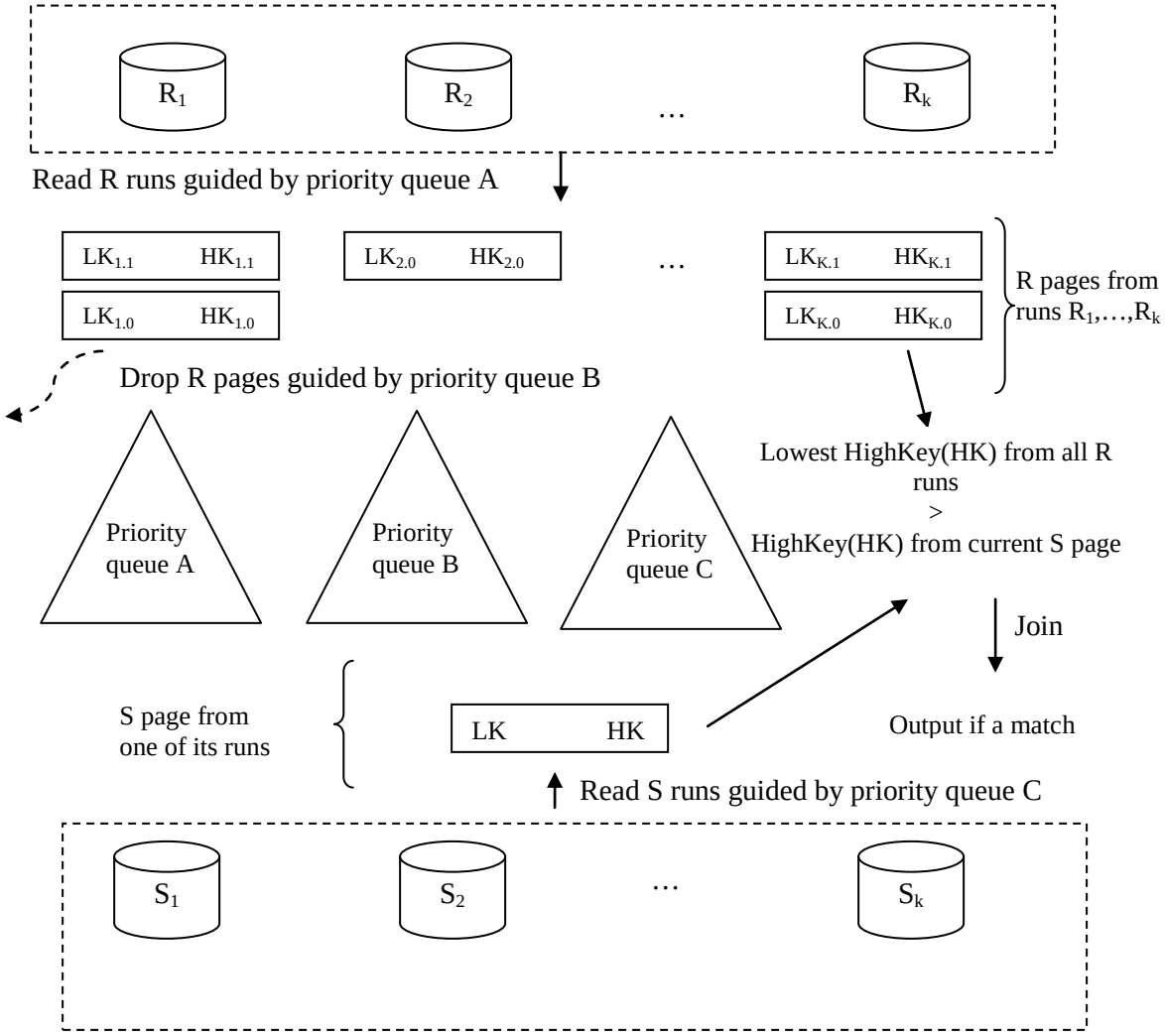


Figure 3.2 Phase 2 of G-Join

The Appendix contains several examples that illustrate the functioning of G-Join to clarify how pages of Input R and Input S come and go during join processing.

4. Design and Implementation of Join Operators in Hyrax

Now that we have reviewed the G-Join algorithm at a high level in Chapter 3, it is time to discuss the detailed design and implementation of the algorithm. We have implemented G-Join as a join operator in the UCI computational platform Hyrax because Hyrax and ASTERIX needed external join methods. Join operations can be easily parallelized based on Hyrax. We first briefly review the Hyrax platform and then discuss the Hyrax realization of G-Join.

4.1 Operator Design in Hyrax

Hyrax is a scalable computational runtime platform for data-parallel computing that is under development at UC Irvine as part of the ASTERIX project [9]. The Hyrax computational model is based on jobs made up of operators and connectors. Being the execution layer for ASTERIX, Hyrax determines and oversees the utilization of parallelism based on information and constraints associated with the job's operators as well as on the runtime state of the cluster.

Clients specify Hyrax jobs as Hyrax Operator Descriptor (HOD) nodes that are connected to each other by Hyrax Connector Descriptors to form an acyclic directed graph. As shown in Figure 4.1, for example, a Scan operator descriptor, a Basic Hash Join operator descriptor, and a Print operator descriptor can be connected to form a Hyrax job specification.

In the Hyrax code base, a HOD consists of one or more activities called Hyrax Activity Nodes (HANs). For example, in Figure 4.1 a Basic Hash Join operator consists of a build activity and a probe activity. Instances of these activities comprise an activity node graph. Hyrax guarantees that all activity nodes derived from the same HOD are co-located on the same Hyrax node at runtime so they can share information. In Figure 4.1, the hash table built in the Build Activity will be shared by the Probe Activity (represented by the dashed arrow).

In the Hyrax code base, the information shared among activities within a HOD is passed as environment variables. For example, the Build Activity builds a hash table which is set as one of the HOD's environment variables; the Probe Activity following the Build Activity can access the hash table by getting it from the environment.

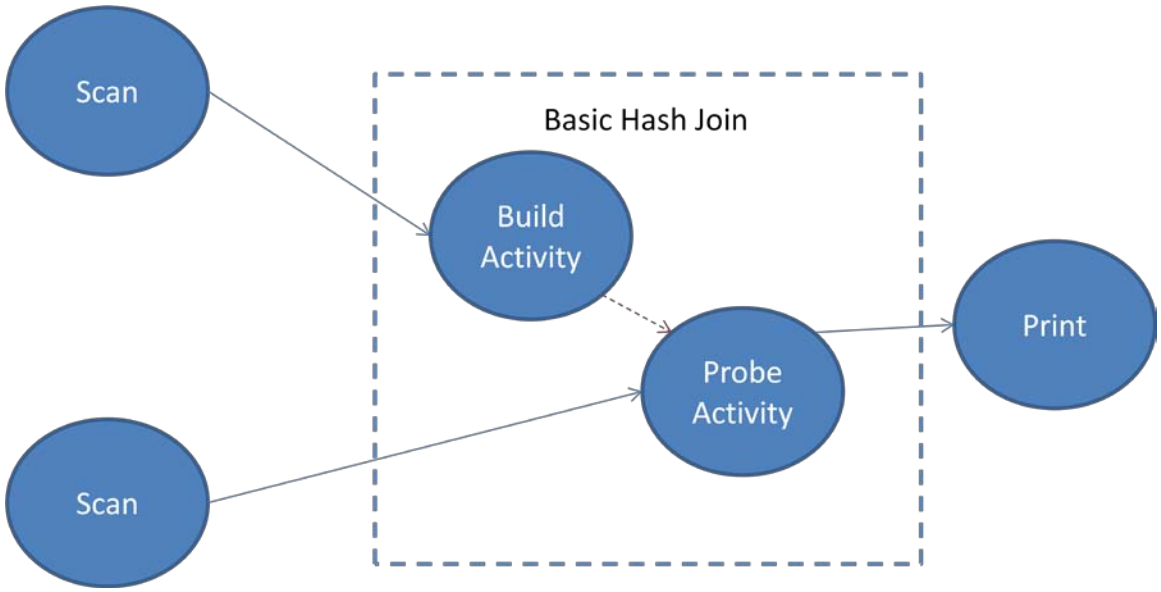


Figure 4.1 Hyrax Activity Node Graph

4.2 Design and Implementation of G-Join in Hyrax

The G-Join algorithm relies on three priority queues. The two priority queues for input R have entries of in-memory pages of runs; the priority queue for input S has entries of information about the next page to load from input S. Java has a priority queue implementation, but it is very general and generates objects at runtime that introduce overheads due to garbage collection. To avoid runtime overhead, fixed-size array-based priority queues for the G-Join algorithm have been implemented following the concept of "A Tree of Losers" [8].

4.2.1 G-Join Priority Queues Details

Because of the design of G-Join, we design two variants of fixed size priority queue.

Reference Priority Queue

Priority queues store the key value or a reference to the key object in their entries. The priority queue implementation in the standard Java SDK requires object construction at runtime whenever an object is to be added into the priority queue. The purpose of our design is to avoid this runtime object construction by keeping the size of the priority queue constant and reusing space. By designing the priority queue as "A Tree of Losers" [8], the number of entries in the priority queue is kept constant during push or pop, only the references in the entries are updated (no object construction). This is particularly suitable for merging a fixed number of run files generated during run generation. This priority queue implementation is used for priority queues A and B in the Hyrax implementation for the G-Join algorithm.

Copy Priority Queue

Typically, entries in a priority queue refer to objects currently in memory. The special design of this new join algorithm only provides one buffer for input S; therefore, the reference to that page is lost after the buffer is replaced with the next page from input S. To this end, our other variant of the priority queue is designed to store a copy of the key value in the entry of the priority queue. It has been designed very carefully, using a data buffer that starts with a small number of bytes and is able to grow dynamically as needed to store the join key copies. Although it holds the key value, the size of the priority queue keeps constant and the push or pop operation only needs to update the buffer in the entry; therefore, it maintains the benefits of avoiding object construction at runtime. Priority queue C in G-Join is a variant of this priority queue.

4.2.2 G-Join Run Generation Details

Given a fixed allocation of buffer pages, sorted runs are generated for input R and input S respectively. In order to make sure that writes are sequential during run generation, instead of generating a number of independent run files each of memory size, a single big file in which runs are written physically consecutive is generated. An array in memory is then used to keep track of the offset of each run within this big file. This array is used as needed to seek to the offset of a particular run within the run file.

During run generation for input S, the lowest key for each run is remembered in memory so that priority queue C can be seeded with accurate initial values regarding the first page of each S run. If the last run of input S is undersized, this run is merged with one of the other runs of input S so that undersized runs of input S are avoided. (This is crucial to ensure smaller page densities for R and S. The detailed motivation will be explained in Section 5.2.2.)

4.2.3 Detailed Description of G-Join in Hyrax

Assume the following parameters are given:

(1) estimated size of input R: |R|;

(2) the range of join keys of input R : [MIN, MAX];

(3) the operator's given memory size: M.

The implementation of the new join in Hyrax distributes the phases among two activities.

Activity 1. Build hash table and generate runs for R:

If R fits in memory, build a hash table for R; go to Activity 2;

otherwise, the algorithm will run in hybrid mode. The value of the cut point (mentioned in Chapter 3.2) is decided as follows:

CutPoint = $\frac{M-B}{|R|}$ * (MAX-MIN) + MIN,

where $B = \left\lceil \frac{|R| * F - M}{M - 1} \right\rceil$ (the B value is the same as in Hybrid Hash Join [3]).

Keys less than CutPoint will be used to build the in-memory hash table R_0 of size M –B; records with keys greater than the CutPoint will be used to generate sorted R runs using the remaining B buffer pages.

Activity 2. Probe hash table, do run generation for S, and then join runs of R and S:

1) If in Phase 1, R fits totally in memory, simply probe the in-memory hash table with each page loaded from S once at a time; otherwise, records with keys from S which are less than CutPoint will be used to probe R_0, while records with keys from S which are greater than CutPoint will be used to generate sorted S runs using B pages of memory.
2) The run files of R and S will then be processed as follows:

   Create priority queues A and B for R, and priority queue C for S.
   Each run of R has a list of buffer pages; initially each list is of size one. The buffer pool is composed of the buffer lists of runs from input R.

