Comput Sci Res Dev
DOI 10.1007/s00450-011-0186-9

SPECIAL ISSUE PAPER

# New algorithms for join and grouping operations

**Goetz Graefe**

© Springer-Verlag 2011

**Abstract** Traditional database query processing relies on three types of algorithms for join and for grouping operations. For joins, 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. For grouping, an index-based algorithm has been used in the past whereas today sort- and hash-based algorithms prevail. 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 and among grouping algorithms by replacing the three traditional types of 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 find matching input records, the new join algorithm can also replace index nested loops join.

In addition to join operations, a very similar algorithm supports grouping ("group by" queries in SQL) and duplicate elimination. For unsorted inputs, candidate output records take on the role of one of the inputs in a join operation. Our goal is to define a single grouping algorithm that can replace grouping by repeated index searches, by sorting, and by hashing. In other words, our goal is to end mistaken algorithm choices not only for joins and other binary matching operations but also for grouping and other unary matching operations in database query processing.

Finally, these new algorithms can be instrumental for efficient and robust data processing in a map-reduce environment, because 'map' and 'reduce' operations are similar in essentials to join and grouping operations.

Results from an implementation of the core algorithm are reported.

**Keywords** Relational databases · SQL · Query processing · Robust performance · Query optimization choice · Error · Mistake · Query execution algorithms · Inner join · Semi-join · Outer join · Intersection · Union · Difference · "Group by" · Grouping · Duplicate removal · Uniqueness · Equivalence classes · Sort order · "Interesting orderings" · Order-preserving · Run generation · Merging · Buffer pool · Map-reduce

This paper largely overlaps with a paper presented at the BTW 2011 conference. Newly added sections cover prior work in much more detail and, more importantly, a similar generalized algorithm for grouping and aggregation.

G. Graefe (✉)
Hewlett-Packard Laboratories, Madison, WI, USA
e-mail: goetz.graefe@hp.com

## 1 Introduction

SQL is the only truly successful non-procedural programming language. In consequence, relational database management systems support physical data independence. For example, the set of indexes and their partitioning may

Published online: 30 June 2011

change without effect on the tables, views, integrity constraints, etc. 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 execution times worse than optimal by orders of magnitude which in turn leads to dissatisfied users and disrupted workflows in the data center. Investigation and resolution of intermittent 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 multi-dimensional histograms [59] or run-time feedback and statistical learning [55], our efforts center on query execution techniques. Adaptive merging [33] is one result of those efforts—automatic creation and optimization of B-tree indexes only if, when, and where warranted by the workload. Earlier results include B-tree techniques that permit efficient bulk load into fully indexed tables as well as pausing and resuming long-running index creation [24]. A recent research result is a set modifications of external merge sort that permit growing and shrinking the available memory allocation at any time and by nearly any amount with nearly no loss in efficiency and no loss of work already performed [30]. Contrary to prior efforts, e.g., [61], this includes exploiting memory growth even during the output phase.

The new algorithm for join and grouping operations is another result of this research. Its design objective is a single algorithm that can serve as a replacement for all three traditional join algorithms. In order to be viable, it must match 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 hash join, including 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 particularly small, the new join algorithm must perform as well as index nested loops join exploiting a temporary or permanent index for the large input. The performance requirements for grouping algorithms are analogous.

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 in order to match its performance goals. Therefore, we call it "generalized join algorithm" or, abbreviated, "g-join." The algorithm for grouping and duplicate removal is called "g-distinct."

**Table 1** Join algorithms and exploited input properties

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

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. Replacement selection (using a priority queue) and runs twice the size of memory are required to match the I/O volume of efficient hash join algorithms. 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 (Sect. 2) and then introduce g-join (Sect. 3). Algorithm details for unsorted inputs of various input sizes and unknown input sizes (Sect. 4) are followed by answers for the "usual questions" about any new query execution algorithm or new algorithmic variant (Sect. 5). Based on those details and answers, replacement of the traditional join algorithms is discussed in depth (Sect. 6). Grouping algorithms can be derived from the join algorithm by letting the candidate output records play the role of records from the small join input (Sect. 7). Open issues are then listed to the best of our ability (Sect. 8). Two partial prototype implementations permit an initial performance evaluation of g-join (Sect. 9). The last section offers our summary and conclusions from this effort so far as well as some ideas for future work.

## 2 Prior work

The present section assumes 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 in the discussion. Similarly, we ignore join operations without equality predicates. After the discussion of the three families of traditional join algorithms, the corresponding grouping algorithms are briefly reviewed.

G-join competes with the well-known (index) nested loops join, merge join, and (hybrid) hash join algorithms, which are reviewed in detail elsewhere [22]. The present section describes the traditional join algorithms as well as many of their optimizations, because g-join ought to compete not with their worst but with their best variants.

The diag-join algorithm [41] 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 [57]. Sorting the search keys prior to B-tree index search seems to ensure fairly robust query performance [37]. 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 [63], algebraic transformations [19, 32, 54], queries with very many joins [44], and dynamic join reordering [2, 6, 53]. Most of those research directions and their results are orthogonal to g-join and its relationship to the traditional join algorithms.

Other research efforts have exploited join algorithms for specific purposes, e.g., set operations such as an index intersection or a covering index join [36], or in specific settings, e.g., massively parallel query processing [3, 16]. Index operations (such as index intersection for a conjunction of predicates on a single table) use record identifiers as join keys. Parallel operations often partition both join inputs on a hash value calculated from the join keys, effectively adding an equality predicate on the hash values to the join predicate specified in the user query. G-join also works for these purposes and in those settings.

Hash values can also speed up local join operations. For example, a merge join and two sort operations on international strings can avoid many expensive comparisons if a hash value is computed for each string prior to the sort operations and strings are compared only in the case of equal hash values. The hash function employed can be order-preserving but that is not truly required.

Finally, hash values can speed up traditional indexes. For example, a B-tree index with a hash value as leading key field permits very efficient search based on fast comparisons as well as interpolation search rather than binary search [27]. Just as importantly for software vendors and users, it preserves all traditional B-tree virtues such as efficient creation using a sort, efficient maintenance, robust concurrency control and recovery, etc.

In the following discussion, the two join inputs are called R and S. Cost calculations may use the functions $pages(R)$ and $rows(R)$. $R$ used in a cost function means $pages(R)$. The function $keys(R)$ indicates the number of distinct values in the join column. 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 can usually be ignored.

In addition to calling the two inputs of a single join operation the small and the large inputs, the following text employs "tiny" for 10s of rows fitting into the CPU cache, "small" for 1,000s of rows fitting in memory, "medium-size" for 1,000,000s of rows requiring only a single merge step in an external merge sort, "large" for 1,000,000,000s of rows requiring multiple merge steps, and "huge" for 1,000,000,000,000s of rows requiring multiple merge levels, multiple partitioning levels in a recursive hash join, and multiple levels of nodes in a B-tree index.

