A generalized join algorithm

Goetz Graefe
Hewlett-Packard Laboratories

Abstract

Database query processing traditionally relies on three alternative join algorithms: index nested loops join exploits an index on its inner input, merge join exploits sorted inputs, and hash join exploits differences in the sizes of the join inputs. Cost-based query optimization chooses the most appropriate algorithm for each query and for each operation. Unfortunately, mistaken algorithm choices during compile-time query optimization are common yet expensive to investigate and to resolve.

Our goal is to end mistaken choices among join algorithms by replacing the three traditional join algorithms with a single one. Like merge join, this new join algorithm exploits sorted inputs. Like hash join, it exploits different input sizes for unsorted inputs. In fact, for unsorted inputs, the cost functions for recursive hash join and for hybrid hash join have guided our search for the new join algorithm. In consequence, the new join algorithm can replace both merge join and hash join in a database management system.

The in-memory components of the new join algorithm employ indexes. If the database contains indexes for one (or both) of the inputs, the new join can exploit persistent indexes instead of temporary in-memory indexes. Using database indexes to match input records, the new join algorithm can also replace index nested loops join.

Results from an implementation of the core algorithm are reported.

1 Introduction

Non-procedural queries and physical data independence both enable and require automatic query optimization in a SQL compiler. Based on cardinality estimation, cost calculation, query rewrite, algebraic equivalences, plan enumeration, and some heuristics, query optimization chooses access paths, join order, join algorithms, and more. In most cases, these compile-time choices are appropriate, but poor choices often cause poor performance, dissatisfied users, and disrupted workflows in the data center. Investigation and resolution of intermittent performance problems are very expensive.

Our research into robust query processing has led us to focus on poor algorithm choices during compile-time query optimization. In order to avoid increasing complexity and sophistication during query optimization, e.g., by run-time feedback and statistical learning [MLR 03], our efforts center on query execution techniques.

The new join algorithm introduced here is a result of this research. Its design goal is a viable single replacement for all three traditional join algorithms by matching the performance of the best traditional algorithm in all situations. If both join inputs are sorted, the new algorithm must perform as well as merge join. If only one input is sorted, it must perform as well as the better of merge join and hash join. If both inputs are unsorted, it must perform as well as hybrid hash join. If both inputs are very large, it must perform as well as hash join with recursive partitioning or merge join and external merge sort with multiple merge levels. Finally, if one input is small, the new join algorithm must perform as well as index nested loops join exploiting a temporary or permanent index for the large input.

Table 1 summarizes the input characteristics exploited by index nested loops join, merge join, hybrid hash join, and the new join algorithm. Rather than merely performing a run-time choice among the traditional join algorithms, it combines elements from these algorithms and from external merge sort. Therefore, we call it "generalized join algorithm" or abbreviated "g-join."

	INLJ	MJ	HHJ	GJ
Sorted input(s)		+		+
Indexed input(s)	+			+
Size difference			+	+

Table 1. Join algorithms and exploited input properties.

With one or two sorted inputs, g-join avoids run generation and merging, instead exploiting the sort orders in the inputs. For indexed inputs, it exploits the index either as a source of sorted data or as a means of efficient search.

For unsorted inputs, g-join employs run generation quite like external merge sorts for a traditional merge join. Unlike external merge sort, g-join avoids all or most merge steps, even leaving more runs than can be merged in a single merge step. Like hybrid hash join, it divides memory into two regions, one for immediate join processing and one for handling large inputs. If the size of the small join input is similar to the memory size, most memory is assigned to the first region; if the size of the small input is much larger, most or all memory is assigned to the second region. As in hybrid hash join, the size of the large join input does not affect the division of memory into regions.

The following sections review the traditional join algorithms (Section 2) and then introduce g-join (Section 3). Algorithm details for unsorted inputs of various input sizes and unknown input sizes (Section 4) are followed by answers for the "usual questions" about any new join algorithm or new algorithmic variant (Section 5). Based on those details and answers, replacement of the traditional join algorithms is discussed in depth (Section 6). Two partial prototype implementations permit an initial performance evaluation of g-join (Section 7). The last section offers our summary and conclusions from this effort so far.

2 Prior work

G-join competes with the well-known (index) nested loops join, merge join, and (hybrid) hash join algorithms, which are reviewed in detail elsewhere [G 93]. The diag-join algorithm [HWM 98] can serve as preprocessing step for most join algorithms including g-join. Moreover, the merge algorithm of g-join might seem similar to the diag-join algorithm as both exploit sorting and a buffer pool with sliding contents. The algorithms differ substantially, however, because diag-join only applies in the case of foreign key integrity constraint whereas g-join is a general join algorithm, because diag-join depends on equal insertion and scan sequences whereas g-join does not, and because diag-join is inherently heuristic whereas g-join guarantees a complete join result.

In addition to join algorithms, prior research has investigated access paths, in particular index usage – from covering indexes (also known as index-only retrieval) to index intersection (combining multiple indexes for the same table) and query execution plans with dynamic index sets [MHW 90]. Inasmuch as such access plans require set operations such as intersection, g-join serves the purpose; otherwise, source data access in tables and indexes is not affected by g-join.

Prior research also has investigated join orders in the contexts of dynamic programming [SAC 79], very large queries [IK 90], and dynamic join reordering [AH 00, BBD 05, LSM 07]. Most of those research directions and their results are orthogonal to g-join and its relationship with the traditional join algorithms.

The following sections assume a join operation with an equality predicate between the two join inputs. Special cases such as joining a table with itself, joining on hash values, etc. are feasible but ignored. Similarly, we ignore join operations without equality predicates.

3 The new join algorithm

G-join is a new algorithm; it is not a run-time switch among algorithms, e.g., the traditional join algorithms. It is based on sorted data and thus can exploit sorted data stored in b-tree indexes as well as sorted intermediate results from earlier operations in the query execution plan. For unsorted inputs, it employs run generation very similar to a traditional external merge sort. Thereafter, it avoids most or all of the merge steps in a traditional merge sort. Moreover, the behavior and cost function of recursive and hybrid hash join have guided the algorithm design for unsorted inputs. Nonetheless, g-join is based on merge sort and is not a variant of hash join, e.g., partitioning using an order-preserving hash function. Delaying all details to subsequent sections, the basic idea is as follows.

For unsorted inputs, run generation produces runs like an external merge sort, but merging these runs can be omitted in most cases. Any difference in the sizes of the two join inputs is reflected in the count of runs for each input, not in the sizes of runs.

With sufficient memory and a sufficiently small number of runs from the smaller input, join processing follows roughly (but not precisely) the sort order of join key values. A buffer pool holds pages of runs from the small input. Pages with higher keys successively replace pages with lower keys. A single buffer frame holds pages of runs from the large input (one page at a time) while such pages are joined with the pages in the buffer pool. In other words, during join processing, most memory is dedicated to the small input and only a single page is dedicated to the large input. With respect to memory management, the buffer pool is reminiscent of the in-memory hash table in a hash join, but its contents turns over incrementally in the order of join key values.