   Read a page from S guided by the top entry of C, called pageC.
   If the top entry of A is greater than the high key of pageC
      join tuples in pageC with tuples in the buffer pool
      while the top entry of B is less than the top entry of C
         shrink the buffer pool by dropping the first buffer page in the list guided by B
   otherwise, grow the buffer pool by reading pages from R guided by the top entry of A until pageC is able to join with all the buffer pool pages.

4.2.4 Buffer Pool and In-Memory Join Details

Each run of input R has a list of in-memory buffers. Given R and S pages with comparable join key density, the average length of the buffer list per run will be around 2. This is because, if the join keys of R and S are drawn from the same domain, and the input is randomly ordered, each run of R and each run of S should roughly cover the same key range. That is, as sizes of runs of R and S are the same, i.e., the same number of pages per run, a page of a run from either R or S should roughly cover the same key range. A hash table is built for the in-memory R run pages. The hash table is designed to efficiently handle pages coming and going.

5. Performance Results

After implementing Hybrid G-Join in Hyrax, we have done a series of experiments to evaluate its performance and buffer pool usage. In this chapter, we will first introduce the parameters of our experiments. Second, we will show some experimental results for Hybrid G-Join, Grace Hash Join and Hybrid Hash Join. Finally, we will share some experiences and lessons we have learned while designing, implementing, and evaluating the Hybrid G-Join algorithm.

5.1 Experimental Parameters

Table 5.1 lists the parameter values that we have used. We study the G-Join algorithm's behavior for the tables based on the TPC-H benchmark [10]. Two different data set sizes are used – 0.1X and 1.0X. 0.1X data is generated using TPC-H database generator with a scale factor of 0.1, generating a set of data of size 100MB. The customer table is 2.4MB, having 15,000 records; the orders table is 16.8MB, having 150,000 records. 1.0X data is generated using TPC-H database generator with scale factor 1.0, generating a set of data of size 1GB. The customer table is 24MB, having 150,000 records; while the orders table is 170MB, having 1,500,000 records. In each case,

the generated customer table is originally sorted by its primary key CID, while the generated orders table is originally sorted by its primary key OID. The second attribute of the orders table is the (unsorted) foreign key CID. From these generated tables, we then created both CID-sorted and unsorted versions of each table for use in our experiments.

Table 5.1 Parameters

Parameter	**Meaning**	**Value(s)**
P	Page size	32 KB
F	Universal fudge factor	1.2 [3]
|R|	Size of Input 1 in pages	0.1X: 86 pages; 1X: 855 pages
|S|	Size of Input 2 in pages	0.1X: 649 pages; 1X: 6487 pages
M	Memory assigned in pages	8-1280 pages

The implementations of G-Join, Grace Hash Join, and Hybrid Hash Join for the experiments are written as Hyrax operators in Java. The experiments were run on the Hyrax platform on a DELL OPTIPLEX 960 Desktop Computer, which has a Core 2 Duo CPU @3.16GHz, 4GB memory, and a 320GB hard-drive (WDC WD3200AAKS-7) with a 16MB cache. The operating system is Linux Ubuntu 9.04. In order to avoid the effect of file system caching, we compiled a modified OpenJDK 1.7 [11] that enables Direct I/O operations for the Java class RandomAccessFile. (All file operations in our implementations of these three join algorithms use RandomAccessFile.) To try to avoid timing variance, we ran each join method for a given memory size five times and we report the average value.

5.2 Performance Experiments

The experiments use both the 0.1X and 1.0X TPC-H data sets and join the primary key CID of the customer table with the foreign key CID of the orders table for the eight different combinations of inputs listed in Table 5.2. In each case, “Sorted” refers to whether or not the input is sorted by CID value.

Table 5.2 Experiment List of Eight Combinations of Inputs

Combination	**Data Set Size**
Sorted Input 1 , Sorted Input 2	0.1X
Sorted Input 1 , Sorted Input 2	1.0X
Unsorted Input 1 , Sorted Input 2	0.1X
Unsorted Input 1 , Sorted Input 2	1.0X
Sorted Input 1 , Unsorted Input 2	0.1X
Sorted Input 1 , Unsorted Input 2	1.0X
Unsorted Input 1 , Unsorted Input 2	0.1X
Unsorted Input 1 , Unsorted Input 2	1.0X

In this section, we will first look at the join time comparisons among G-Join, Hybrid Hash Join, and Grace Hash Join. We then will look into the details of the buffer pool behavior of G-Join.

5.2.1 Join Time Comparisons

To evaluate the performance of a join operator, we measured the execution time that each join operator takes to finish the join operation. In Hyrax, a join operator is assigned a soft budget of memory, in the sense that it is given a target but may over-use memory by getting extra pages from the buffer pool. This can occur when a hash partition is larger than its estimated size or when the number of in-memory pages for a run exceeds the estimated window of two pages. In the experiments, we measured the average and the maximum memory usage of an operator.

For the sake of simplicity, in Figure 5.1 to Figure 5.8, “HHJ” is short for Hybrid Hash Join, “GHJ” is short for Grace Hash Join, and “HGJ” is short for Hybrid G-Join. We now examine the results.

5.2.1.1 Sorted Input 1, Sorted Input 2

Figure 5.1 presents the 0.1X results for sorted inputs. (Note that the algorithms do not "know" that the inputs are initially sorted.). Figure 5.2 shows the corresponding results for the larger 1.0X data set.

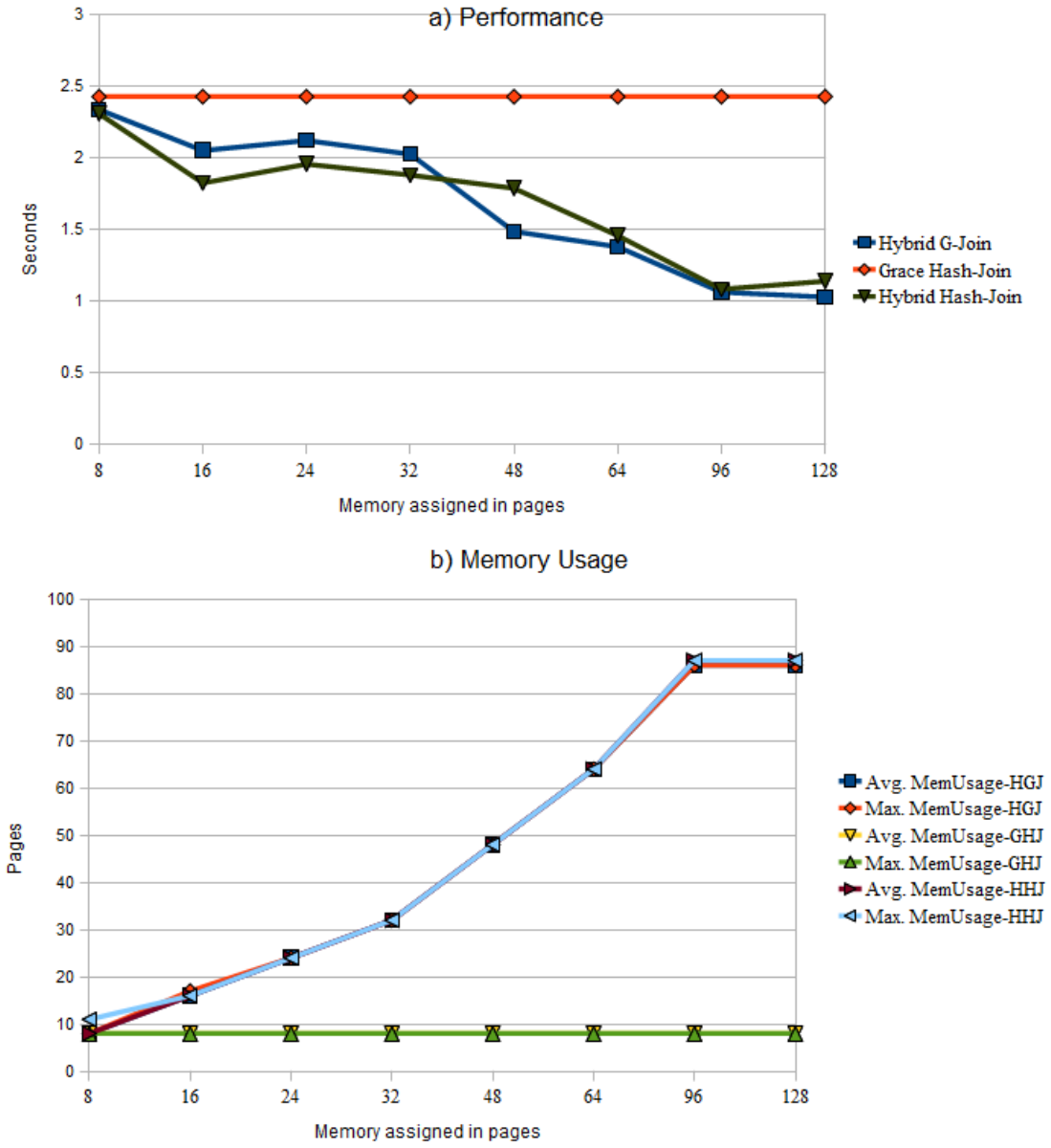


Figure 5.1 Sorted Input 1, Sorted Input 2, 0.1X Data Set

As Figure 5.1a shows, when M=8 (low memory), Hybrid Hash Join runs in Grace mode, behaving almost the same as Grace Hash Join. As memory size increases, the time for Grace Hash Join remains constant, while the times for Hybrid Hash Join and G-Join both decrease. Grace Hash Join is not affected by memory size changes as it utilizes a fixed amount of memory. For Hybrid Hash Join and Hybrid G-Join, their performances are quite similar. More and more records are stored in their in-memory hash tables as memory increases, and therefore fewer I/Os are involved. Since both inputs are sorted on the join key, G-Join actually behaves much like a merge join. Runs of Input 1 are loaded into the buffer pool sequentially, page by page, guided by priority queue A. Similarly, runs of Input 2 are sequentially loaded into their single buffer page for the join guided by priority queue C. Order is a very beneficial property for G-Join, because joining during Phase 2 only involves sequential I/Os in the case of ordered input. (We will see why this is crucial in the experiments following.) Moreover, note that Hybrid G-Join doesn't see in advance whether one or both inputs are sorted. If it knew, it would skip run generation(s) and would be even cheaper. The initial larger dip on the curve when M=16 may come from the variance of timing, because the total time is only about two seconds; although we took an average of five measurements, it is still possible for the timing to be affected by some noise.

Turning to memory usage, Grace Hash Join only takes a fixed amount of memory pages. When the memory allocation is M=8, Grace Hash Join and Hybrid Hash Join each need more than 8 pages so as to allow one buffer for each partition. Thus, both algorithms must overuse memory relative to their allocation. When M=8, Hybrid G-Join uses all the memory to do run generation, so it will not use more than the memory budget in Phase 1.When $M \geq 16$, G-Join runs in its hybrid mode; also as its inputs are sorted, the buffer pool barely needs to grow, and therefore memory usage stays low for G-Join.

In Figure 5.2, with the larger data set 1.0X, Hybrid G-Join performs better than Hybrid Hash Join at low memory. The reason is the CPU cost. For example, according to our

measurement when M=16, the number of comparisons for G-Join is about 24 million; On the other hand, Hybrid Hash Join requires over 33 million comparisons to join all of the partitions. This can happen due to hash table collisions inside a partition, i.e., when multiple join keys fall into the same hash line in the hash table built for this partition[2]. Joining by building and probing the hash table incurs multiple comparisons.