## 2.1 Nested loops join

Nested loops join is both the simplest and the most versatile join algorithm. In its prototypical form, it loops over the rows of the outer input and, for each of those, loops over the rows of the inner input in order to find matches. The algorithm is competitive only for very small inputs or if some of the many optimizations are exploited.

A simple join of two tables may scan the inner input or search it using an index. A complex join of many tables may employ multiple levels of nested iteration, i.e., a nested subquery within a nested query [25]. For example, a standard rewriting for universal quantification employs two levels of nesting. As a specific example, a student who has attended all available database courses is one for whom there does not exist an available database course for which there does not exist an appropriate enrollment record [31]. As a join algorithm exploiting indexes, nested loops join may also be used for index-to-index navigation, e.g., when fetching entire rows in a primary index after obtaining appropriate references through a search in a secondary index.

Index nested loops join is the superior join algorithm if the outer input is tiny and the inner input is large and indexed. The index may be permanent or temporary, e.g., created and dropped within the current query execution. Modern storage devices with very short access latency, e.g., flash storage, will likely tilt query processing towards index usage both in selections (index search instead of table scans) and in joins (index nested loops join instead of merge join and hash join).

### 2.1.1 *Cost functions*

The I/O cost of nested loops join is dominated by searches in the inner input. In the case of a naïve nested loops join, the scans can be exceedingly expensive. In the case of index nested loops join, the index searches require root-to-leaf searches. If the buffer pool is "warm," only leaf pages incur actual I/O costs. Thus, the I/O cost of index nested loops join can be approximated by the number of searches, i.e., the number of records in the outer join input, multiplied by the cost of a random read operation.

The CPU cost is also dominated by the search in the inner input. In the case of an index nested loops join relying on a B-tree index and binary search within each B-tree node, the number of comparisons can be approximated by $\log_2(N)$ for an inner join input with $N$ records. The number of buffer pool operations in each search is equal to the depth of the B-tree index.

### 2.1.2 *Variants and optimizations*

Early optimizations of naïve nested loops join included block nested loops join including reverse scans [47] and balanced memory allocation [39]. Nonetheless, compared to multiple scans of the inner input, a temporary index might be more advantageous [67].

There are many further software optimizations to avoid or speed up I/O in nested loops join with large outer inputs. Prefetching the B-tree root page and 1 or 2 additional B-tree layers (and perhaps pinning them in the buffer pool) reduces each index search to 1–2 I/O operations at most. A batch of multiple look-ups permits using multiple disks and ordering disks accesses [57]. A continuous process based on a priority queue [45] avoids the burst effect. Sorting the entire outer input limits the cost of nested loops join even for a large outer input [17, 38]. In this case, the sequence of page accesses during the index nested loops join is very similar to that of a merge join [21]. Moreover, sorting the outer input partially or completely may detect duplicate key values in the outer input; in the best case, the number of searches in the index of the inner input is reduced from *rows*(*outer*) to *keys*(*outer*).

In order to reduce the CPU effort of index nested loops join, a cache may retain, for each B-tree level, the highest key value present in the buffer pool. Seeking from one B-tree leaf entry to another one may start with the current leaf page, thus avoiding almost all root-to-leaf searches. Other techniques for the same purpose include interpolation search [27] and pointer swizzling [58] among index pages in the buffer pool.

Alternatively, records of the outer input may contain direct pointers to records of the inner input [65], in particular when the database represents complex objects with enforced component relationships or foreign key referential constraints. Better yet, components of the same object may be stored together using master-detail clustering, e.g., invoices with their line items or even customers with all their orders, invoices, and all their line items. An implementation within traditional B-tree indexes [28] can keep the implementation effort moderate and preserve the many operational advantages of B-trees, e.g., efficient index creation [56] and robust concurrency control and recovery [29].

In summary, index nested loops join is the algorithm of choice in many queries. When appropriate software optimizations and modern hardware technologies are employed, this set of queries goes far beyond the tiny queries common in transaction processing. Nonetheless, our goal for g-join is to replace nested loops join, perhaps with the exception of nested queries not amenable to "flattening" or other appropriate query rewrite.

## 2.2 Merge join and sorting

Given the many variations and optimizations for nested loops join, maybe merge join is simpler after all. Merge join relies on both inputs being sorted on their respective join keys and merges them to find matching join key values. The inputs may be sorted due to appropriate index scans (e.g., "interesting orderings" due to B-tree indexes), prior operations that produce sorted output ("interesting orderings" due to in-stream aggregation or merge joins), or sort operations explicitly included in the query execution plan.

External merge sort is the standard sorting algorithm used in database systems and in query processing [26]. If sorting is required, its cost is much higher than the cost of the actual merge join, and most relevant optimizations apply to sorting rather than merge join.

### 2.2.1 *Cost functions*

The I/O cost of merge join is dominated by the cost of sorting the inputs, if required. The only specific issue is the final fan-in, i.e., how much memory is dedicated to each of the two final merge steps during the join phase. Primitive sort implementations that produce a single sorted file before the join can be modeled as a final fan-in of 1. If both join inputs require sorting, their final fan-in ought to be proportional to the input sizes, ensuring equal merge depth for both inputs as much as possible.

The CPU cost is also dominated by the sort operations. The number of comparisons in the merge join can be approximated by the number of records in both inputs, because one or the other input is advanced by one record in response to each comparison.

### 2.2.2 *Variants and optimizations*

Unsorted inputs and perfectly sorted inputs are the extreme cases. Intermediate cases exist and are exploited in some database systems. For example, a prior operation may deliver the intermediate query result sorted on fields (a, b) but the next operation needs it sorted on (a, b, c). In these cases, the intermediate result can be sorted in segments defined by unique values of (a, b). If the records from an input sorted on (a, b) are needed sorted on (b) only, the distinct values of (a) can be interpreted as identifying run sorted on (b) such that merely a merge step is required. If a sort order on (b, c) is needed, the techniques can be combined: distinct values of (a) identify runs sorted on (b); as these runs are merged, records are sorted on (c) for each unique value of (b). More complex variations and combinations of techniques are also possible.

If temporary runs are held on traditional disks with high transfer bandwidth but also high access latency, merging after initial run generation can be optimized by employing large units of I/O [35, 62]. With memory contention by concurrent virtual machines, applications, connections, queries, threads, and query operations, large inputs may require multiple merge steps. With optimized merge plans, the number of merge levels might not be uniform for all records [26]. A logarithmic function without rounding is not a precise but a reasonable approximation of the merge depth.

If a merge join requires an explicit sort operation for both inputs, their final (or only) merge steps can pipeline their results directly into the join logic. In this case, the sort operations compete with each other (and the join) for memory. The optimal division of memory assigns memory in proportion to the input sizes [22]. For example, if one input is much larger than the other one, the small input might be merged to a single sorted run and all available memory is assigned to the large input during the merge step, thus avoiding a complete merge level for the large input. Thus, in this limited way, merge join is able to take advantage of differences in input sizes, although not as much as hash join.