If merging is required, the merge depth is kept equal for both inputs. This is rather similar to the recursion depth of recursive hash join and quite different from a merge join with two external merge sorts for inputs of different sizes. For fractional merge depths – similar to hybrid hash join – memory is divided into segments for immediate join processing and for buffers for temporary files. Thus, g-join requires memory and I/O for temporary files in very similar amounts as recursive and hybrid hash join. Even bit vector filtering and role reversal are possible, as are integration of aggregation and join operations.

If one or both inputs are sorted, run generation and merging can be omitted. With two sorted inputs, the algorithm “naturally simplifies” to the logic of a traditional merge join. If the small input is sorted, the buffer pool holds only very few pages, very similar to the “back-up” logic in merge join with duplicate key values. If the large input arrives sorted, g-join joins its pages with the buffer pool contents by strictly increasing join key values.

If one or both of the inputs is indexed, g-join exploits available indexes in its merge logic. If one input is tiny and the other input is huge, the merge logic skips over most data pages in the huge input, thus mimicking traditional index nested loops join. If the small input is indexed and the index can be cached in memory, there is no need for sorting the large input – like hash join as well as the non-traditional mode of index nested loops join.

4 Unsorted inputs

Many of the algorithm's details can best be explained case by case. This section focuses on inner joins of two unsorted, non-indexed inputs with practically no duplicate key values, with uniform (non-skewed) key value distributions, and with pages holding multiple records and thus non-trivial key ranges. Later sections relax these assumptions.

As the size of the small input is crucial to achieving join performance similar to recursive and hybrid hash join for unsorted inputs, the discussion divides cases by the size of the small input relative to the memory size. In all cases, the large input may be larger than the small input by a few percent or by orders of magnitude. The core algorithm that most other cases depend on is covered in Section 4.2.

In order to explain g-join step-by-step, Sections 4.1 through 4.5 assume accurate a priori knowledge of the size of input R. Section 4.6 relaxes this assumption.

The following descriptions assume that compile-time query optimization anticipated that input R will be smaller than input S. Therefore, input R is consumed first and drives memory allocation and run generation.

The memory allocation is *M* (pages or units of I/O) and the maximal fan-in in merge steps as well as the maximal fan-out in partitioning steps is *F*. *F* and *M* are equal except for their units and a small difference due to a few buffers required for asynchronous I/O etc. If *M* is around 100 or higher, this difference is not significant and usually ignored.

During merge steps, read operations in run files require random I/O. Large units of I/O (multiple pages) are a well-known optimization for external merge sort and for partitioning, e.g., during hash join and hash aggregation. The same optimization also applies to the runs and to the merge operations described in 4.2.1.

4.1 Case R ≤ M

The simplest case is a small input that fits in memory, i.e., $R \leq M$. No run generation is required in this case. Instead, g-join retains R in memory in an indexed organization (e.g., a hash table, but it may be a b-tree), and then processes the pages of S one at a time.

With all temporary files avoided, the I/O cost of g-join in this case is equal to that of traditional in-memory hash join. When using the same in-memory data structure, the CPU effort of the two join algorithms is also the same.

4.2 Case R = F × M

The next case is the one in which Grace hash join [FKT 86] and hybrid hash join operate in the same way, with *F* pairs of overflow files, no immediate join processing during the initial partitioning step, and all memory required during all overflow resolution steps.

In this case, g-join creates initial runs from both inputs R and S. With replacement selection for run generation, the number of runs from input R is $F/2+1$. Even if the number of runs from input S is much larger than the maximal merge fan-in *F*, no merging is required. Instead, inputs R and S are joined immediately from these runs. Practically all memory is dedicated to a buffer pool for pages of runs from input R. Input S requires only a single buffer frame as only one page at a time is joined with the contents of the buffer pool.

4.2.1 Page operations

Careful scheduling governs read operations in runs from inputs R and S. At all times, the buffer pool holds some key range of each run from input R. The intersection of those

key ranges is the "immediate join key range." If the key range in a page from input S falls within the immediate join key range, the page is eligible for immediate join processing.

The schedule focuses on using, at all times, the minimal buffer pool allocation for pages of the small input R. It grows its memory allocation only when necessary in order to process pages from the large input S one at a time, shrinks the buffer pool as quickly as possible, and sequences pages from the large input S for minimal memory requirements.

The minimal memory allocation for the buffer pool requires one page for each run from input R. Its maximal memory allocation depends on key value distributions in the inputs, i.e., distribution skew and duplication skew. With uniform key value distributions and moderate amounts of duplicate key values, about two pages for each run from input R should suffice. Two pages for each of $F/2+1$ runs amount to M pages. In other words, g-join can perform the join immediately after run generation in this case, independently of the size of input S. Thus, if indeed two pages per run from input R suffice, memory needs and I/O effort of g-join match those of hash join.

Algorithm overview

Figure 1 illustrates the core algorithm of g-join. Various pages (double-ended arrows) from various runs cover some sub-range of the domain of join key values (dotted horizontal arrow). Some pages of runs from input R are resident in the buffer pool (solid arrows) whereas some pages have already been expelled or not yet been loaded (dashed arrows). The pages in the buffer pool define the immediate join key range (dotted vertical lines). It is the intersection of key ranges of all runs from input R. Some pages of runs from input S are covered by the immediate join key range (solid arrows) whereas some have already been joined or cannot be joined yet (dashed arrows). Differently from the diagram, there usually are more runs from input S than from input R. Again, at any one time, memory holds multiple pages of each run from input R but only one page from one of the runs from input S.

In Figure 1, the buffer pool contains 2 pages each from runs 1 and 3 of input R and 3 pages from runs 2. In the illustrated situation, the next action joins a page from run 2 of input S with the pages in the buffer pool, whereupon the buffer pool may shrink by a page from run 2 of input R.

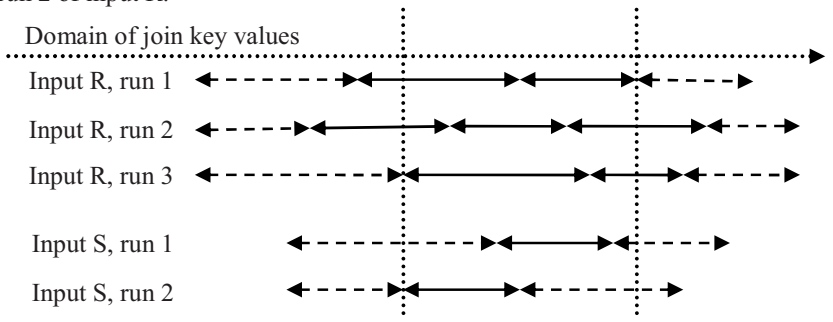

Figure 1. Runs, pages, buffer pool, and the immediate join key range.

Data structures

The immediate join key range expands and contracts as it moves through the domain. Multiple priority queues guide the schedule of page movements. These priority queues require modifications when the buffer pool reads or drops pages of input R and when a page

of S joins with the pages in the buffer pool. All priority queues are sorted in ascending order such that top entry holds the smallest key value. The priority queues are named A through D:

A. This priority queue guides how the buffer pool grows. Each run from the small input R has one entry in this priority queue at all times. Its top entry indicates next page to load into memory. The sort key is the high key of the newest page in the buffer pool for a run.
B. This priority queue guides how the buffer pool shrinks. Each run from input R has one entry in this priority queue at all times. Its top entry indicates the next page to drop from memory. The sort key is the high key of the oldest page in the buffer pool for a run.
C. This priority queue guides how the buffer pool shrinks. Each run from input S has an entry in this priority queue. Its top entry indicates the next page to join from the large input S. The sort key is the low key value in the next page on disk.
D. This priority queue guides how the buffer pool grows. Each run from input S has an entry in this priority queue. Its top entry indicates the next page to schedule from input S. The sort key is the run with the high key value in the next page on disk.