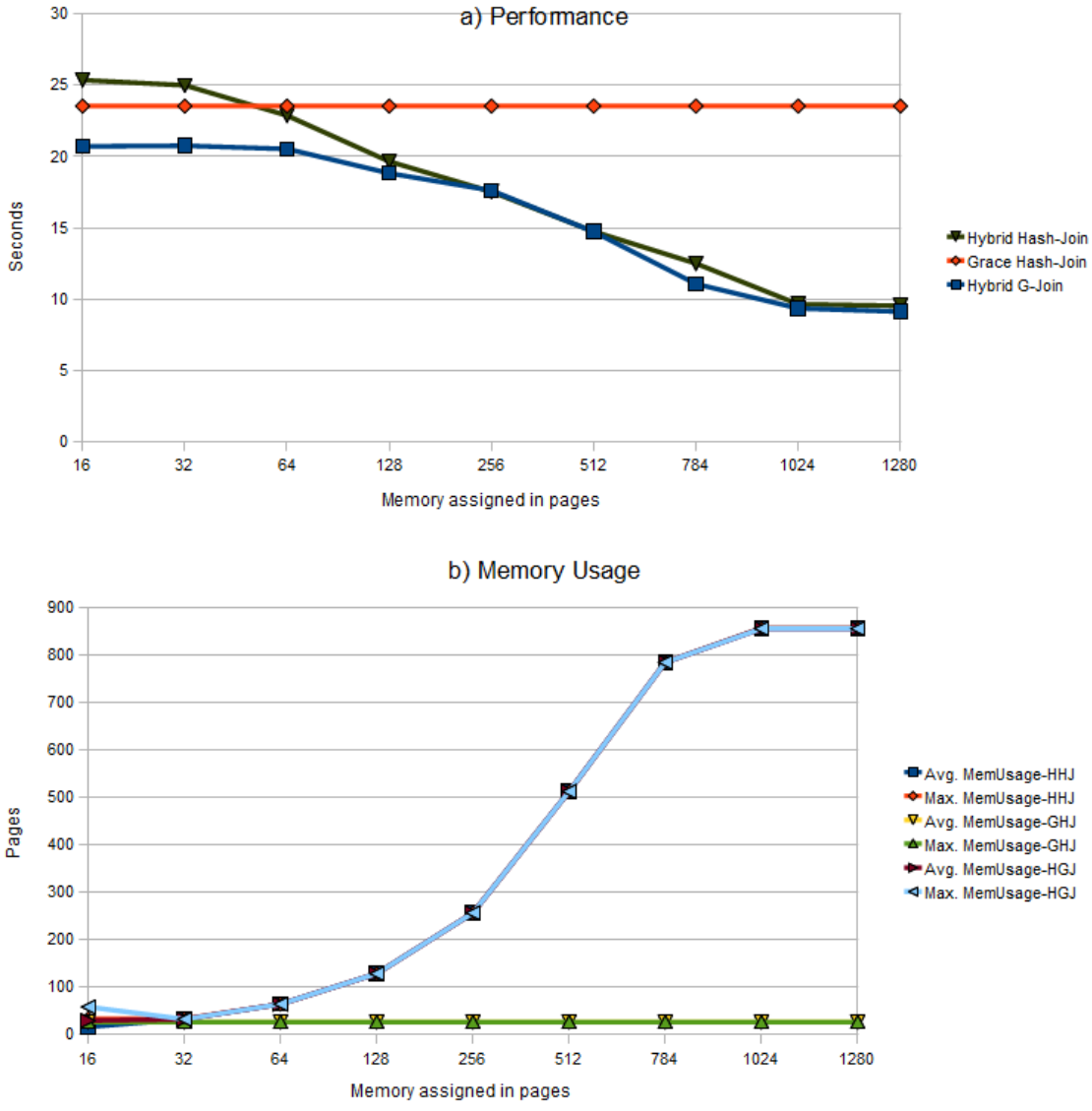


Figure 5.2 Sorted Input 1, Sorted Input 2, 1.0X Data Set

[2] This is a bug from bad hash function for repartitioning tuples in each partition of Input 1.

5.2.1.2 Unsorted Input 1, Sorted Input 2

Figure 5.3 shows the performance and memory use for the 0.1X data set for Unsorted Input 1 and Sorted Input 2, and Figure 5.4 shows the corresponding results for the larger 1.0X data set.

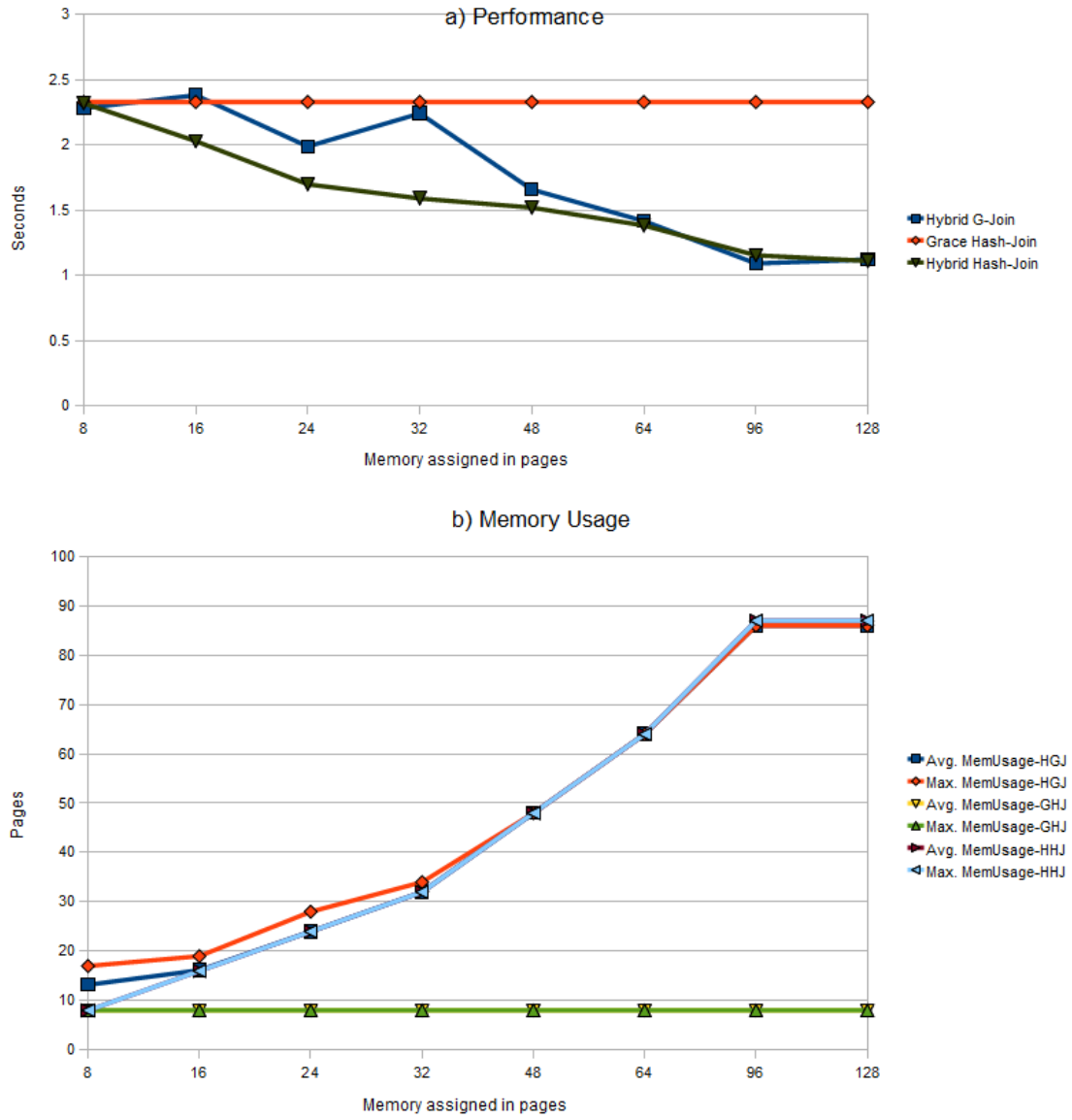


Figure 5.3 Unsorted Input 1, Sorted Input 2, 0.1X Data Set

The performance curves in Figure 5.3 are similar to the ones in Figure 5.1. With data set 0.1X, the ordering of Input 1 seemingly does not matter, in the sense that G-Join is insensitive to the order. G-Join's performance is close to Hybrid Hash join, and its memory usage is low because of the ordering of Input 2. Although Input 1 is unsorted, as long as the key range of a page of runs of Input 1 is not too narrow, the chance of requiring growing the buffer pool is small.

In Figure 5.4 we switch to larger data set to see if there is any difference at all caused by the ordering of Input 1.

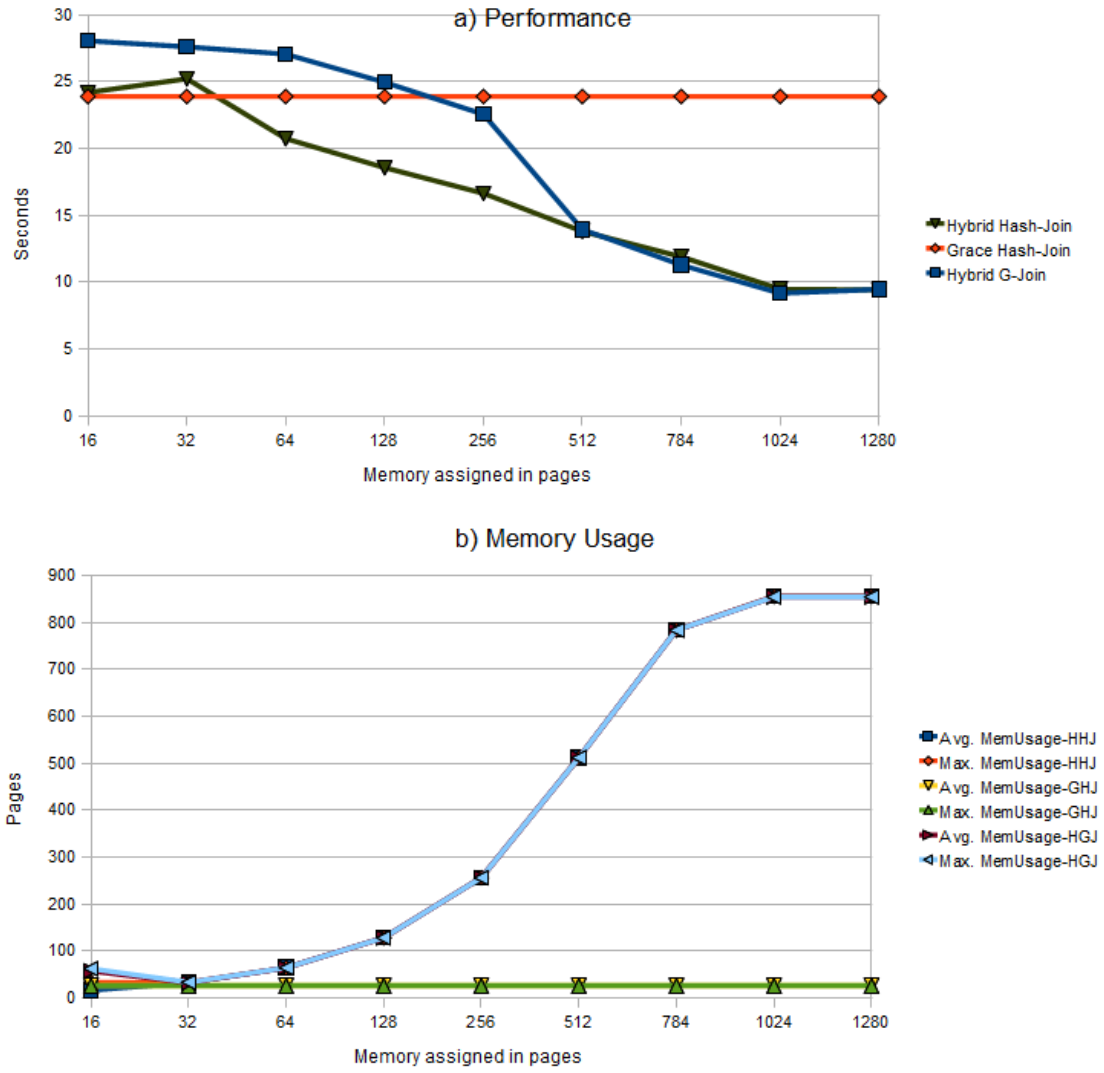


Figure 5.4 Unsorted Input 1, Sorted Input 2, 1.0X Data Set

The new trend in Figure 5.4 is that Hybrid Hash-Join beats Hybrid G-Join nearly all the way along, except when M≥512. We measured the number of I/Os that Hybrid G-Join and Hybrid Hash Join each involve and found out they are doing a very similar amount of I/Os. The only difference is that Hybrid G-Join sequentially writes runs of Input 1 onto disk in Phase 1 and then in Phase 2 it reads them back guided by priority queue A (random reads). Hybrid Hash-Join randomly write partitions of Input 1 to disk in Phase 1 and then reads them back sequentially, partition by partition. Our measurement shows that the join time of G-Join in Phase 2 increases when the Input 1 is unsorted. This indicates that Hybrid G-Join's random reads during Phase 2 contributes to the time difference between Hybrid G-Join and Hybrid Hash Join; random reads seem to be more expensive than random writes. We will see more clearly how and why random reads matter in G-Join in next section with the larger input, i.e., Input 2, also being unsorted, and we will discuss there why the ordering of Input 2 is crucial to the G-Join implementation's current performance and memory usage.