Just as index nested loops join with a sorted outer is similar to merge join, merge join can be similar to index nested loops join. Instead of scanning forward after comparing unequal keys of the two join inputs, merge join can also search for a matching key. In particular for two inputs of very different sizes, skipping forward can save many comparisons. For example, if the two sorted join inputs are organized in pages, the merge join can skip over an entire page with a single comparison. If the issue is not the difference in input sizes but two very different key value distributions, this optimization may apply in both directions, giving rise to the name zigzag merge join [1]. If one or both inputs of the merge join have ordered indexes such as B-trees, skipping forward can be faster than scanning forward by orders of magnitude.

Many sorting techniques and optimizations have been explored in the context of database query processing [26]. Important in the present context is run generation by replacement selection [50], i.e., a continuous process consuming unsorted input and producing initial runs, keeping the workspace in memory full at all times, organizing records in memory in a priority queue, and producing initial runs on average twice the size of memory, i.e., $2M$. With first and last runs of smaller size, $R/(2M) + 1$ initial runs should be expected for an input in random order and of size $R$. The sizes of first and last runs together are also about $2M$.

Priority queues are used in multiple roles in an efficient external merge sort: in addition to run generation and merging records, a merge step may forecast the run most urgently needing asynchronous read-ahead (with one entry in the priority queue for each run being merged), and merge planning may choose which runs to assign to the next merge step (with one entry in the priority queue for each run waiting to be merged).

## 2.3 Hash join

Three separate research efforts invented and developed hash join algorithms in the 1980s [7, 14, 15, 20, 48, 60, 64]. Since then, hash join has been considered the best join algorithm to exploit large memory and to process unsorted inputs. This is due to the fact that a huge input can be joined without writing and reading on temporary storage if the other join input (organized in a hash table) fits in memory. Usually, the inputs of hash join are called "build" and "probe" inputs rather than "outer" and "inner" inputs. Query optimization attempts to assign the "build" role to the small input.

If the small input nearly fits in memory, hybrid hash join avoids much or most I/O for the large input. An in-memory hash table is built with records of the small input and probed with records of the large input. If both inputs are large, the two inputs are partitioned into pairs of overflow files. The fan-out is limited by the number of output buffers quite similar to the limit on the merge fan-in in external merge sort. For huge inputs, multiple levels of partitioning may be required. Only the smaller input determines the number of partitioning levels, independent of the size of the larger input. Thus, hash join exploits the difference in the sizes of its inputs.

### 2.3.1 *Cost functions*

The I/O cost of hash join is usually dominated by the cost to obtain the inputs, often by table scans. The I/O within the hash join depends on the volume of overflow during partitioning. Hybrid hash join, for example, spills only parts of its inputs to temporary overflow files. Recursive hash join, if required, spills each record multiple times. Using large units of I/O strikes a balance between I/O access times and transfer times, quite similar to external merge sort.

The CPU cost of hash join also depends heavily on the overflow volume, plus building and probing an in-memory hash table with each input record.

### 2.3.2 *Variants and optimizations*

If it is not known a priori which of the join inputs is smaller, role reversal after the first partitioning step ensures minimal overall partitioning effort. In addition to role reversal, another optimization often associated with hash join is bit vector filtering. Key values in the build input records determine which bits are set to ‘1’ and probe input records that hash to ‘0’ bits are immediately discarded, potentially saving most of the I/O cost for the large input [14].

Other optimizations include integration of aggregation (grouping) and join—construction of the hash table from the build input can identify groups and aggregate them as appropriate, i.e., compute counts and sums. This applies only to the build input, not the probe input. Role reversal interferes with this variant of hash join.

Many queries use the same columns for grouping and join, because both usually focus on foreign key columns. For example, a many-to-one relationship such as invoices and their line items suggests queries that combine invoice attributes with summaries of line items. Moreover, aggregation operations often are “pushed down” [11] during query optimization, resulting in a query execution plan with a join following an aggregation on the same columns—the ideal situation for a hash join with integrated aggregation.

Another case elegantly addressed by a hash join with integrated aggregation is matching with long “in” lists, e.g., “from T where T.a in (@v1, @v2, ..., @v1024)”. The values with “@” indicate parameters supplied by the application at run-time. Some database applications generate such queries with enormously long lists. The traditional execution algorithm stores these values in an array and scans over the entire array for each row in table T. A more efficient implementation puts them into a hash table after removing duplicate values from the list. Duplicate removal is a form of aggregation, and the look-up in the hash table is a hash join.

Finally, teams of hash aggregation and hash join operations are quite similar in effect to multiple merge join operations with interesting orderings [36, 46]. Just as interesting orderings avoid the cost of sorting intermediate results, hash teams avoid partitioning intermediate results.

## 2.4 Comments on prior join algorithms

In addition to the detailed description of the traditional join algorithms and many of their optimizations, two comments seem pertinent to a new join algorithm.

First, the traditional wisdom seems to be that the larger the database and its tables, the larger the need for an efficient set-based join algorithm such as hash join. The following observation about database growth and scalability contradicts that wisdom, however. If the large input table doubles in size, query execution time doubles for merge join and hash join, whereas it barely grows for index nested loops join. Thus, if index creation and maintenance can be optimized, index nested loops join is the join algorithm of choice in large databases. Flash storage amplifies this argument due to its excellent access latency. Therefore, a new join algorithm must mirror the scalability of all three traditional join algorithms.

Second, the traditional mode of operation of index nested loops join assumes a small outer input and a large, indexed, inner input. If, on the other hand, the small input and its index fits in memory, index nested loops join can employ the large input as outer input. Some optimizations of the index seem appropriate, e.g., pointer swizzling and interpolation search as mentioned above. Interestingly, the resulting join algorithm is somewhat similar to an in-memory hash join. G-join exploits a similar mode of operation, as described later.

## 2.5 Sort-based aggregation

Early work on implementation techniques for query processing in relational databases assumed that aggregation requires an input sorted on the grouping columns. Once the input sorted, the cost of the actual aggregation is minimal or even trivial. Thus, sorting the input was usually the dominant cost of aggregation operations.

Over time, researchers and developers realized that many optimizations apply, e.g., early duplicate removal during sorting [5], exploiting functional dependencies among database columns [66] and reducing the sort effort if an input is already sorted on some but not all required ordering fields.

## 2.6 Nested loops aggregation

In order to avoid the cost of sorting, some relational products employ an index holding partially aggregated records. For example, for the SQL query “select dept-id, average (salary) from employee group by dept-id,” the index contains department identifiers as key plus sum and count of salaries encountered so far in the input. Early Sybase systems, for example, use a B-tree index for aggregation operations.