Priority queue A is similar in function and size (*F/2* entries) to a priority queue guiding page prefetch ("forecasting") in a traditional external merge sort. Priority queue B could have an entry for each page in memory rather than for each run from input R. In that case, it would be similar in function and size (*M* entries) to a priority queue used for page replacement in a buffer pool. Priority queues C and D are similar in size (*S/(2M)+1* entries) to that of a priority queue guiding a traditional merge sort to merge in each step the smallest ones among all remaining runs, which is the fastest way to reduce the number of runs.

Priority queues C and D employ information from pages not yet read during the join process. With a very moderate loss in predictive precision, priority queue C can use the highest key value already seen instead of the lowest future key value.

Finally, it is possible to omit priority queue D and schedule pages of input S entirely based on priority queue C. This algorithm variant does not require a larger maximal buffer pool, although it may require a large buffer pool over longer periods.

Algorithm

The algorithm initializes the buffer pool and priority queues A and B with the first page of each run from input R. Priority queues C and D initially hold information about the first page of each run from input S. The algorithm continues step by step until all pages of all runs from input S have been joined, i.e., priority queues C and D are both empty.

Each step tests the top entries of priority queues C and then D whether they can be joined immediately. If so, it reads the appropriate page of input S and joins it to the pages of input R in the buffer pool. It then replaces the page of input S in priority queues C and D with the next page from the same run from input S. If the end of the run is reached, the run is removed from priority queues C and D. If the replaced page used to be the top entry of priority queue C, the buffer pool may drop some pages, guided by priority queue B.

Otherwise (if the top entries of priority queues C and D cannot by joined immediately), the buffer pool grows by loading some additional pages from input R. Priority queue A guides this growth until the top entry in priority queue D can be joined immediately.

The overall complexity of the priority queue operations is modest: each page in all runs from inputs R and S goes through precisely two priority queues. Replace and erase

operations are required in these priority queues. Tree-of-loser priority queues [K 73] can implement these operations with a single leaf-to-root pass.

The actual I/O operations as well as operations on individual records will likely exceed the CPU cost for the priority queues. Operations on individual records include insertion and deletion in an in-memory index when pages of input R enter and exit the buffer pool as well as search operations in this index to match records of input S. These insertions, deletions, and searches are similar in cost and complexity to the equivalent operations in a hash join and its in-memory hash table.

Prototype implementation

Our first prototype implementation of the g-join logic employs priority queue B to guide shrinking the buffer pool. It does not merge key values into an in-memory b-tree index. The prototype has run-time switches that control whether or not priority queue D is used and whether priority queue C is sorted on the highest value already seen or the lowest value not yet seen. Priority queue D is not used in any of the experiments reported below.

For a uniform distribution of join key values and a uniform distribution of run sizes, it requires about two pages of input R, as discussed in detail later. Two inputs with 100 and 900 runs of 1,000 pages, i.e., a total of 1,000,000 pages, can be processed in the priority queues in less than 1 second using a laptop with a 2.8 GHz CPU. Clearly, 1,000,000 I/O operations take more time by 3-4 orders of magnitude. Maintenance of the priority queues takes a negligible amount of time.

4.2.2 Record operations

As pages of runs from input R enter and exit the buffer pool, their records must be indexed in order to permit fast probing with join key values from records of input S.

A local index per page of input R is not efficient, as each record of input S would need to probe as many indexes as there are pages in the buffer pool. This can substantially increase the cost of probing compared to a global index for all records in the buffer pool.

A global index must support not only insertions but also deletions. After a new page has been read into the buffer pool, its records are inserted into the global index; before a page is dropped from the buffer pool, its records are removed from the global index.

The implementation of the global index can use a hash table or an ordered index such as a b-tree. B-tree maintenance can be very efficient if the b-tree contains only records in the key range eligible for immediate join processing. For efficient insertion, the runs from input R can be merged (as in a traditional merge sort) and then appended to the b-tree index. For efficient deletion, entire leaf pages can be removed from the b-tree.

On the other hand, a hash table may support more efficient join processing than a b-tree index. Even if every record in input R eventually joins with some records of input S, each page of input S join with only a few records of input R. Thus, a lot of skipping and searching is required in a global b-tree index.

Hash table implementations with efficient insertion and deletion therefore seem the most appropriate data structure for in-memory join processing, i.e., the buffer pool with records from input R with individual pages of runs from input S.

4.3 Case $M < R < F \times M$

This case falls between the prior two cases, i.e., the case in which hybrid hash join shines. During the initial partitioning step of hybrid hash join, some memory serves as out-

put buffers for the partition overflow files and some memory holds a hash table and thus enables immediate join processing. If the small input is only slightly larger than the available memory allocation, most of the join result is computed during the initial partitioning step and only a small fraction of both join inputs incurs I/O costs. If the small input is much larger than the available memory allocation, hardly any join result is computed immediately and a large fraction of both join inputs spills to overflow partitions.

G-join also employs a hybrid of two algorithms. It divides memory among them and employs the same division of memory as hybrid hash join. A fraction of memory equal to the size of the hash table in hybrid hash join enables immediate join processing as in Section 4.1. Thus, while consuming the join inputs for the first time, g-join computes the same fraction of the join result as hybrid hash join. The remaining memory enables run generation quite similarly to the algorithm of Section 4.2.

The less memory run generation uses, the smaller the resulting runs are. The goal is to produce *F/2* runs from input R, because the algorithm of Section 4.2 can process *F/2* runs in a single step. Interestingly, the required formulas for the memory division are equal in hybrid hash join and g-join.

Specifically, hybrid hash join requires at least K overflow partitions and output buffers, with K derived as follows. K partitions can hold $K \times M$ pages from input R. This memory allocation leaves $M-K$ pages for the hash table and immediate join processing. In order to process all input in a single step, input R must fit within the hash table plus these partitions, i.e., K must satisfy $R \leq (M-K)+K \times M$. Solving for K gives $K \geq (R-M) \div (M-1)$ or $K = ceiling((R-M) \div (M-1))$.

G-join uses the same division of memory as hybrid hash join. Immediate join processing uses $M-K$ pages and K pages are used for preparation of temporary files. In g-join, these pages serve as workspace for run generation. Thus, $R-(M-K)$ pages are the input to run generation, with $R-(M-K) \leq K \times M$ because $R \leq (M-K)+K \times M$.

With replacement selection using a workspace of K pages, the average run size is $2K$. The resulting count of runs is $(K \times M)/(2K)=M/2$. This is precisely the number of runs from the smaller input R that can be processed by the algorithm described in Section 4.2.