5.2.1.3 Sorted Input 1, Unsorted Input 2

Figure 5.5 shows the performance and memory use for the 0.1X data set for Sorted Input 1 and Unsorted Input 2, and Figure 5.6 shows the corresponding results for the larger 1.0X data set.

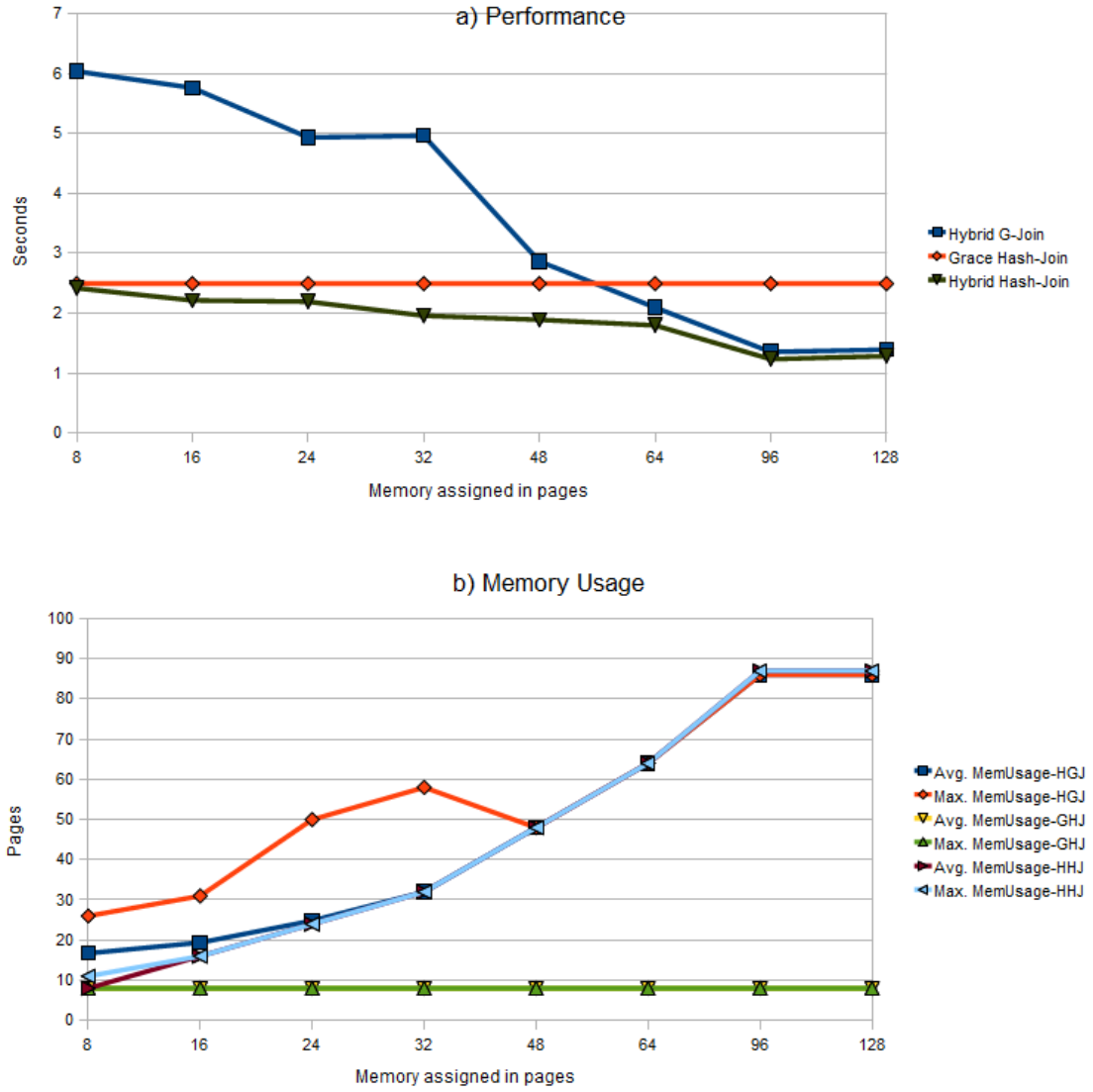


Figure 5.5 Sorted Input 1, Unsorted Input 2, 0.1X Data Set

From Figure 5.5, we can see a prominent difference given that Input 2 is now unsorted. G-Join has poorer performance at low memory—it requires using more than twice the time as

Hybrid Hash Join needs to do the join when M=8. This appears to be because Hybrid G-Join does random reads as it progresses through Input 2 during the join phase, always taking the next best page from the run the priority queue C indicates. Although Hybrid Hash Join also does some random I/Os, it is different – Hybrid Hash Join does random writes (not reading), which can be absorbed by the operating system and disk cache. The number of I/Os for Hybrid Hash Join and Hybrid G-Join involved are almost the same (Hybrid G-Join may read one extra run of Input 2 to merge with the last run to solve the "last run" problem), yet their performance difference is large due to this random I/O effect. This was an interesting (and surprising) finding in our experiments.

With Sorted Input 1 and Unsorted Input 2, we see in Figure 5.5b that the average memory usage for Hybrid G-Join was roughly the memory size it is assigned, in the sense that only one page per run of Input 1 is needed. Notice the maximum memory usage for Hybrid G-Join can be twice as high as the memory size. Our current implementation of Hybrid G-Join uses quick sort instead of replacement sort to generate runs of memory size, so there are twice the number of runs of Input 1 generated. Therefore using twice the amount of memory is fine for now. Once we have replacement sort, the memory usage will be halved.

As shown in Figure 5.6, the patterns of performance and memory usage using the 1.0X data set are pretty much the same as those using the 0.1X data set.

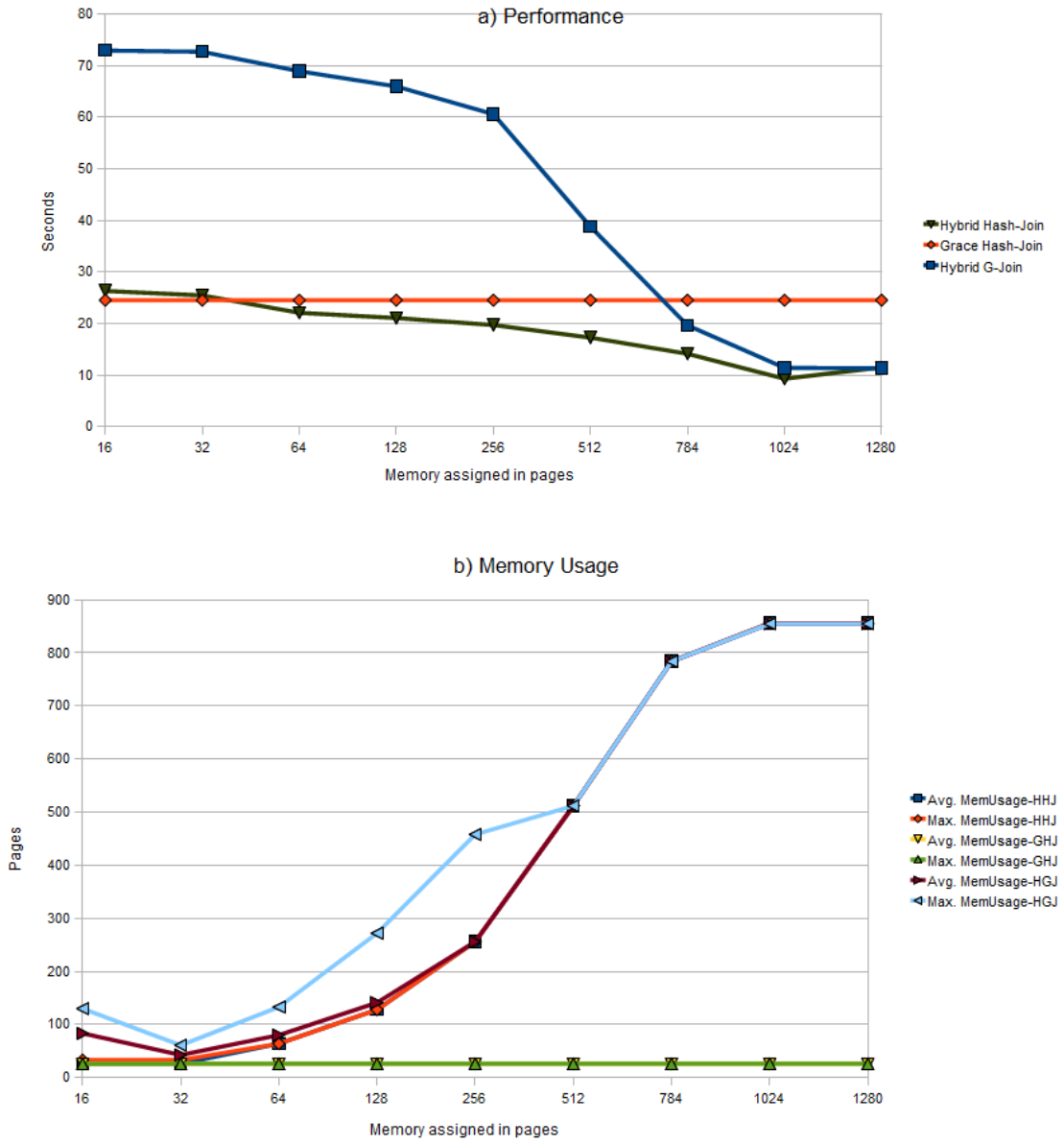


Figure 5.6 Sorted Input 1, Unsorted Input 2, 1.0X Data Set

5.2.1.4 Unsorted Input 1, Unsorted Input 2

Figure 5.7 shows the performance and memory use for the 0.1X data set for Unsorted Input 1 and Unsorted Input 2, and Figure 5.8 shows the corresponding results for the larger 1.0X data set.