Ideally, the buffer pool can retain this temporary index throughout the operation, such that no I/O is required for the aggregation output. When the output is larger than the available memory allocation, index nested loops aggregation relies on the buffer pool and its replacement policy.

### 2.7 Hash aggregation

Instead of a B-tree index in shared temporary database space and hopefully in the buffer pool, hash aggregation employs an in-memory, thread-private hash table. When the output size exceeds the memory allocation, hash aggregation relies on partitioning just like hash join. Again, hash partitioning is akin to a distribution sort on hash values instead of standard column values.

For output sizes only slightly larger than the memory allocation, hybrid hashing can be employed. The in-memory hash table retains either all records belonging to specific hash buckets or the records with the highest consumption rate, i.e., the records that represent the largest groups within the input and that thus absorb the most input records. This latter variant is not a straight distribution sort but akin to a "top" operation with an in-memory priority queue. A possible implementation approximates the consumption rate by tracking, for each record in the hash table, the most recent (or 2$^{\text{nd}}$- or 3$^{\text{rd}}$-most recent) aggregation. Whenever a replacement victim is required, the record is chosen that has not absorbed an input record for the longest time.

### 2.8 Summary of prior work

In summary, database query processing relies on matching algorithms for many unary and binary operations. The unary operations include grouping (for "group by" queries) and duplicate removal. The binary operations include inner and outer joins, semi joins, and set operations such as intersection and difference. For unary operations, there are three traditional algorithms based on sorting, hashing (partitioning), and temporary indexes. Similarly, there are sort-based, hash-based, and index-based join algorithms. One of the tasks of compile-time query optimization is to choose the correct algorithm for each query and each operation based on estimated sizes of intermediate query results and anticipated key value distributions.

## 3 The new join algorithm

G-join is a new algorithm; it is not a run-time switch between 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 a query execution plan. For unsorted inputs, it employs run generation very similarly to a traditional external merge sort. Thereafter, it avoids most or all of the merge steps in a traditional merge sort. The required effort, in particular the I/O effort, is similar to that of hash join, because 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. For example, it does not rely partitioning using an order-preserving hash function. Delaying all details to subsequent sections, the basic ideas are as follows.

For unsorted inputs, run generation produces runs like an external merge sort, but merging these runs can be omitted in most cases. With a fixed memory allocation, all initial run sizes are about equal. 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 two segments, one used for immediate join processing and one used as buffers for temporary files. In this way, 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 and the join output is also strictly sorted.

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 and the algorithm behaves rather like hash join and like 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 key value distributions (neither duplicate skew nor distribution skew), 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 Sect. 4.2.

In order to explain g-join step-by-step, Sects. 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, run generation, and merging.

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 Sect. 4.2.1.

### 4.1 Case R $\leq$ 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 or 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 $R = F \times M$

The next case is the one in which Grace hash join [20] 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 $F/2+1$. The number of runs from input S is larger by a factor $S/R$. 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. Additional buffer frames might be used as output buffer and for asynchronous read-ahead and write-behind, but those are ignored in the cost calculations here.

#### 4.2.1 *Page operations*

Careful scheduling governs read operations in runs from inputs R and S. This scheduling is the core technique that enables the new, generalized join and aggregation algorithms, discussed here for the case of join algorithms for two unsorted inputs.

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

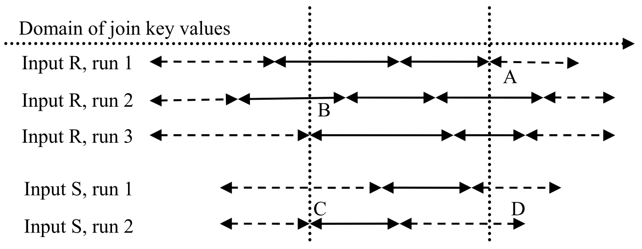

**Fig. 1** Runs, pages, buffer pool, and the immediate join key range

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. Thus, even if there are many more pages from input S than from input R, the memory requirements do not change. The letters A through D refer to the priority queues explained below.

In Fig. 1, the buffer pool contains 2 pages each from runs 1 and 3 of input R and 3 pages from run 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.

*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. Runs are represented in these priority queues until processed to completion. 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. In Fig. 1, the top entry of this priority queue would be run 1 of input R, the source of the next page to be loaded into the buffer pool (see label "A" in Fig. 1).
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. In Fig. 1, the top entry of this priority queue would be run 2 of input R, because the buffer pool will shrink next by dropping a page of that run (see label "B" in Fig. 1).
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. In Fig. 1, the top of this priority queue would be run 2 of input S, because joining the next page of that run will permit shrinking the buffer pool (see label "C" in Fig. 1).
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 *high key value in the next page* on disk. In Fig. 1, the top entry of this priority queue would be run 2 of input S, because its next page can be joined with the least growth of the buffer pool (see label "D" in Fig. 1).

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.

An alternative to priority queue B merges individual records of input R, which is similar to a traditional merge sort of input R producing a single sorted stream. While records may remain in pages in the buffer pool, the sorted stream of record pointers is captured as a in-memory B-tree index, similar to efficient creation of B-tree indexes from sorted streams. The key range represented in the in-memory B-tree index is the immediate join key range, which is bounded by the top entries in priority queues A and B. The buffer pool may drop a page of input R when the index no longer contains any references to the page.

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. The algorithm without priority queue D does not require a larger maximal buffer pool, although it may require a large buffer pool over longer periods. Priority queue D provides guidance required for growing the buffer pool as late or as slowly as possible, but it cannot avoid buffer pool growth or reduce the maximal buffer pool size. Priority queue D requires information about the maximal key value in pages not yet read; this is possible only if external merge sort saves its runs in an appropriate format, e.g., B-trees [24].

*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.

The algorithm variant that merges individual records into an in-memory index may drop low key values in the index up to the top key value in priority queue C, as priority queue C indicates the lowest key value of input S that might still require join processing.

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 [49] 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.

*First and last runs* During run generation, the first and last runs from each input may differ from the runs produced by steady-state run generation. The most significant issue is a small last run in the large input. For example, run generation with read-sort-write cycles and in-memory algorithms such as quicksort can produce a last run much smaller than memory and thus all previous runs. A small run and thus few data pages imply a large key range per page. During join processing, a page from the large input with a large key range imposes a large buffer pool requirement for the small input.

Continuous run generation with replacement selection alleviates the issue but does not solve it entirely. Even when replacement selection is employed, the first and last runs have, on average, larger key ranges per page than the runs produced by steady-state replacement selection. The first run is simply smaller than twice the size of memory; the last run covers only an initial part of the key range and is limited to the memory size.