As in hybrid hash join, immediate in-memory join processing must be assigned a specific set of key values. In hybrid hash join, an appropriate range of hash values is assigned to the in-memory partition. G-join can employ a similar hash function or it can simply retain the lowest key values from input R. For the latter variant, the required data structure and algorithm is very similar to that of an in-memory "top" algorithm [CK 97], i.e., a priority queue. This design choice is best with respect to producing sorted output suitable for the next operation in the query execution plan.

While g-join consumes input R, it employs a priority queue to determine the key range eligible for immediate join processing. While g-join consumes input S, it employs a hash table for join processing. The hash table contains precisely those records that remained in the priority queue after consuming input R.

In summary, g-join running in hybrid mode divides memory like hybrid hash join, retains the same fraction of the smaller input R in memory, performs the operations required for in-memory just as efficiently as hybrid hash join, and produces nearly sorted output.

4.4 Case $R = F^2 \times M$

In this case, hash join requires precisely two partitioning levels. Assuming a perfectly uniform distribution of hash values, two partitioning levels with fan-out F produce over-

flow files from input R equal in size to the available memory, enabling in-memory hash join for each partition. Suitable overflow partitions from input S require the same two partitioning levels, independent of the size of input S and its partition files.

G-join similarly moves each input record through two temporary files. After run generation produces $R \div (2M) = (F^2 \times M) \div (2M) = F^2/2$ initial runs for input R, a single merge level with merge fan-in F reduces the count of runs to $F/2$. Input S also goes through two levels, namely run generation and one level of merging. Thereafter, the algorithm of Section 4.2 applies, independent of the size of input S and its number of remaining run files.

4.5 Case $F \times M < R < F^2 \times M$

In this case, a hash join requires more than one partitioning level but less than two full partitioning levels. The partial level is realized by hybrid hash join when joining partitions produced by the initial partitioning step.

G-join first aims to produce $F^2/2$ runs from input R by dividing memory similar to the algorithm in Section 4.3. If the size of R is close $F \times M$, most memory is used for immediate join processing during this phase. If the size of R is close to $F^2 \times M$, very little memory is used for immediate join processing and most memory is used as workspace for run generation. More specifically, the calculation $K = ceiling((R - M) \div (M - 1))$ of Section 4.3 is replaced with $K = ceiling((R/F - M) \div (M - 1))$ to account for one additional merge level.

After this initial hybrid step, g-join merges all runs once, reducing the number of runs from input R by a factor of F to $F/2$. All runs from input S are also merged once with a fan-in F. The final step applies the algorithm of Section 4.2 to the remaining runs.

4.6 The general case

The preceding descriptions assume precise a priori knowledge of the size of input R. Dropping this assumption, the following discussion assumes that actual run-time sizes are not known until the inputs have been consumed by the join algorithm, that input R is consumed before input S, and that R is smaller than S. Should S be smaller than R, role reversal is possible after run generation for both inputs but it is not discussed further.

In order to calibrate expectations, it is worthwhile to consider the behavior of hybrid hash join under these assumptions. The preceding discussions of hybrid hash join assume perfectly uniform distributions of hash values. For a perfect assignment of hash values to the in-memory hash table and to the overflow partitions, hybrid hash join also requires prior knowledge of the desired size of the in-memory hash table, i.e., of the precise size of the build input. Without this knowledge, hash join loses some degree of efficiency. Different designs and implementations of hash join suffer in different places. In all cases, changing the size of the in-memory hash table and its hash buckets is quite complex.

G-join, with two unsorted inputs of unknown sizes, first consumes the input anticipated to be the smaller one. If that input R fits in memory (case $R \leq M$), run generation for input S can be avoided entirely, and g-join performs similarly to an in-memory hash join.

Otherwise, the algorithm divides memory between immediate join processing and run generation. With an unknown input size, the best current estimate is used. This estimate may change over time, and the memory allocation is adjusted accordingly. Note that such an adjustment is easily possible in g-join.

The most conservative variant of g-join prepares for two huge inputs, i.e., run generation uses all available memory. If the first input turns out to be small and fit in memory, run

generation for the second input is skipped in favor of immediate join processing. Otherwise, run generation for both inputs is completed. In this variant, g-join performs rather like Grace hash join [FKT 86] without dynamic de-staging [NKT 88].

The memory allocation at the end of consuming input R is preserved throughout run generation and immediate join processing for input S. After run generation for input S, if one of the two inputs has produced no more than *F/2* runs, final join processing can commence immediately without any intermediate merge steps.

Otherwise, runs from the smaller input are merged until *F/2* runs remain. Each merge step merges the smallest remaining runs, which reduces the number of remaining runs with the least effort [K 73]. Due to the effects of replacement selection, this will most likely affect the first and last initial runs, because the sizes of all other runs tend to be similar to the sizes of these two runs together. If run generation produces precisely *F/2+1* runs, merging the first and last runs produces *F/2* runs of similar size.

The merge policy also attempts to minimize the size of the largest run of input R left for final join processing. Thus, it might be useful to perform multiple merge steps with moderate fan-in rather than one merge step with maximal fan-in, even if doing so requires merging slightly more than the minimal data volume.

Next, g-join merges runs from the larger input. Again, each merge step merges the smallest remaining runs. Even with no other merge steps, it might be useful to merge the first run and the last run produced during run generation. In fact, it is often possible to merge the first and last runs immediately after the end of the input, i.e., while the last run is being formed. Merging continues until the smallest remaining run is at least as large as the largest remaining run from the smaller input. For unsorted inputs, this stopping condition leads to equal merge depth for both inputs. For join inputs of very different sizes, this is a crucial performance advantage of g-join when compared to merge join, very similar to the main advantage of hash join over merge join.

The crucial aspect is not the count of runs but their sizes. Ideally, the runs from input S should be of similar size as the runs from input R. More specifically, the smallest run of input S should be at least as large as the largest run of input R. Assuming reasonably uniform distributions of join key values, this ensures that a buffer pool of *M* pages suffices to join *F/2* runs from input R with any number of runs from input S, which is the final step in g-join for unsorted inputs of unknown size.

4.7 Summary for unsorted inputs

In summary, g-join processes two unsorted inputs about as efficiently as recursive hybrid hash join. This is true for input sizes from tiny to huge and for both known and unknown input sizes. In particular, g-join exploits inputs of very different sizes by limiting the merge depth for both inputs to that required by the smaller input, quite similar to the recursion depth in hash join. Moreover, g-join is able to divide memory between immediate join processing and preparation of temporary files, very similar to hybrid hash join in both memory allocation and performance effects.

G-join is based on sorting its inputs rather than on hash partitioning. It even produces the join result roughly in sorted order such that it might be useful in subsequent operations within the query execution plan. This and similar questions are discussed in the following section, and the issue whether g-join can substitute for the traditional join algorithms is considered thereafter.

5 The usual questions

This section discusses skew in the distribution of join key values, binary operations of the relational algebra other than inner join, complex queries with multiple join operations, and parallel query execution.

5.1 Skew

Skew can affect the performance of g-join in several ways. For example, extreme duplication of a single key value in the small input may temporarily force a very large buffer pool. A temporary file might be required, comparable to a buffer pool in virtual memory rather than real memory. In those extreme cases, both hash join and merge join effectively resort to nested loops join, usually using some form of temporary file and repeated scans.