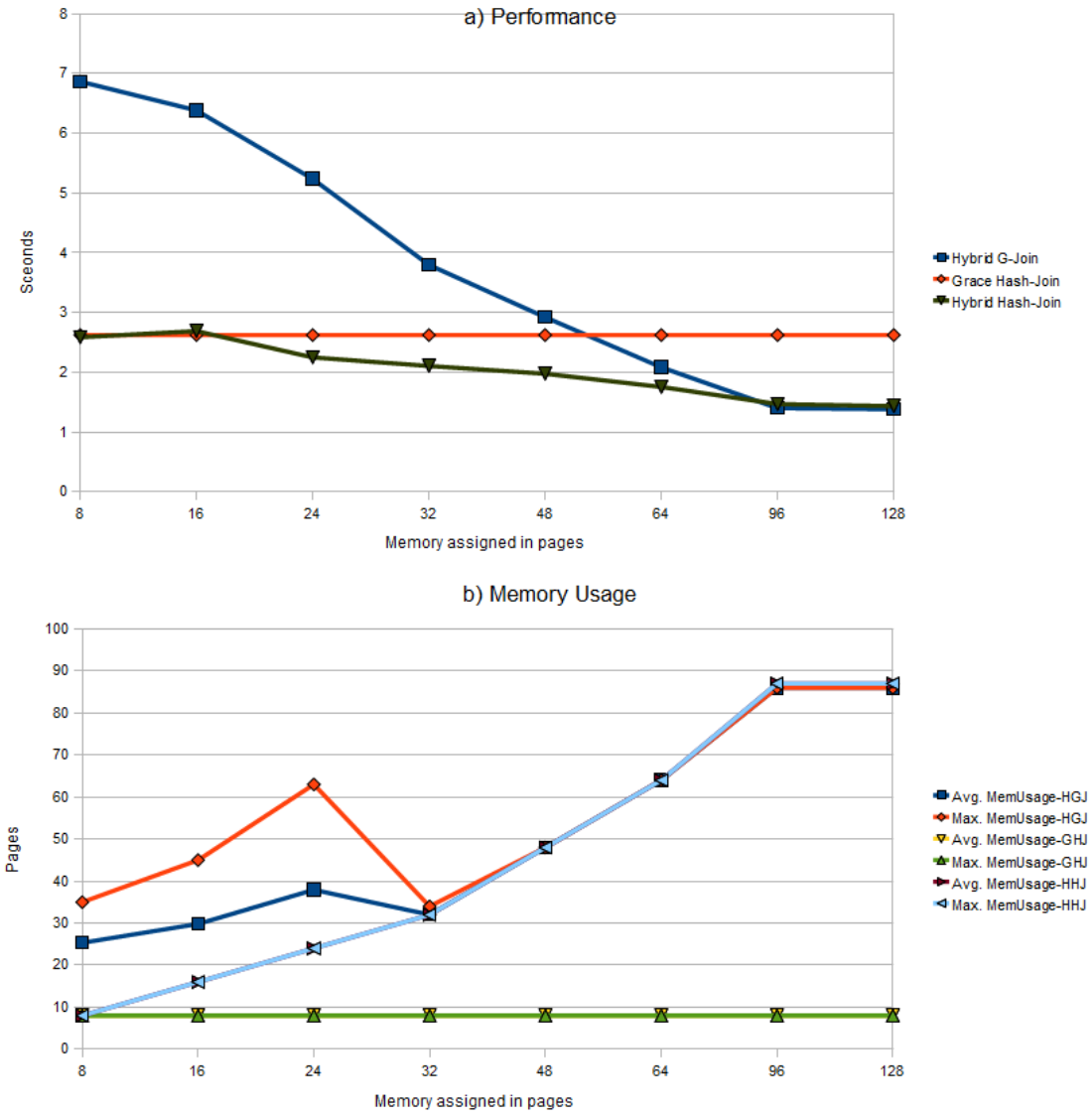


Figure 5.7 Unsorted Input 1, Unsorted Input 2, 0.1X Data Set

Not surprisingly in Figure 5.7, G-Join again performs poorly at low memory, just like Section 5.2.1.3. The memory usage for G-Join increases significantly when M=24. According to our measurement, when M=24, there are 23 runs of Input 1 generated in Phase 1 of G-Join; in Phase 2, the average buffer requirement per run of Input 2 is 1.65 pages, while the maximum requirement per run is 2.74 pages. In other words, the maximum memory requirement in Phase 2 of G-Join is 2.74*23 = 63 pages, more than twice of the memory budget (24 pages) assigned to the operator. This maximum memory requirement (although only temporary) is surprising.

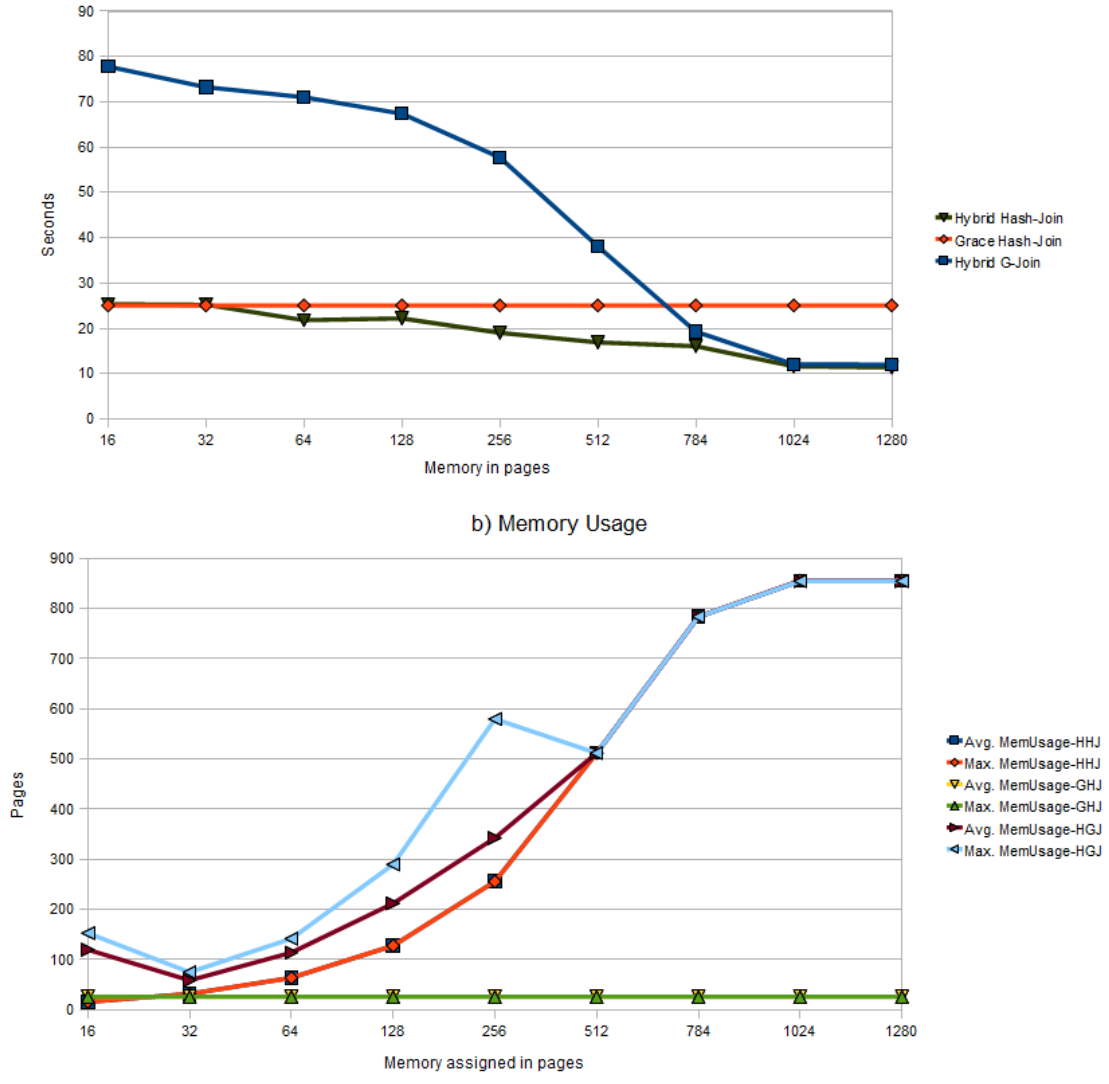


Figure 5.8 Unsorted Input 1, Unsorted Input 2, 1.0X Data Set

With the larger 1.0X data set, the patterns of performance and memory usage shown in Figure 5.8 are very similar to the ones shown in Figure 5.7. The surprising and interesting thing in Figure 5.7 and Figure 5.8 is why the maximum memory usage can be so high. We will further discuss this kind of case in the next section.

5.2.2 Buffer Pool Behavior for Input 1

One concern (as we discussed in Section 5.2.1) is the size of the buffer pool for the smaller input (Input 1). If the join keys of Input 1 and the join keys of Input 2 come from the same domain, and if the runs of Input 1 and Input 2 are of the same length, the width of the key range of a page (called its key density) in either Input 1 or Input 2 should be more or less the same. Therefore, at most two consecutive pages from each run of Input 1 should need to be loaded into memory to cover the key range of the page from Input 2 [4]. In other words, G-Join is expected to take no more than 2 buffers for each run of Input 1. For example, in the experiments in Section 5.2.1.3 and 5.2.1.4, we show that the maximum number of buffers for a run is often between 2 to 3. This is surprisingly high. The results reported in Section 5.2.1 are already based on our optimization that aims to address this problem, but the maximum memory requirement is still more than 2 pages per run of Input 1. Let's look at some statistics that we collected by watching the memory usage of G-Join before we implemented that optimization.

Figure 5.9 shows the average buffer requirement per run of Input 1 during join processing. Figure 5.10 shows the buffer requirement for Run#0 of Input 1. The x-axis represents the number of steps that the G-Join has proceeded and can simply be regarded as time. One step of the G-Join algorithm can be a join operation, growing the buffer pool of Input 1 once, or dropping one page from the buffer pool. The y-axis represents the number of in-memory pages that a run requires at a particular time or step.

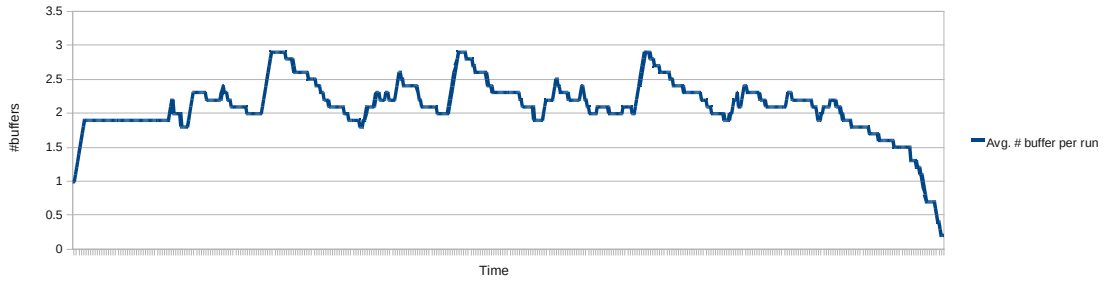


Figure 5.9 Average Number of Buffers per Run

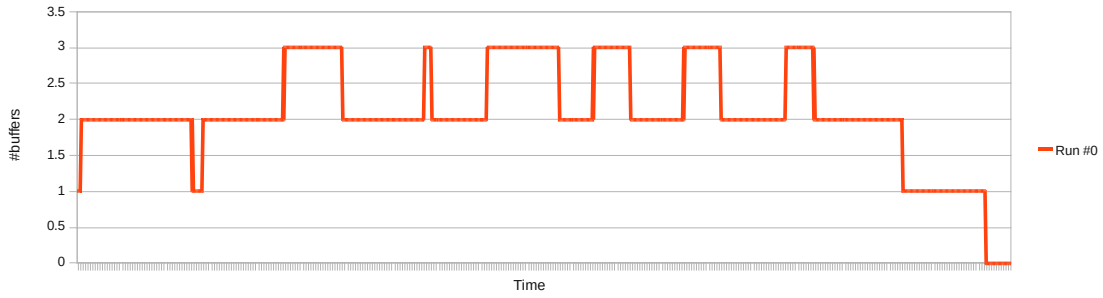


Figure 5.10 Buffer Status of Run#0 of Input 1

In Figure 5.9, we see that the maximum requirement of buffers per run can reach as high as 2.9, which is higher than the expectation of 2. This can be bad for Hybrid G-Join, as the maximum buffer requirement for the entire buffer pool can then be as high as 2.9 * the number of Runs of Input 1. A deeper look into the buffer pool for one run is shown in Figure 5.10; Run#0 experiences a buffer peak of 3 buffers at six different time periods. This phenomenon happens if the key range of a page from Input 2 is slightly larger than average when the key ranges of the pages from runs of Input 1 are slightly smaller than average; a page from Input 2 can then overlap three pages of a run from Input 1. Table 5.3 is a snapshot of buffer pool key ranges taken as G-Join proceeds.