A straightforward approach merges the first and last runs prior to join processing. Another solution dedicates an additional buffer page to the first and last runs of the large input. In that solution, the large join input requires a total of three memory pages rather than one as discussed so far. One of the additional memory pages remains dedicated to the first run. The other additional memory page serves the last run until it ends and then serves the second-to-last run. In the key range covered by the last run, the second-to-last run is similar to runs produced during steady-state run generation; its key range per page grows only after the end of the unsorted input had been reached. Join processing for the contents of those dedicated pages proceeds in multiple steps or key ranges, as appropriate for the contents of the buffer pool from the small input and its incremental progress through the domain of join keys.

Alternatively, a single memory page for the large join input remains possible if pages with particularly large key ranges are read repeatedly, i.e., each time another range of join key values can be joined with the buffer pool contents. More generally, a small buffer pool of fixed size, e.g., 3 pages, could be managed for the large input in order to minimize the buffer pool requirements for the small input. Even more generally, a single buffer pool for runs from both the small and large inputs could be managed for minimal buffer pool and I/O requirements. The detailed design, its computational cost and its advantages are not yet known. The buffer pool and I/O requirements should be less than those of any rigid scheme, e.g., one or three pages for the large join input.

*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 runtime 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 in-memory 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 joins with only a few records of input R. Thus, a lot of skipping and searching is required in a global B-tree index, whether the actual in-memory join algorithm is a merge join, an index nested loops join, or a zigzag merge join.

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. Even a hash table permits bulk deletion or recycling of records based on the low boundary of immediate join key range.

## 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 output 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 Sect. 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 Sect. 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 Sect. 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 $K = \mathit{ceiling}((R - M) \div (M - 1))$.

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 Sect. 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 [8, 9], 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 overflow 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 Sect. 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 Sect. 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 $K = \mathit{ceiling}((R - M) \div (M - 1))$ of Sect. 4.3 is replaced with $K = \mathit{ceiling}((R/F - M) \div (M - 1))$ 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 Sect. 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, dynamic changes in the size of the in-memory hash table and its hash buckets are 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 [20] without dynamic de-staging [60].

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 [49]. 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 dynamic memory allocation, memory hierarchies, integrity constraints, partial and incidental sort orders in the inputs, skew in the distribution of join key values, binary operations of the relational algebra other than inner join, unary operations such as duplicate elimination, algorithm variants for early results, 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 to 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.

If the buffer pool is about to grow beyond the expected size, i.e., two pages per run of input R, it seems worthwhile to "compact" memory contents after joining a page of input S. This requires keeping or tossing individual records from input R rather than entire pages. Thus, the algorithm actually merges the runs from input R rather than managing entire pages in the buffer pool. It requires at least one additional copy step for each record of input R. On average, this technique will free $1/2$ page for each of the $F/2$ runs, or about $1/4$ of memory.

Finally, it seems worthwhile to optimize the selection of key values at page boundaries. The same heuristics for short separator keys that optimize suffix truncation in B-tree indexes [4] may also align page boundaries in runs within each input and between the two inputs. If that optimization applies, fewer pages from input S require multiple pages from input R in the buffer pool.

Two special cases are worth calling out and including in a test suite for g-join. First, if one or both inputs are sorted on the join columns but query optimization and plan generation did not take advantage of it, run generation by replacement selection will produce only a single run for each input and join processing equals a traditional merge join thereafter. Second, if one or both inputs are sorted in reverse order, replacement selection offers no advantages over read-sort-write cycles using quicksort, i.e., run sizes equal the memory allocation during run generation. Additional merging may be required to reduce the number of runs from the smaller input to $F/2$. Otherwise, this case should be almost as efficient as join processing for random inputs.

In addition to skew detection and resolution, some techniques might be useful designed to avoid skew and its negative effects. For example, in all cases in which hash join and hash aggregation apply, sort-based algorithms such as merge join and g-join can sort, group, and join on hash values rather than key values. This technique should reduce or eliminate problems due to skew just as it does in hash join and hash aggregation. It remains possible to exploit interesting orderings for intermediate query results. For data in the database, B-tree indexes on hash values can avoid explicit sort operations. Moreover, these hash values may reduce the CPU effort compared to key values because they function as poor man's normalized key [26] in the sort logic and in the join logic.

## 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 algorithms 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" [22]. 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 [12, 23, 34].

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 [36], which are more complex than traditional binary hash join algorithms but relatively simple compared to generalized hash teams [46] that exploit partial overlap of join predicates.

This benefit occurs whether the first instance of g-join is the first or the second input (R or S) of the second instance. Thus, g-join might be particularly beneficial in bushy query execution plans.

In relational data warehouses with star schemas around one or more fact tables, star joins combine multiple small dimension tables with a 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.

Parallel query execution can benefit from semi-join reduction or its approximation by bit vector filtering. If the matching operation in semi-join reduction is implemented by g-join, the main join algorithm benefits not only from data reduction but also from the interesting ordering. In other words, the semi-join (in the data sources) can speed up the final join operation (in the data destinations). Even if the semi-join does not produce fully sorted results, any amount of pre-sorting in the intermediate data streams improves the run length and thus the run count in the final join. Exploiting partially sorted inputs is inherent in run generation by replacement selection and applies both to traditional sort operations and to g-join.

## 5.5 Dynamic memory allocations

For large inputs, g-join runs for some period of time, just like sorting, merge join, and hash join. During that time, resource contention might fluctuate. Can g-join adjust its memory allocation after it has started consuming input?

Mechanisms for dynamic memory allocation during run generation have been explored elsewhere. For example, one sort algorithm primarily designed for variable-size records and for graceful degradation employs a priority queue for run generation, ejecting records from the in-memory workspace to runs on disk only as needed for additional input records, thus naturally adjusting to a growing or shrinking workspace [51].

Mechanisms for dynamic memory allocation during intermediate merge steps have been proposed in the past, including some that adjust only between merge steps and some that can adjust within a merge step. For example, the unit of I/O (page size) might be adjustable or the merge fan-in may be modified [69].

Dynamic memory allocation during the final join processing phase is as difficult as during a final merge step in an external merge sort. Once both inputs R and S have been merged such that input R is in $F/2$ runs and input S has been merged into runs of similar sizes, shrinking the memory allocation requires to interrupt join processing and to perform some additional intermediate merge steps. It is yet more difficult to grow the memory allocation during join processing and improve performance as appropriate for the new memory size. A new technique found to be effective for external merge sort is expected to be transferable to the new join algorithm [30].

In summary, for long-running queries, g-join promises to be as dynamic as external merge sort. A small loss of efficiency is incurred for the flexibility of growing and shrinking the memory allocation. The loss in efficiency, however, is much smaller than the loss incurred by not exploiting all available memory when it becomes available.

## 6 Replacing traditional 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 [36], 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 Sect. 3, 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 input sizes and in particular 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.

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 Sect. 3. 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 Sect. 3 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 values 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 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.