In general, a buffer pool extended by virtual memory or an equivalent technique is one of two "water proof" methods for dealing with extreme cases of skew. The other one reduces both inputs R and S to a single run and then performs a merge join. Short of these methods, however, a variety of techniques may reduce the impact of skew on the performance of g-join. The following describes some of those.

Run generation may gather some statistics about the range and distribution of key values in each run. If skew is an issue, the merge step may process inputs R and S just a bit further than discussed so far. As a result, input R will have fewer than *F/2* runs remaining and the available memory allocation can support more than two buffer pool pages per run. Input S will have larger runs with a smaller key range per page, thus requiring fewer pages of input R in the buffer pool during join processing.

Even in the case of uniform distributions of join key values, merging runs from input S until the smallest run from input S is twice as large or even larger than the largest run from input R reduces the buffer pool requirements. Again, the key ranges in each page of input R and in each page of input S are crucial. If the pages from input S have a smaller key range on average, fewer runs from input R require multiple pages in the buffer pool at a time.

Rather than merging entire runs, it is also possible read individual pages from input S twice. If the buffer pool is at its maximal permissible size, cannot be shrunk, and no pages can be joined immediately, joining the low key range of some pages from input S might enable shrinking the buffer pool and then growing it again to extend the immediate join key range. Priority queue C can track key ranges already joined. If the key value in priority queue C falls in the middle of a page rather than a page boundary, the page must be read again to complete its join with the buffer pool and input R.

5.2 Beyond inner joins

In addition to inner joins, traditional join algorithms also serve semi-joins and anti-semi-joins (related to "in" and "not in" predicates with nested queries) as well as outer joins (preserving rows without matches from one or both inputs). In fact, some of the joins permit some optimizations. For example, a left semi-join can avoid the inner loop in nested loops join, avoid back-up logic in merge join, and short-circuit the loop traversing a hash bucket in hash join. On the other hand, some join algorithm require additional data structures. For example, a right semi-join implemented as nested loops join needs to keep track of which rows in the inner input have already had matches from earlier outer rows, and a hash join needs an additional bit with each record in its hash table.

In addition to join operations, relational query processing employs set operations such as intersection, union, and set difference. These operations may be specified in the query syntax or they may be inserted into a query execution plan during query optimization, in particular in plans exploiting multiple indexes for a single table. For example, conjunction queries might employ two indexes and intersect the resulting sets of row references.

G-join supports all of these operations. For some of them, it requires an additional bit for each record from the first input while a record is resident in the buffer pool. All other decisions for left and right semi-join, anti-semi-join, and outer join can readily be supported with small changes and optimizations in the join processing logic.

5.3 Complex queries

A join method is useful for database query processing only if it passes the "Guy Lohman test" [G 93]. It must be useful not only for joining two inputs but also in complex query execution plans that join multiple inputs on the same or on different columns.

In complex query execution plans with multiple join operations, g-join can operate as pipelined operation (in particular with pre-sorted inputs) or as "stop-and-go" operation or "pipeline breaker" for one or both inputs. The choice is dictated by input sizes or by external control, e.g., from the query optimizer. For example, a pipeline break can avoid resource contention with an earlier or a later operation in the query execution plan, it can enable a later operation to improve its resource management based on more accurate estimates of the join output size, or it can enable general dynamic query execution plans.

The output of g-join is almost sorted. If a perfect sort order is desired, the sort can be optimized to take advantage of the guaranteed key range. At any point in time, g-join can produce output only within a certain range of join key values defined by the current contents of the buffer pool. While the join output within that key range fits into the memory allocation of the sort operation, the sort operation can avoid temporary run files and immediately pipeline its output to the next operation in the query execution plan. Even if temporary run files are required, they can be merged eagerly up to a key value defined by the key range in the buffer pool of g-join.

If two instances of the new join form a producer-consumer relationship within a query execution plan and thus pipeline the intermediate result from one to the other, and if the join columns in the two join predicates share a prefix (or ideally are entirely the same), the output order produced by the first join improves the performance of the second. Even if the intermediate result is not perfectly sorted, its ordering has a high correlation with the required ordering in the second join operation. Thus, run generation in the second join operation achieves longer intermediate runs, fewer runs, and thus less intermediate merge effort or a smaller buffer pool during final join processing.

For this effect, it is not required that the join columns in the two join predicates be equal. It is sufficient that they share a prefix. If so, longer runs and thus more efficient join processing is entirely automatic. While this is also true for merge join with explicit sort operations, exploiting equal join predicates requires hash teams [GBC 98], which are more complex than traditional binary hash join algorithms but relatively simple compared to generalized hash teams [KKW 99] that exploit partial overlap of join predicates.

In relational data warehouses with star schemas around one or more fact tables, star joins combine multiple small dimension tables with a very large fact table. Optimizations for star joins include star indexes (b-tree indexes for the fact table with row identifiers of dimension tables as search keys), semi-join reduction (semi-joins between dimension tables

and secondary indexes of the fact table), and Cartesian products (of tiny dimension tables). It appears that g-join can support all required join operations and in fact exploits the size difference in joins of small (dimension) tables and large (fact) tables as well as hash join.

5.4 Parallel query execution

Parallel query execution relies on partitioning a single intermediate result, on pipelining intermediate results between operations in producer-consumer relationships, or on both. G-join can participate in all forms of parallel query execution. Partitioning intermediate results into subsets is entirely orthogonal to the choice of local algorithms. Pipelining intermediate results might be aided by exploiting not only equal column sets in join predicates of neighboring operations but also by exploiting join predicates that merely share a prefix. In other words, there is reason to expect that g-join enables efficient pipelining more readily than multiple merge join operations with intermediate sort operations. Compared to query execution plans with multiple hash join operations, g-join enables similar degrees of pipelining but it does so making much better use of the sort order of intermediate results.

6 Replacing traditional join algorithms

It is unrealistic to expect that g-join will displace all traditional join algorithms rapidly. Even if this goal succeeds eventually, it will take many years. As an analogy, it has taken decades for hash join to be implemented in all products. On the other hand, slow adoption permits additional innovation beyond the initial ideas. For example, after hybrid hash join was first published in 1984, Microsoft SQL Server included hash join only in 1998 [GBC 98], but it also included hash teams to mirror the advantages of interesting orderings in query execution plans based on merge join. Nonetheless, even if it is unrealistic to propose or to expect a rapid adoption of g-join, it seems worthwhile to make the case for replacing the traditional join algorithms.

6.1 Hash join

Hash join offers advantages over the other traditional join algorithms for unsorted, non-indexed join inputs. Thus, the focus of this discussion must be the case of unsorted, non-indexed input, e.g., intermediate results produced by earlier operations in the query.

Throughout Section 4, the cost of the new algorithm mirrors the cost of hash join including recursive partitioning and hybrid hash join. In addition to a fairly similar cost function for unsorted inputs, g-join also produces nearly sorted output without any extra effort.

The traditional optimizations of hash join readily transfer to g-join. For example, if compile-time query optimization errs in estimating relative input sizes, role reversal after run generation for both inputs is trivial. Similarly, due to separate phases consuming the two join inputs, bit vector filtering readily applies to g-join.