Table 5.3 A Snapshot of Key Ranges in the Buffer Pool of Input 1

Run #	**Key ranges of pages in the buffer pool**		
Run#0	[1578 ,3357]	[3359 ,4966]	[4972 ,6830]
Run#1	[1724 ,3334]	[3336 ,4950]	[4986 ,6597]
Run#2	[1828 ,3227]	[3233 ,5062]	[5072 ,6497]
Run#3	[1475 ,3235]	[3258 ,4901]	[4917 ,6643]
Run#4	[1714 ,3304]	[3308 ,4860]	[4865 ,6395]
Run#5	[1504 ,3585]	[3592 ,5086]	
Run#6	[1583 ,3386]	[3401 ,5080]	[5081 ,6604]
Run#7	[1739 ,3429]	[3436 ,5198]	
Run#8	[1649 ,3178]	[3181 ,4755]	[4766 ,6462]
Run#9	[4 ,3493]	[3501 ,6897]	

At the moment of the snapshot showing in Table 5.3, the in-memory page from Input 2 had a key range [2840 ,6091], whose width is 3251, twice as wide as the average width of key range (about 1600 as indicated in Table 5.3) of Input 1. In the experiments, we observed that the average buffer requirement is just slightly larger than 2 per run, but the maximum buffer requirement can reach as high as 2.9. Where can this maximum buffer requirement happen? Further investigation revealed that this wide range [2840 ,6091] happened at the very last run of Input 2. This last run can have many fewer pages than average; therefore, given the same domain, each page of the last Input 2 run can have a much wider key range than the pages of Input 1.

Currently, we have addressed the “last run” problem by merging the last Input 2 run with one of the other runs of Input 2. In this way, we were able to lower the maximum buffer requirement from 2.9 to 2.6 (see Figure 5.11). Note that this has still not met the original expectation of 2 buffers, however. (Our experiments reported in Section 5.2 have already employed this optimization; otherwise memory usage would be higher than the results reported.)

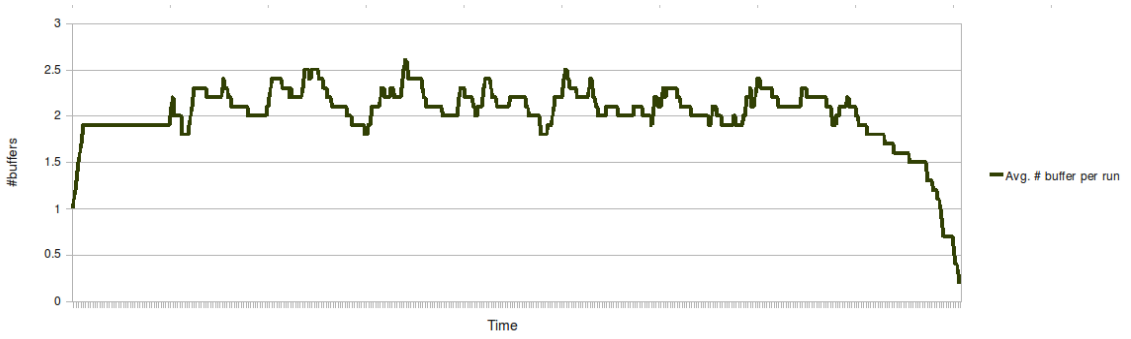


Figure 5.11 Average Number of Buffers per Run After Fixing the “last run” Problem

To further see what’s going on, let us look into the histograms of runs (after fixing the “last run” problem) of Input 1 and Input 2 respectively. In Figure 5.12, the histogram is based on the customer table in the 0.1X dataset. The width of a bucket in the histogram is 100. The x-axis shows the range width, and the y-axis shows the number of pages that fall in a particular range. For example, the x-axis value “13” means the key range is from 1300 to 1399; and the y-axis value “2” means there are two pages in Input 1 that fall into this key range.

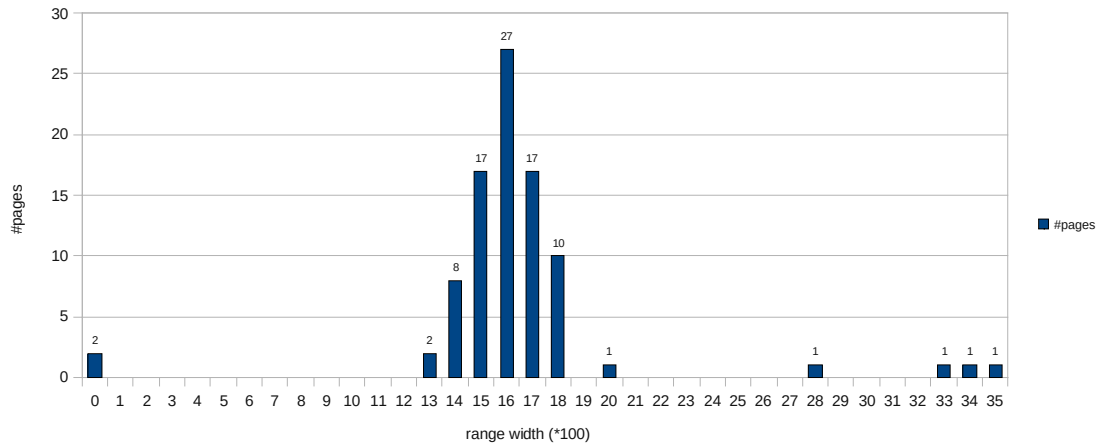


Figure 5.12 Histogram of Input 1

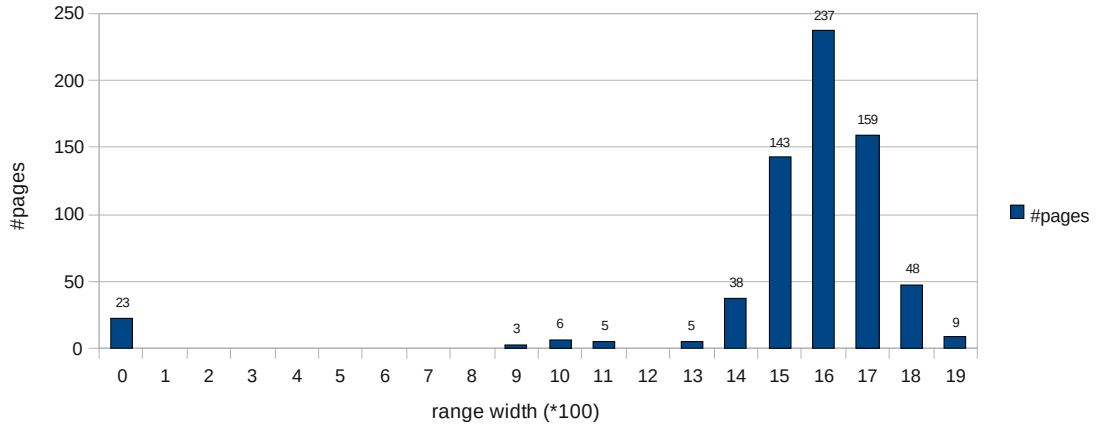


Figure 5.13 Histogram of Input 2

As shown in Figure 5.13, even after fixing the “last run” problem, there are still 9 pages from Input 2 with a width of key range in the 1900’s. In those cases, a slightly overweight page of Input 2 will overlap three pages of the slightly underweight runs of Input 1. This is not due to skew, but usually happens due to natural variations in the data. Thus, the conjecture of an average of 2 pages per run seems to turn out to be intuitive but overly optimistic.

5.3 Experiences and Lessons

Having experimented with various alternative details in the implementation of G-Join, we have learned several interesting additional lessons that we want to share with readers. In Section 5.3.1, we discuss one of the difficulties of Hybrid G-Join with respect to its usage in practice. In Section 5.3.2, we describe one of the performance puzzles that we encountered during the implementation and evaluation of the G-Join algorithm and share how we solved it. In Section 5.3.3, we discuss how we chose the in-memory join method for G-Join.

5.3.1 Choice of Cut Point

Since the G-Join algorithm applies a hybrid approach, some part of the data from Input 1 will go to an in-memory hash table; the other part of the data will go for run generation, so G-Join

needs a way to decide the cut point for routing data one way or the other. For Hybrid Hash Join, it is straightforward to use hash partitioning instead of a cut point to route the data to their appropriate partitions. However, G-Join uses range partitioning instead of hashing to route the data. Therefore, a value-based cut point needs to be chosen so that partitioning can be done.

If the join key is numeric, the cut point can be calculated by CutPoint = $\frac{M-B}{|R|}$ * (MAX-MIN) + MIN, where $B = \left\lceil \frac{|R| * F - M}{M-1} \right\rceil$. (The B value is the same as in Hybrid Hash Join [3]). If the join key is string-value, however, then it is harder to choose the cut point. Histogram information could be used to choose it, so this new join algorithm needs to rely on the assumption that the system knows the distribution of the data. This introduces a bit more complexity for this algorithm versus the Hybrid Hash Join.

5.3.2 Random I/O

The random I/O issue discussed in the sections on G-Join performance with Input 2 is crucial to G-Join. We found G-Join's reduced performance (relative to Hybrid Hash Join) because of random reads to be quite surprising. It took us a while to figure out why the random reads for G-Join were not performance duals of the random writes involved when Hybrid Hash Join partitions the data in its first place. As we discussed before, even if Hybrid Hash Join and Hybrid G-Join each involve almost the same amount of random I/Os, their different kinds of random I/Os (i.e., random reads and random writes) turn out to make a big difference. To solve this puzzle of why, we first tried to run Hybrid G-Join and Hybrid Hash Join on the Linux "Memory File System" to eliminate I/O altogether; we then compared their execution times. We found that their two join times were almost the same. This showed us the difference in timing between them is not due to CPU cost, but instead to I/O cost. We then focused on the issue of random I/Os, running experiments with different combinations of sorted/unsorted inputs (e.g., Table 5.2), finally proving that random reads are more expensive than random writes due to the fact that reads are

blocking and non-overlapped whereas writes can be asynchronous due to file system and disk cache behavior.

5.3.3 In-Memory Join Method

When we first tried to implement the in-memory join method for G-Join to join the page of Input 2 with the in-memory pages of Input 1, we chose Merge Join as the join method. The intention was that, as the buffer list for each run of Input 1 is sorted, it is beneficial to make use of the sorted property to join. However, this implementation turned out to involve hundreds of millions of comparisons in the experiment for the 1.0X data set, making Hybrid G-Join take more time as the amount of memory assigned increases. We looked into this behavior and found out that using Merge Join will lead to G-Join joining each run of Input 1 with all of Input 2. This is because each page from Input 2 needs to join with each in-memory run of Input 1, and over time, this means each page from Input 1 will be merge-joined with every page of Input 2 – essentially resulting in a number of comparisons roughly equivalent to that of a page-based Nested Loops Join. Using Merge Join as the in-memory join method therefore turned out to be too expensive. To reduce this join CPU cost, we implemented a hash table that can efficiently handle Input 1 pages coming and going, and switched to using Hash Join as the in-memory join method (as briefly mentioned in Section 4.2.4).

6. Conclusion

In this thesis, we have reviewed and experimentally studied a new join algorithm, G-Join. A detailed design of the G-Join algorithm was proposed and implemented in the context of UCI's Hyrax data-intensive computing platform. This implementation of G-Join was then evaluated by analyzing its execution time, its buffer pool behavior, and other behavioral characteristics such as its response to being fed sorted or nearly sorted inputs. The G-Join algorithm was compared to

implementations of Grace Hash Join and Hybrid Hash Join, two of the "gold standard" methods for performing large ad hoc equijoins.

The objective of the G-Join design is to provide comparable performance to hash-based join methods for large joins of unordered inputs, given similar memory budgets, while also being able to exploit ordered inputs like Merge Join does. The first major finding of this thesis was that the performance of G-Join is negatively affected by random reads from the larger input (i,e., Input 2). That is, when Input 2 is unsorted, reading its next page guided by priority queue C involves random reads due to switching between runs of Input 2. This turns out to be much more expensive than the random writes needed in Hybrid Hash Join, as reads are blocking and non-overlapped whereas writes can be asynchronous due to file system and disk cache behavior. The second major finding was that the number of in-memory pages required per run of the smaller input (i.e., Input 1) can be more than the initially expected two pages per run. We saw in experiments with uniformly distributed TPC-H data that the maximum number of in-memory pages for Input 1 could sometimes be nearly three pages per run, especially before we adjusted the G-Join algorithm to address the "last run" problem.