### 6.4 Summary

In summary, g-join is competitive with hash join, merge join, and index nested loops join in all situations. 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 Grouping, aggregation, and duplicate elimination

Join operations—including outer joins, semi joins, and set operations such as intersection—are just one type of expensive operations in database query processing; the other important type is grouping including "group by" aggregation and duplicate removal. The most obvious difference between these two types of operations is that joins have two inputs whereas grouping operations have only one input. Their commonality is that records "match" by equal attribute values and that matching records must be brought together by searching, sorting, or partitioning. Database systems usually employ merge sort for sort-based operations such as merge join and index creation.

The g-join algorithm can easily be transformed into an equally new algorithm for grouping. The resulting generalized algorithm may be called g-distinct, g-grouping, or g-aggregation. It combines algorithmic aspects as well as performance characteristics of the three traditional grouping algorithms that are based on a temporary index, sorted data, or hash partitioning. For sorted input data, g-distinct exploits the sort order and is indistinguishable from traditional in-stream aggregation for sorted inputs. For unsorted input data, g-distinct employs run generation like g-join but avoids sorting the input completely. This latter aspect is much like g-join but also like hash aggregation, which avoids much partitioning work by aggregating input records as early as possible. For in-memory processing, g-distinct employs a temporary in-memory hash index or other index.

It might be instructive to compare g-distinct to traditional grouping algorithms based on merge sort and hash partitioning. Like a traditional merge sort, g-distinct relies on generation and merging of sorted runs. Differently from a traditional sort operation, which employs its memory as input buffers for runs being merged, g-distinct uses only a single input buffer and employs all its memory for candidate output records. Like a traditional hash aggregation algorithm, g-distinct employs an in-memory index where it creates and maintains candidate output records. Differently from hash aggregation, the contents of that in-memory hash table incrementally turn over as the algorithm proceeds through the domain of grouping values. Moreover, g-distinct exploits sorted input and can easily produce sorted output. Even partially or inversely sorted input may be exploited depending on the specific techniques implemented for run generation.

### 7.1 Aggregation and g-distinct

The essence of the new grouping algorithm, compared to the new join algorithm for unsorted inputs, is to let the aggregation output serve in the role of the small input of g-join. The principle is analogous to the relationship between hash join and hash aggregation. Hash join builds the in-memory hash table with one input and probes the hash table with the other one, whereas hash aggregation collects and holds output records (or intermediate records) in the hash table and probes and augments the hash table with its one input.

These roles can perhaps be explained most concretely using the record formats involved in a non-trivial aggregation calculation, e.g., when computing the average salary per department. All record formats contain a department identifier. The input records contain employee name etc. including most importantly (here) an individual salary value. Intermediate records contain a sum of salaries as well as a count of employees represented by the sum. Final output records contain neither sum nor count but their quotient, i.e., the average salary. In g-distinct used as a generalized aggregation algorithm, records of the intermediate format occupy the buffer pool, just like the small input in a g-join operation. When an input record refers to a group not yet in the buffer pool, a new intermediate record is created. When an input record refers to a group already in the buffer pool, the appropriate aggregation takes place. When an intermediate record is ejected from the buffer pool, a final record is derived from the intermediate record. Again, this is analogous to hash aggregation.

In order to serve as an algorithm for grouping, the g-join algorithm requires some modifications. First, the "small input" is created incrementally in memory rather than read from multiple runs. Consequently, priority queues A and B are omitted. Second, priority queue C guides both reading from the input and eviction of records from the buffer pool. Until all input records within a key range have been consumed and the key range can be evicted, new values in the grouping attributes may be encountered and thus new output records may need to be created. As in g-join, the optional priority queue D may reduce the duration of large buffer pool requirements but it does not reduce the maximal buffer pool requirement. Third, space management in the buffer pool must allow out-of-order creation of candidate output records. Moreover, the buffer pool must also allow changes in record sizes in some cases, e.g., a “max” operation for variable-length string attributes. Fourth, the conditions change for the hybrid of internal and external operation, i.e., in-memory aggregation versus run generation and merging.

Hash aggregation switches from pure in-memory aggregation into overflow mode if the output size exceeds the memory allocation. In hybrid hash aggregation, the required number of overflow partitions uses essentially the same calculation as in hybrid hash join. For build input size $R$ and probe input size $S$ in hybrid hash join (with memory size $M$, partitioning fan-out $F$, typically $F = M$, and $M \leq R \leq F \times M$), the number of required overflow partitions is $K = (R - M) \div (M - 1)$. Similarly, hybrid hash aggregation with input size $I$ and output size $O$ (and $M \leq O \leq F \times M$) requires $K = (O - M) \div (M - 1)$ overflow partitions, with $K = 0$ for $O = M$ and $K = F$ for $O = F \times M$.

This calculation of $K$ applies to both hash aggregation and g-distinct. In the latter case, $K$ memory pages are employed for run generation, just as in g-join in hybrid mode. The remaining $M - K$ memory pages contain candidate output records with partial sum and counts. $M - K$ memory pages absorb $(M - K) \times I \div O$ input pages (assuming a sufficiently uniform key value distribution). The remaining input data form runs of *2K* pages on average (assuming run generation using replacement selection).

If $O \leq F \times M$, g-distinct in hybrid mode applies, even if the input size is many times larger and a traditional sort-based aggregation (or duplicate removal) operation would require multiple merge steps. For $O > F \times M$, even g-distinct should employ some intermediate merge steps such that the final aggregation step consumes large runs with many pages and thus pages with small key ranges. As in g-join’s large input, g-distinct input pages with large key ranges increase the required size of the buffer pool. Note that $O > F \times M$ also defines the case in which hash aggregation requires intermediate partitioning steps.

While g-join produces nearly sorted output, g-distinct produces completely sorted output. Moreover, it produces its first output very quickly after consuming its input, in particular in the hybrid operating mode. Its first output is earlier than in traditional hash aggregation because it produces its output page by page, not an entire hash table at a time; and its output is earlier than in traditional sort-based duplicate removal because it might not need intermediate merge steps. Moreover, one could force a hybrid initial step in g-distinct, thus producing some output from records in the buffer pool immediately after initial run generation. Such earliest aggregation output incurs additional expenses in the forms of smaller runs, more runs, and additional merging before producing further output.

## 7.2 Integrated operations

G-join and g-aggregation can be combined into a single operation, quite like the integration of hash join and hash aggregation [36]. This algorithm performs grouping on its two inputs and then joins the two grouping results. Consider, for example, a query to find customers whose average invoice shows a lower dollar amount than their average order, indicating an inability to satisfy entire orders at once. Grouping is required for two database tables (“orders” and “invoices”) followed by a join of the group summaries. All three operations focus on customer identifiers and can therefore be integrated into a single physical operation and algorithm.