Hash join can readily integrate aggregation (grouping) on the join column, albeit only for its build input. If aggregation is desired for the inputs of g-join, sorting must merge the affected input until it forms a single run. Thus, some efficiency is lost, more so when aggregation applies to the larger input than for the smaller input, which g-join merges to *F/2* runs in any case. Note that Section 4.2 proposes merging these *F/2* runs to form an in-memory b-tree index. This merge step with a single output stream can readily perform aggregation or duplicate elimination for input R.

Hash teams are another optimization for query execution plans based on hash join and hash aggregation. They mirror the effects of interesting orderings in plans based on merge join and stream-aggregation. Earlier sections already discuss the ability of g-join to produce, consume, and exploit interesting orderings of intermediate results.

In summary, hash join and its optimizations shine for unsorted, non-indexed inputs. G-join closely matches the performance of hash join and its optimizations in all cases. While the performance of g-join does not exceed that of hash join, it produces and consumes sorted intermediate results and it eliminates the danger of a mistaken choice among multiple join algorithms.

6.2 Merge join

Merge join shines when both join inputs are sorted by prior operations, e.g., join or aggregation operations on the same columns or by scans of b-tree indexes. In those cases, g-join exploits the sorted inputs. Run generation is omitted and join processing consumes the join inputs, which take on the roles of runs in the discussion of Section 4. The buffer pool requires one or two pages for the smaller input. Note that a traditional merge join requires a small buffer pool to back up its inner scan in the case of duplicate join key values. In other words, if both inputs are sorted, g-join operates very much like a traditional merge join and its underlying movement of pages in the buffer pool.

If only one join input is sorted by prior operations, g-join consumes it as a single run and performs run generation for the other input, similar to run generation as discussed in Section 4 for two unsorted inputs. The performance of g-join in this case matches or exceeds that of merge join, because merging the unsorted input may stop early when many runs remain. The performance also matches or exceeds that of hash join, because no effort is required for partitioning or merging the input already sorted.

In summary, g-join matches or exceeds the performance of merge join in all cases. Note that qualitative information such as the sort order of indexes, scans, and intermediate results is known reliably at compile-time or at least at plan start-up-time; the decision whether or not sorting is required does not depend on error-prone quantitative information such as cardinality or cost estimation.

6.3 Index nested loops join

Index nested loops join shines in two distinct cases. First, when the outer input including an index fits in memory, the resulting algorithm is rather like an in-memory hash join. Second, if there is an index for the large inner input and there are fewer rows (or distinct join key values) in the outer input than pages in the inner input, then index nested loops join avoids reading useless pages in the large inner input. In both of these cases, g-join can match the performance of index nested loops join.

In the first case, run generation stops short of writing records from input R to temporary run files. Instead, all records remain in the run generation workspace, which takes on the role of the buffer pool. In-memory join processing may use an in-memory index structure like in-memory hash join or order-based merge logic like merge join.

In the second case, which is the traditional case for index nested loops join, g-join sorts the small input and then performs a zigzag merge join of the two inputs, i.e., the merge logic attempts to skip over useless input records rather than scan over them, and it applies this logic in both directions between the join inputs. If the number of distinct join key val-

ues in the smaller input is lower than the number of pages in the larger input, many of these pages are never needed for join processing. This is, of course, precisely the effect and the performance advantage of index nested loops join over merge join and hash join, and g-join mirrors this performance advantage precisely.

In order to achieve full performance, index access should to be optimized with proven techniques such as asynchronous prefetch, pinning page on the most recent root-to-leaf path in the buffer pool, leaf-to-root search using cached boundary keys, etc. These techniques limit the I/O cost of an index nested loops join to that of scanning the two indexes involved.

It is important to note that all required choices – whether to sort or to rely on the sort order of the input, whether to build an in-memory index or rely on a database index – are based on schema information, not on cardinality estimation. In other words, the detrimental effects of errors in compile-time cardinality estimation are vastly reduced.

7 Performance

A prototype of the core algorithm produced the results reported here. Michael Carey and his students are building a query execution system at UC-Irvine that includes g-join.

G-join combines well-studied elements of prior query processing algorithms. Implementation techniques and behavior of run generation, replacement selection, merging, in-memory hash tables, index creation, index search, etc. are all well understood. A new implementation of those algorithmic components is unlikely to yield new insights or results.

The principal novel component and the core of g-join is the schedule of page movements during join processing, i.e., the technique described in Section 4.2. The buffer pool loads and drops pages of runs from input R while individual pages of runs from input S are scheduled, read, and joined with the buffer pool contents. This is the algorithm component modeled in detail in the prototype. Actual I/O operations with on-disk files and operations on individual records are not included in the prototype.

The crucial performance characteristic that is not immediately known from prior work is the required size of the buffer pool. In the best case, only a single page of each run from input R is required; in the worst case, the buffer pool must grow to hold all of input R. The expectation from the discussion above is that about 2 pages per run from input R are required in the steady state. If this expectation is accurate, the I/O volume of g-join is practically equal to that of recursive and hybrid hash join. This assumes, of course, an unsorted input for g-join (as g-join would exploit pre-sorted inputs) and a perfectly uniform distribution of hash values in hash join (which is required to achieve balanced partitioning and to match the standard cost function in practice).

7.1 Implementation status and baseline experiment

The prototype focuses on page movements in the algorithm of Section 4.2. Input parameters include the run count for each input (defaults 10 and 90 runs), the page counts for each input (default 40 pages per run), and the number of values in the domain of join key values (default 1,000,000 distinct values). With random key ranges in input pages, the run sizes are only approximate. The output includes the average and maximum buffer pool sizes, and may include a trace showing how the buffer pool grows and shrinks over time.

With the default values, the prototype simulates a join of input R with about 400 pages to input S with about 3,600 pages. Figure 2 illustrates the size of the required buffer pool

for input R over the course of an experiment. The x-axis indicates how many pages of the large input S have already been joined. The y-axis shows the size of the buffer pool at that time, indicated as the average number of pages per run from input R. It can be clearly seen that this size hovers around 2 pages per run from input R. The buffer pool repeatedly grows beyond that, but not by very much. The maximum in this experiment is 2.3 pages per run (equal to 23 pages total in this experiment). The buffer pool also shrinks below 2 pages per run repeatedly and in fact more often and more pronounced than growing beyond 2 pages per run. At the end of the join algorithm, the buffer pool size shrinks to 1 page per run.

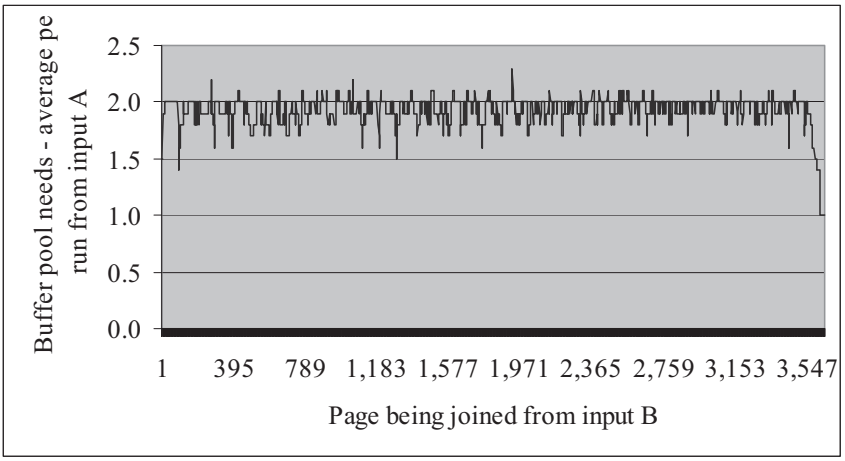

Figure 2. Buffer pool requirements over time.