While G-Join has not yet performed as well as we hoped at the outset, there is ample room for future work on improving as well as further testing its performance. We plan to try adding asynchronous reads to G-Join to see whether (and how much) asynchrony and next-page forecasting can help to alleviate its random read challenges. We also plan to compare G-Join with Merge Join and Indexed Nested Loops Join, and we plan to try using tournament sort to generate sorted runs of twice the memory size per the original G-Join design; the latter may especially help in cases involving multi-way join queries (i.e., where there can be sequences of several G-Joins). Last but not least, we also plan to analyze and experiment with G-Join to see how it behaves when given inputs with different degrees of join column value skew.

References

1. Ramakrishnan, Raghu and Gehrke, Johannes. Database Management Systems 3rd Edition. 2002.

2. Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. 1979. Access path selection in a relational database management system. In Proceedings of the 1979 ACM SIGMOD international Conference on Management of Data (Boston, Massachusetts, May 30 - June 01, 1979).

3. Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. 3, 1986, ACM Trans. Database Syst., Vol. 11, pp. 239-264.

4. Graefe, Goetz, Li, Guangqiang(Aries), Borkar, Vinayak and Carey, Michael J. A New Join Algorithm. 2010. In preparation.

5. Graefe, Goetz. Query Evaluation Techniques for Large Databases. 2, 1993, ACM Computing Surveys, Vol. 25, pp. 73-170.

6. DeWitt, David J., Naughton, Jeffrey F. and Burger, Joseph. Nested Loops Revisited. 1993, In Proceedings of the Symposium on Parallel and Distributed Information Systems, pp. 230-242.

7. Graefe, Goetz. Implementing Sorting in Database Systems. 3, 2006, ACM Computing Surveys, Vol. 38.

8. Knuth, D. The Art of Computer Programming. 1973. Vol. 3.

9. Borkar, Vinayak and Carey, Michael. Hyrax: A New Foundation for Data-Parallel Computation. 2010. In preparation.

10. TPC-H. [Online] http://www.tpc.org/tpch/.

11. OpenJDK. [Online] http://openjdk.java.net/.

12. Haas, Laura M., et al. SEEKing the Truth About Ad Hoc Join Costs. 3, 1997, The VLDB Journal, Vol. 6, pp. 241–256.

13. ASTERIX. ASTERIX Website. [Online] 2010. http://asterix.ics.uci.edu/.

14. Chaudhuri, S. 1998. An overview of query optimization in relational systems. In Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (Seattle, Washington, United States, June 01 - 04, 1998).

15. Borkar, Vinayak R., et al.ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving World Models. In preparation.

Appendix A

An Example for G-Join

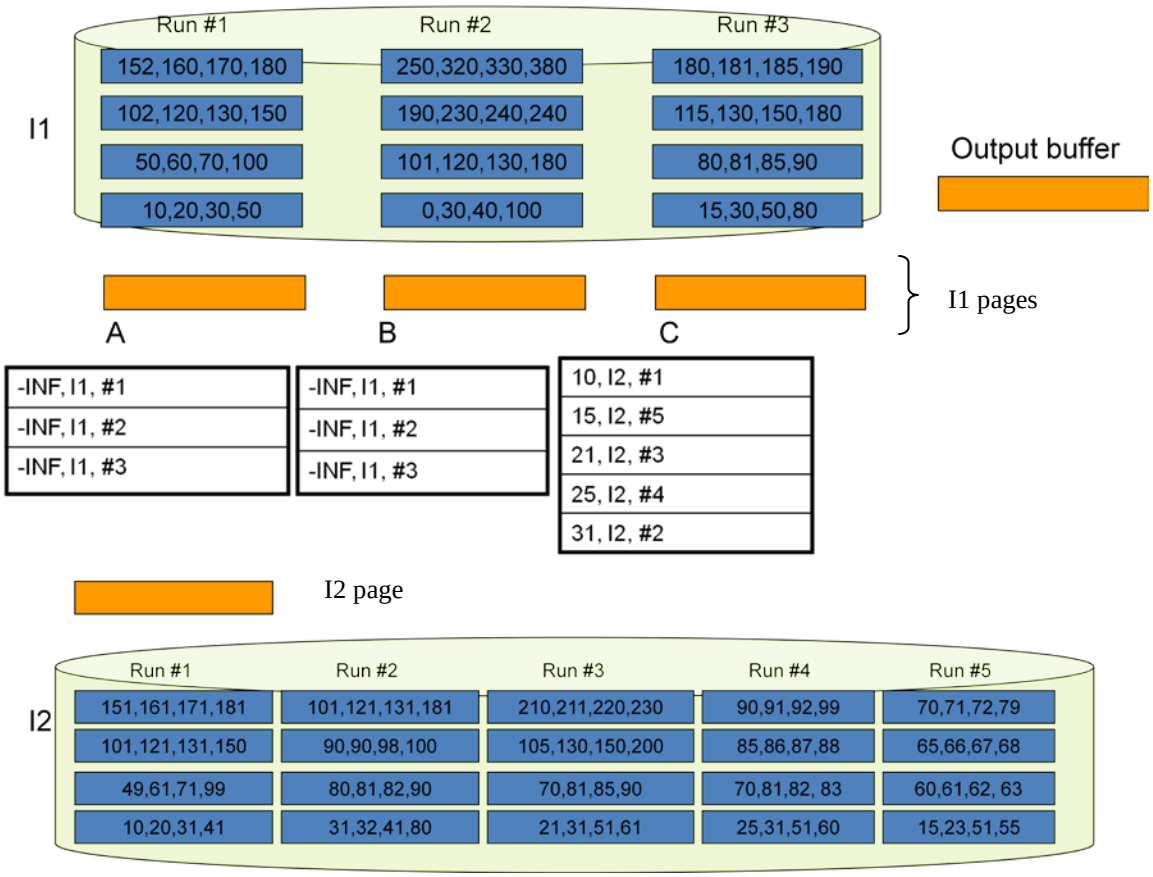
I1: Input 1

I2: Input 2

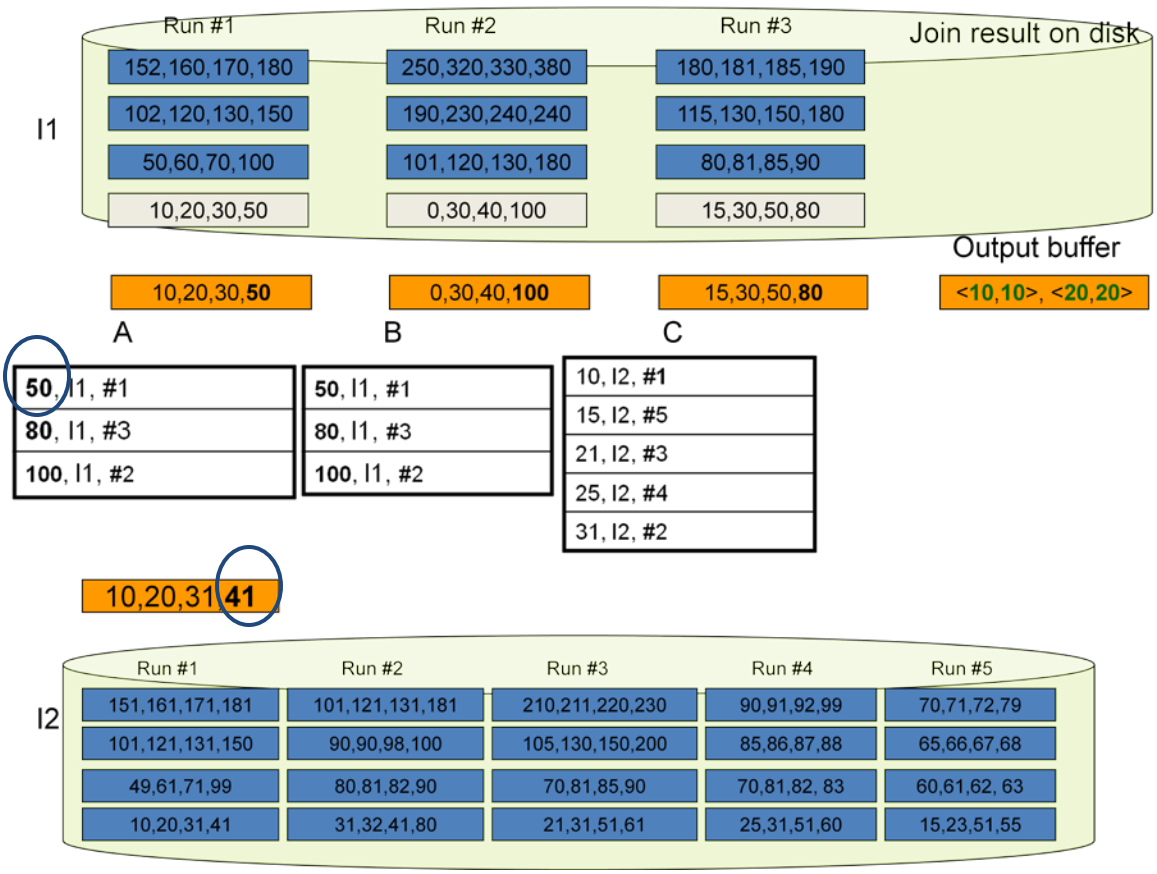
Assumptions: a) 4 keys per page; b) 3 runs of I1; c) 5 runs of I2.

A, B, C are priority queues. Each entry of the priority queue contains key value, input#, and run#.

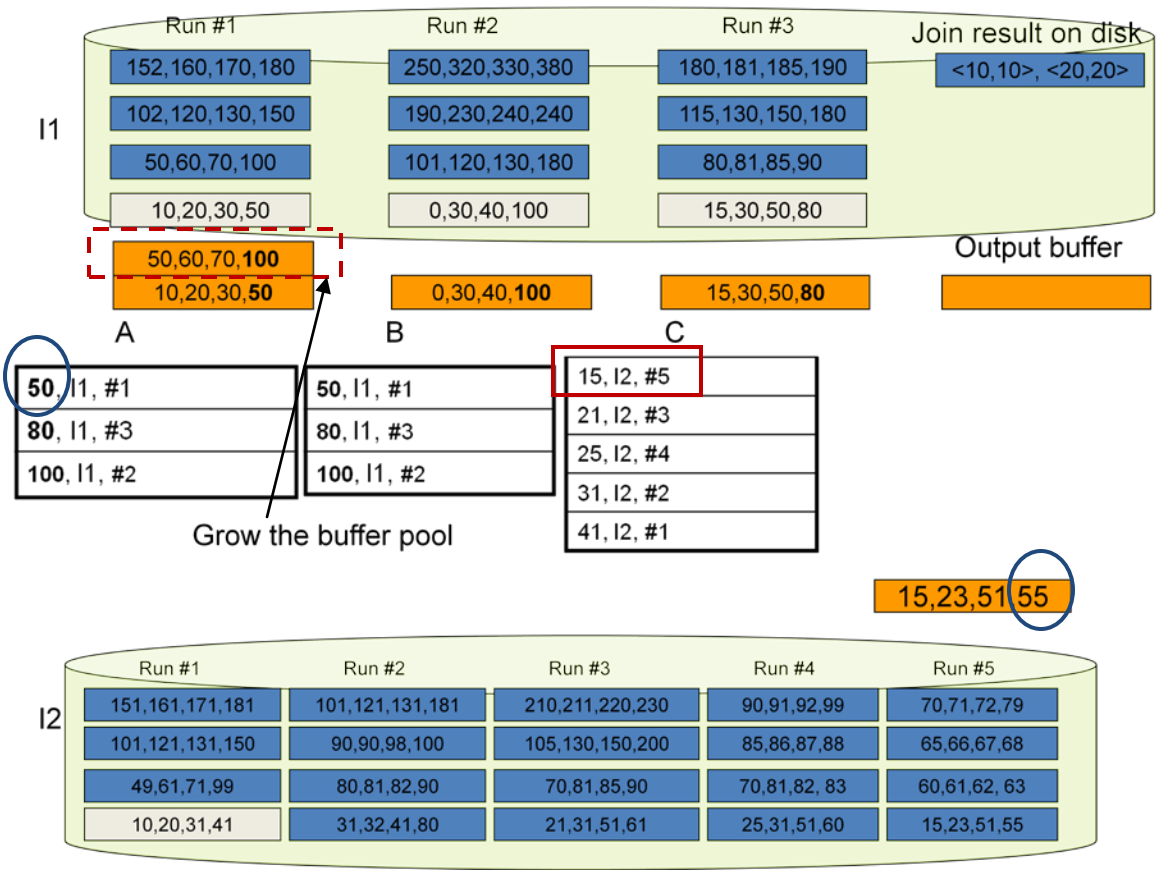
For example, [10, I2, #1] indicates that this entry has key value 10, and is from run#1 of I2.