Existing implementations (including [36]) integrate grouping only on the build input of the join operation. In other words, the example query above requires at least two operations, each with its own hash table organized by customer identifier. Moreover, due to its asymmetry, this design prohibits role reversal in integrated operations. Recently it has become clear, however, that this restriction is not required, even for query processing using hash-based algorithms. In the improved design, records in the hash table may contain preliminary aggregations (e.g., sums and counts but not averages) for both inputs. In the example above, each record in the hash table summarizes information about one customer, including information from orders as well as invoices. Producing an output record from a record in the hash table finalizes all aggregations (e.g., averages from sums and counts) and applies the join predicate (e.g., comparing average order amount and average invoice amount).

Three further comments on this new improvement of hash join, hash aggregation, and their integration: First, the equality predicate applies prior to aggregation in the probe input and thus eliminates some records from the probe input. Second, minor modifications permit semi-join and outer join operations. For example, the example query above might be a full outer join such that both customers with orders but no invoices and customers with invoices but no orders can be included in the output. Third, the hash join algorithm with aggregation supported for both inputs also permits role reversal, e.g., if the build input (or its aggregation) unexpectedly turns out to be larger than the probe input (or its aggregation).

Like hash join with these new improvements, g-join can support aggregation on both inputs. The design is similar to the one outlined above for hash join. Records in the buffer pool contain the appropriate aggregation fields. In the example query, this would include sum and count of orders and sum and count of invoices. Semi-joins and outer joins, e.g., customers with orders or invoices but not both, can be supported. Role reversal, bit vector filtering, etc. all remain possible in g-join with aggregation on one or both inputs.

Moreover, immediate aggregation and join processing may begin even if run generation leaves more than $F/2$

runs for each input. The output of the aggregation must be smaller than $F \times M$ for at least one of the inputs. This input then takes the role of the small input of g-join and runs from that input must be aggregated in the buffer pool.

This is actually quite similar to overflow during hash aggregation prior to a join. In a sense, in both hash join and g-join the sizes of aggregation output drive the in-memory data structures (and thus memory requirements and overflow) and both join inputs are aggregated into these data structures.

For g-join with aggregation of huge inputs, intermediate merge steps may be required. If so, these merge steps can apply early duplicate removal and aggregation as in traditional merge sort [5].

### 7.3 Integration with partitioned B-trees

Integration of g-distinct with g-join is one opportunity; integration of g-distinct with partitioned B-trees is another. Partitioning within a B-tree is based on an artificial leading key field and combined with online reorganization. Partitions are created and removed as easily as insertion and deletion of records; catalog operations are not required. Partitioned B-trees can be exploited during external merge sort for accurate deep read-ahead and for dynamic resource allocation, during index creation for a reduced delay until the first query can search the new index, and for miscellaneous other operations. If only a single value for this leading B-tree column is present, which is the usual and most desirable state, the B-tree index is rather like a traditional index. If multiple values are present at any one point in time, which usually is only a transient state, the set of index entries is effectively partitioned.

Additional use cases for partitioned B-trees include efficient bulk import (initial and incremental loading) and adaptive indexing. Among the two approaches to adaptive indexing, adaptive merging [33] is more suitable here than database cracking [43], which relies on an in-memory columnar database stored in arrays and on partitioning steps similar to those in quicksort [42]. Adaptive merging relies on partitioned B-trees [24], where each partition is quite similar to a run in external merge sort.

Adaptive merging creates and optimizes B-tree indexes as side effects of query execution. The first query extracts future index entries from the primary storage structure (e.g., a clustered index), sorts them into runs within the limits of a convenient memory allocation, and saves those runs in a partitioned B-tree. The run identifier in each record serves as artificial leading key field and thus as partition identifier. Subsequent queries merge key values satisfying their range predicates. Eventually, all key values in active key ranges can be found in a single partition. From this point forward, the index performs just like a traditional B-tree index, yet index creation and optimization have been side effects of query execution.

The initial phase of adaptive merging leaves sorted runs, represented as partitions within a partitioned B-tree index. In many cases, G-distinct can consume those partitions immediately, i.e., without merging them. Recall that g-distinct can process arbitrarily many runs in a single pass over the data as long as the aggregation output is smaller than memory times the merge fan-in, i.e., $O \leq M \times F$. In contrast, hash aggregation succeeds in a single pass over the data only for output sizes smaller than memory, i.e., $O \leq M$. With merge fan-in $F$ at least in the 10s and often in the 100s, the former condition is much more likely to hold than the latter. In other words, g-distinct takes full advantage of the pass over the data implicit in creation of the index in its initial format, i.e., in runs or partitions.

Partitioned B-trees aid not only incremental index creation, e.g., in adaptive merging, but also efficient index maintenance, e.g., bulk insertion or loading. For a table only one index, e.g., a primary B-tree index or an index-organized table, replacement selection can organize the stream of new records into runs twice the memory size. These runs can immediately form partitions within the partitioned B-tree. For a table with multiple indexes, e.g., multiple secondary B-trees in complete independent sort orders, quicksort and traditional read-sort-write cycles can organize index entries into partitions aligned across all indexes.

For example, a fact table in a data warehouse might include a foreign key for each dimension table and indexes on most or all of them. Loading additional fact rows is a frequent and traditionally very expensive operation, which partitioned B-trees can speed up considerably. Moreover, aggregation operations by foreign key columns are very common, because they derive summary information about dimension entities, e.g., the sum of sales by retail store.

For a table with one or multiple indexes, if a g-distinct operation finds data partially sorted, i.e., organized into runs or index partitions, it can take full advantage of this prior sort effort, whereas hash aggregation does not. Again, while hash aggregation can process a table in memory only if $O \leq M$, g-distinct can do so if $O \leq M \times F$, with merge fan-in $F$ in the 10s or the 100s. Similarly, hash aggregation completes with a single level of external partitioning and overflow files if $O \leq M \times F$, whereas g-distinct starting with input partially sorted in a partitioned B-tree requires only a single merge level if $O \leq M \times F^2$.

Adaptive merging and incremental loading using partitioned B-trees interact not only with g-distinct but also with g-join. For the small input, existing B-tree partitions must be merged in preparation of a g-join execution until no more than $F/2$ partitions remain. For the large input, the number of partitions is not important and can be arbitrarily large. It is important, however, that runs or partitions sizes in the large input match or exceed those of the small input, as discussed earlier (Sect. 4.6). More research into adaptive merging, the g-join family of algorithms, and their interaction is required to optimize robust query execution algorithms and adaptive physical database design.

## 8 Open issues

Many open issues have been resolved in the two partial prototypes and in many discussions with knowledgeable colleagues. Several hard issues remain, however, and are listed here to the best of our knowledge.

Given multiple "waterproof" methods to cope with skew as well as multiple heuristics for reducing the performance effects of skew, it will require practical experience with real systems and real data before a preferred skew management technique will emerge. Note that this is quite similar to hash join. Algorithms to cope with skew in hash value distributions, in particular due to duplicate key values, have been described only after hash join had been designed and implemented for products [36, 68].