7.2 Run counts and sizes

The next experiment shows how 2 pages per run from input R is quite stable across a range of memory sizes and input sizes.

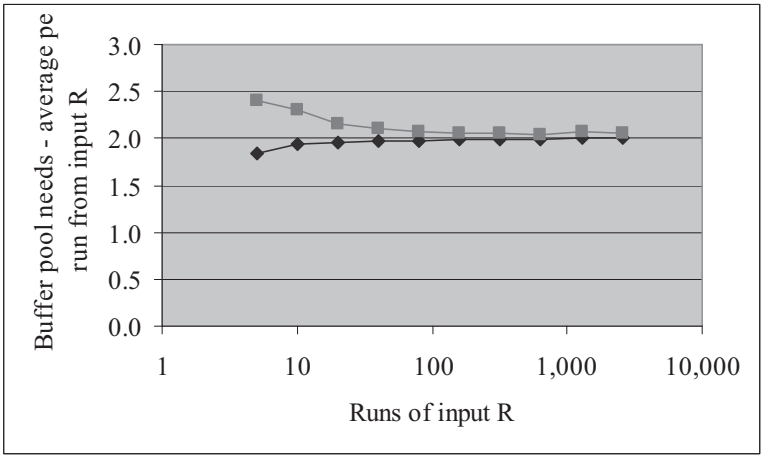

Figure 3. Buffer pool requirements with varying memory and input sizes.

Specifically, memory sizes in this experiment range from 10 pages to 5,120 pages, varied by powers of 2. Run sizes are assumed twice the memory size. The number of runs from input R is half the memory size (such that the buffer pool holds 2 pages per run). The number of runs from input S is 9 times larger, as in the prior experiment. Thus, run sizes vary from 20 to 10,240 pages and run counts vary from 5 to 2,560 for input R and from 45 to 23,040 for input S. Thus, input sizes vary from 100 to 26 million pages for input R and from 900 pages to 236 million pages for input S.

Figure 3 relates the number of runs from input R and the buffer pool requirements, both the average (lower curve) and the maximal (upper curve) buffer pool size for each memory and input size. In all cases, each run from the smaller input R requires about 2 pages in the buffer pool, confirming the basic hypothesis that g-join perform similar to hash join for large, unsorted inputs.

With an increasing number of runs from each input, the average grows closer to 2 and the maximum shrinks closer to 2. The former is due to many runs from the large input S; some page in some run from input S spans any page boundary in the runs from input R, and thus all runs from input R require about 2 pages in the buffer pool at all times. The latter is due to many runs from the small input R; even while some run might need 3 instead of 2 pages for a short period, it has little effect on the number of buffer pool pages when divided by the number of runs from input R. Thus, while the number of buffer pool pages is usually below 2, it sometimes is above 2, but only by a little bit and only for a short time.

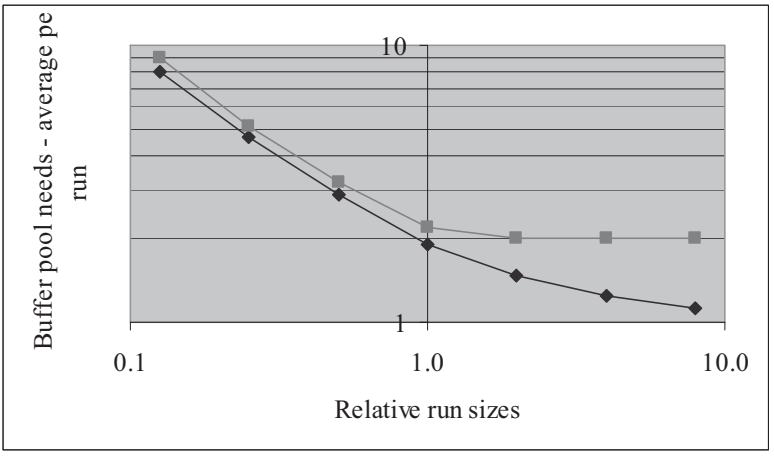

Figure 4. Buffer pool requirements with different run sizes.

Figure 4 illustrates the effect of insufficient or excessive merging in the large input S. In all cases, all runs from input R are of the same size and all runs from input S are of the same size. The x-axis indicates the quotient of runs size from input S to those from input R. The left-most data points indicate runs from input R 8 times larger than those from input S; the right-most data points indicate runs from input S 8 times larger than those from input R. The y-axis, ranging from 1 to 10 on a logarithmic scale, again shows average and maximal buffer pool needs, with the total buffer pool size divided by the number of runs. For all data points, there are 10 runs from input R and 90 runs from input S.

In the left half of the diagram, it is readily apparent that g-join needs many buffer pool pages per run if runs from input S are smaller. This is due to the large key range covered by

each page in such a run: it takes many pages of a larger run from input R to cover such a key range. In the right half of the diagram, where runs from input S are larger than the runs from input R, the average buffer pool requirements shrink almost to 1 page per run from input R. The maximal buffer pool requirements, however, do not.

Figure 4 permits two conclusions. First, in order to minimize buffer pool requirements, runs from input S require merging until all remaining runs are larger than all runs from input R. In this mode of operation, the cost function of g-join for unsorted inputs most closely resembles that of hash join. Second, if buffer space is readily available for runs from input R, it can be exploited to save some effort merging runs from input S. For example, with 10 buffer pool pages for each run from input R, runs from input S may be left smaller than those from input R, thus saving merge effort for input S.

7.3 Skew

Just like hash join suffers from skew in the distribution of hash values, g-join may suffer from various forms of skew in its inputs. There are several forms of skew, e.g., the sizes of runs (due to dynamic memory allocation during run generation) as well as skew in key value distribution. The form of skew most likely to affect the performance of g-join is skew in the sizes of runs. Such skew might be due to dynamic memory management during run generation or a correlation between input order and desired sort order.

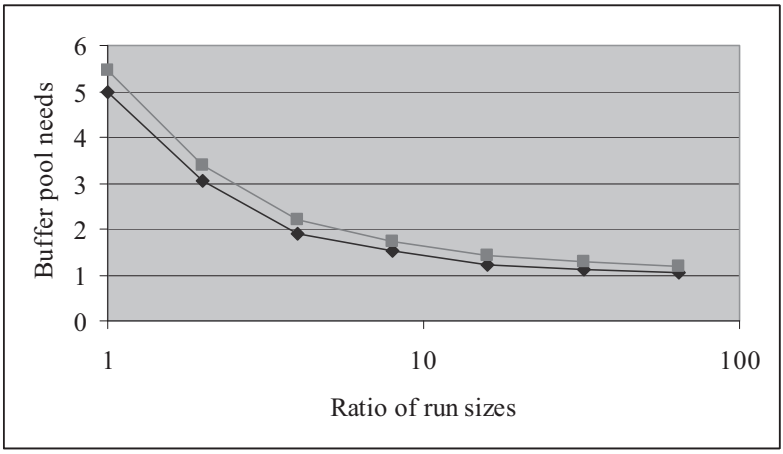

Figure 5. Buffer pool requirements with varying run sizes.