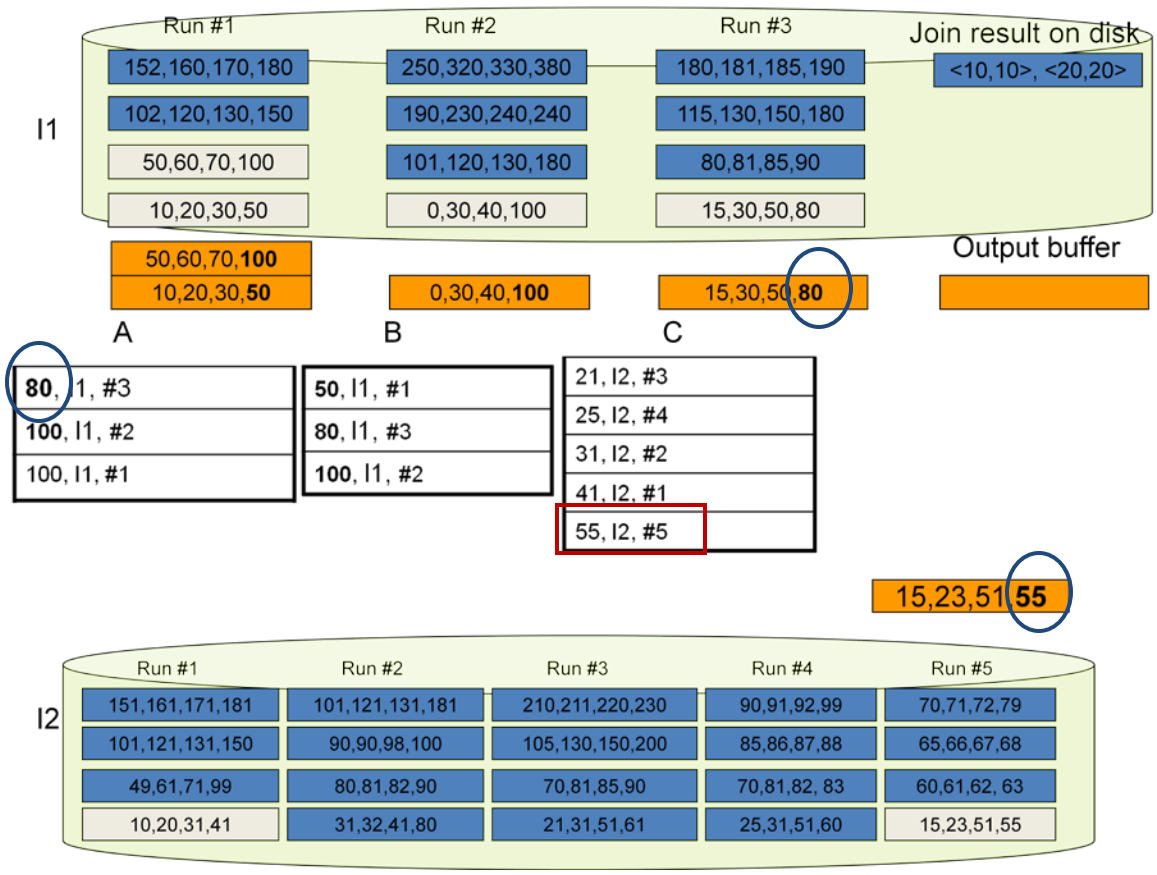
This snapshot is the initial state of G-Join. Sorted runs of I1 and I2 are on disk. Priority queue C is initialized with the low key information of each I2 run remembered during run generation. Priority queues A and B are initialized with –INF for each I1 run.



G-Join loads the first page of each I1 run into the buffer pool and updates priority queues A and B. Then G-Join loads an I2 page from Run#1 guided by priority queue C. Since the high key (41) of I2 page is less than the top entry (50) of priority queue A, the I2 page is immediately eligible to join.

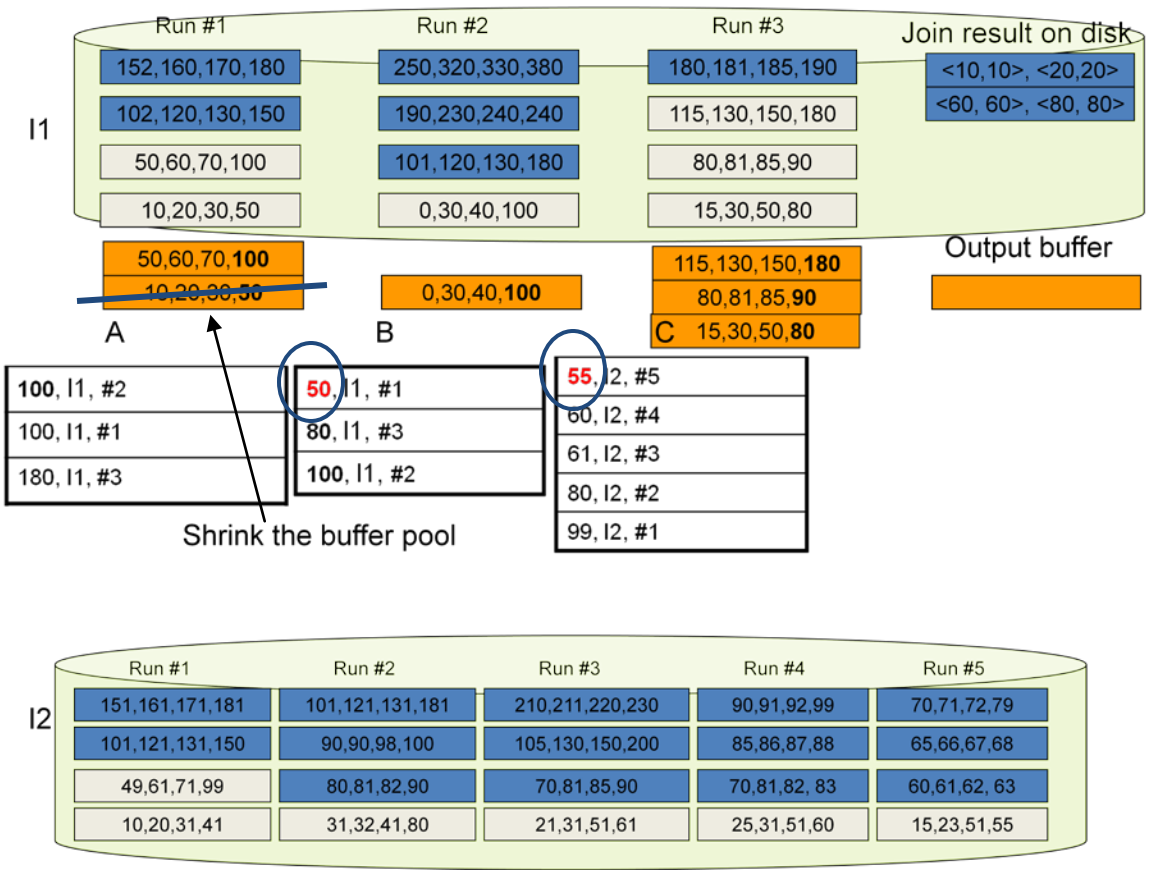


After the join result in the output buffer is flushed to disk, G-Join loads the next page of Run#5 from I2 guided by priority queue C. Because the high key (55) of the I2 page is greater than the top entry (50) of priority queue A, G-Join loads one additional I1 page from Run#1 guided by priority queue A and then updates priority queue A.

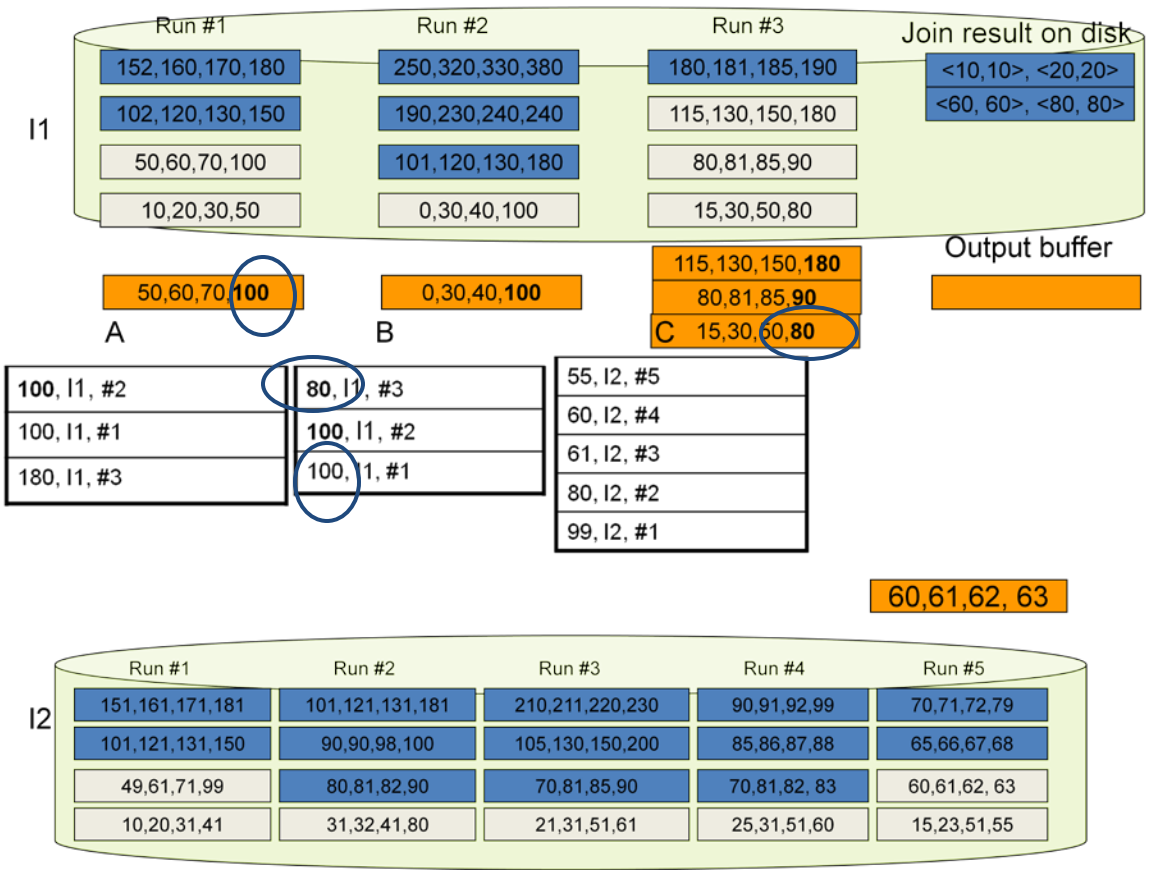


Priority queue C is updated with approximate low key value for the next page of Run#5 from I2. After growing the buffer pool, the lowest high key in the buffer pool of I1 is updated to 80 (top entry of priority queue A) – greater than the high key (55) of the I2 page– the I2 page is now eligible to join with the pages in the buffer pool.

In this way, the G-Join algorithm continues, growing the buffer pool when needed. Next we will see when/how to shrink the buffer pool of I1.



At this moment, the top entry (55) of priority queue C is greater than the top entry (50) of priority queue B, and then the lowest I1 page of Run#1 is dropped from the buffer pool guided by priority queue B.



After shrinking the buffer pool, priority queue B is updated with the high key (100) of the lowest (oldest) in-memory I1 page of Run#1(the run where a page was dropped previously). The high key (80) of the lowest (oldest) in-memory page from Run#3 of I1 now becomes the top entry of priority queue B.

The G-Join algorithm continues until either priority queue B or priority queue C is empty.