Figure 5 illustrates the effect when runs from the same input differ in size. (In Figure 4, all runs from either one input are the same size.) In Figure 5, sizes for runs from input R are chosen from the range 400 to 3,200 pages, i.e., the largest and smallest run might differ by a factor 8. Sizes for runs from input S might also differ by a factor 8, but the range is chosen differently for each data point. For the left-most data point (ratio = 1), the range is also 400 to 3,200 pages; for the right-most data point, the range is 64 times larger or 25,600 to 204,800 pages.

The buffer pool needs are governed by the largest run from input R and the smallest run from input S. They are equal for the central data point (ratio = 8), and the average and maximal buffer pool requirement for each run from input R is about 2 pages. It is actually less because some runs from input R are small and some runs from input S are large.

At the left-most data point, some runs from input R are much larger than some runs from input S. Those runs require many more pages in the buffer pool, and in fact dominate the overall buffer pool requirements. The number of pages per run from input R (about 5) is almost equal to the ratio of runs sizes (about 8).

At the right-most data point, all runs from input R are much smaller than all runs from input S. Thus, each page from input R covers many pages from input S. With fairly small key ranges within pages from input S, only a few runs from input R require 2 pages in the buffer pool at any point in time. Thus, the maximum buffer pool size (divided by the number runs from input R) is approaching the ideal value of 1.

7.4 Hyrax experiences

Michael Carey and students at UC-Irvine have experimented with g-join [L 10] within their Hyrax research prototype. Their implementation differs from the original design described above by using quicksort for run generation rather than replacement selection. Thus, runs are equal in size to the memory allocation, not twice. More importantly, this algorithm choice exacerbates the problem of a last run much smaller than memory and with a key range per page much larger than in other runs. They also observe that incidental ordering in an input has little effect on run sizes and run counts, which of course is different with replacement selection.

Their experiments so far show faster random writes during hash partitioning than random reads during merging and join processing in g-join. This is most likely due to automatic write-behind in hash join (using additional system memory) and the lack of forecasting and asynchronous read-ahead in this implementation of g-join. Nonetheless, their experiments confirm the above observations about the number of I/O operations and the amount of data written to and read from temporary files.

Finally, their experiments show average and maximal buffer pool sizes larger than shown in the experiments above, but still consistently below 3 pages if runs of input S are no smaller than runs of input R. It has been impossible to reproduce these larger buffer pool sizes with the initial implementation of the core algorithm used in the experiments reported above.

8 Summary and conclusions

In summary, the new, generalized join algorithm ("g-join") combines elements of the three traditional join algorithms yet it is an entirely new algorithm. This is most obvious in the case of two unsorted inputs, where g-join performs run generation like an external merge sort but then joins these runs without merging them (or with very little merging even for very large inputs). Therefore, g-join performs like merge join in the case of two sorted inputs and like hash join in the case of two unsorted inputs, including taking advantage of different input sizes. Our partial prototype implementation and our experimental evaluation confirm the analytical performance expectations.

In the case of indexed inputs, g-join exploits the indexes for sorted scans or even for searches in a zigzag merge join. Skipping over many pages in the index and fetching only those input pages truly required for the join is the main advantage of index nested loops join over hash join and merge join. G-join mirrors this advantage by using a zigzag merge join (skipping forward) rather than a traditional merge join (scanning forward). Thus, g-join performs as well as index nested loops join for a large, indexed, inner join input.

In conclusion, we believe that g-join competes with each of the three traditional join algorithms where they perform best. It could therefore be a replacement for each or for all of them. Replacing all three traditional join algorithms with g-join eliminates the danger of mistaken (join) algorithm choices during compile-time query optimization. Thus, g-join improves the robustness of query processing performance without reducing query execution performance.

Acknowledgements

Mike Carey, Guangqiang "Aries" Li, and Vinayak Borkar have implemented a variant of g-join and compared its performance with their implementation of hash join. Barb Peters, Harumi Kuno, and the reviewers suggested numerous improvements in the presentation of the material. Stephan Ewen (TU Berlin), Stefan Krompass, and Wey Guy provided excellent feedback on the algorithm, including test cases for robustness and general stress tests.

References

[AH 00] Ron Avnur, Joseph M. Hellerstein: Eddies: continuously adaptive query processing. SIGMOD 2000: 261-272.

[BBD 05] Pedro Bizarro, Shivnath Babu, David J. DeWitt, Jennifer Widom: Content-based routing: different plans for different data. VLDB 2005: 757-768.

[CK 97] Michael J. Carey, Donald Kossmann: On saying "enough already!" in SQL. SIGMOD 1997: 219-230.

[FKT 86] Shinya Fushimi, Masaru Kitsuregawa, Hidehiko Tanaka: An overview of the system software of a parallel relational database machine GRACE. VLDB 1986: 209-219.

[G 93] Goetz Graefe: Query evaluation techniques for large databases. ACM Comput. Surv. 25(2): 73-170 (1993).

[GBC 98] Goetz Graefe, Ross Bunker, Shaun Cooper: Hash joins and hash teams in Microsoft SQL Server. VLDB 1998: 86-97.

[HWM 98] Sven Helmer, Till Westmann, Guido Moerkotte: Diag-join: an opportunistic join algorithm for 1:N relationships. VLDB 1998: 98-109.

[IK 90] Yannis E. Ioannidis, Younkyung Cha Kang: Randomized algorithms for optimizing large join queries. SIGMOD 1990: 312-321.

[K 73] Donald E. Knuth: The art of computer programming, Volume III: sorting and searching. Addison-Wesley 1973.

[KKW 99] Alfons Kemper, Donald Kossmann, Christian Wiesner: Generalised hash teams for join and group-by. VLDB 1999: 30-41.

[L 10] Guangqiang Li: On the design and evaluation of a new order-based join algorithm. MS-CS thesis, UC Irvine, (2010).

[LSM 07] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, Guy M. Lohman: Adaptively reordering joins during query execution. ICDE 2007: 26-35.

[MHW 90] C. Mohan, Donald J. Haderle, Yun Wang, Josephine M. Cheng: Single table access using multiple indexes: optimization, execution, and concurrency control techniques. EDBT 1990: 29.

[MLR 03] Volker Markl, Guy M. Lohman, Vijayshankar Raman: LEO: An autonomic query optimizer for DB2. IBM Systems Journal 42(1): 98-106 (2003).

[NKT 88] Masaya Nakayama, Masaru Kitsuregawa, Mikio Takagi: Hash-partitioned join method using dynamic destaging strategy. VLDB 1988: 468-478.

[SAC 79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access path selection in a relational database management system. SIGMOD 1979: 23-34.