The motivation for a new join algorithm had been the desire to avoid mistaken choices among the three traditional join algorithms. Another important issue addressed by compile-time query optimization is the join order. Several research and development efforts have addressed poorly chosen join orders by dynamic choice among alternative query execution plans or plan fragments [12, 34], by dynamic reordering of join operations [53], or by routing individual records or groups of records in intermediate results [2, 6]. G-join seems compatible with dynamic query execution plans and, inasmuch as it relies on and resembles nested iteration and index searches, with routing individual records. It is unclear whether it can serve as a stepping stone towards robustness in query execution performance including all of join order, join algorithms, and access paths.

Parallel query execution requires another choice in addition to access paths, join algorithms, and join order. Traditional parallel join algorithms either partition both inputs on the join attribute or partition the large input and broadcast the small join input. Ideally, one or both join inputs are partitioned as required in the database, i.e., no additional partitioning effort is required. Traditional query optimization chooses between symmetric partitioning and broadcasting based on cardinality estimation, with the same perils as in the choice of join algorithms. The solution of this problem might be found in an adaptive hybrid strategy, strategies based on $m \times n$ matrix of join locations, with the values of $m$ and $n$ set dynamically. The details and advantages of such an approach are open questions, as is the compatibility of g-join with such join strategies.

Mistaken choices of join order and join algorithm go beyond read-only queries, e.g., during join processing to validate foreign key constraints during definition and database updates, or during join processing during maintenance of materialized views. Moreover, they apply to data management software that supports streaming data [10], data-driven joins, and early results [18]. At this point, applying or adopting g-join to these operations remains future research.

Finally, nested queries are a special case of joins. In most cases, query rewrite can transform SQL syntax with nested sub-queries into equivalent algebra expressions with inner joins, (anti) semi-joins, and other operations that can be supported with g-join. It remains an open issue, however, whether all nested queries can be transformed in this way and processed with g-join. Nonetheless, it seems certain that any query expression that can be supported by merge join or hash join can also be supported by g-join.

## 9 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 Sect. 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).

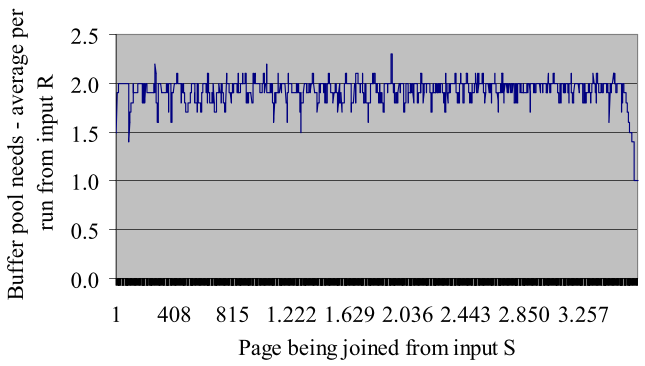

**Fig. 2** Buffer pool requirements over time

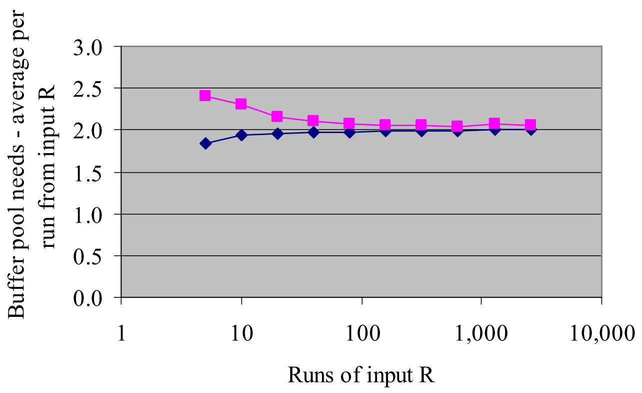

**Fig. 3** Buffer pool requirements with varying memory and input sizes

### 9.1 Implementation status and baseline experiment

The prototype focuses on page movements in the algorithm of Sect. 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. Managing the priority queues takes less than 0.1 seconds on a laptop with a 2.8 GHz CPU. Thus, the overhead of the priority queues seems negligible compared to sorting and joining records as well as the I/O during join processing.

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.

### 9.2 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.

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 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.

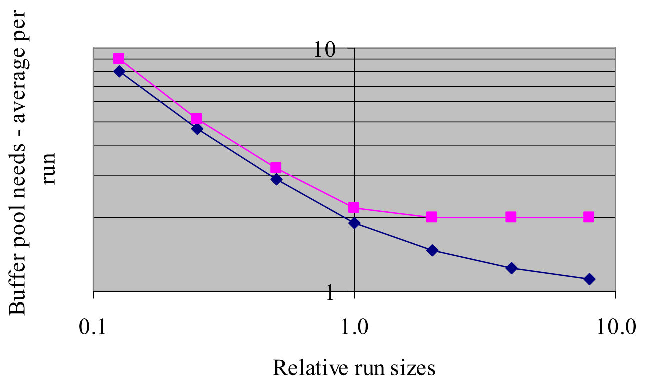

**Fig. 4** Buffer pool requirements with different run sizes

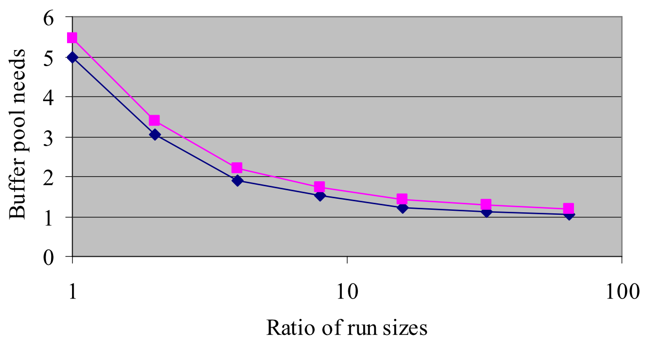

**Fig. 5** Buffer pool requirements with varying run sizes

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.

## 9.3 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 in run generation by replacement selection.

Figure 5 illustrates the effect when runs from the same input differ in size. (In Fig. 4, all runs from either one input are the same size.) In Fig. 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.

## 9.4 Hyrax experiences

Michael Carey and students at UC-Irvine have experimented with g-join [52] 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 per run 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.

## 10 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 huge 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, g-join performs as well as merge join and hash join for sorted and unsorted inputs, and as well as index nested loops join for large indexed inputs. Thus, 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.

Reducing the repertoire to a single algorithm for join and for grouping also simplifies integration of traditional relational operations for set operations into modern execution frameworks such as MapReduce and Hadoop [13]. G-join can process map-reduce operations—'reduce' operations are usually equivalent to aggregation operations and 'map' operations can be either joins with other data sets or function applications. Caching results of expensive functions is very similar to duplicate removal operations and the same basic algorithms apply [40]. A single implementation for each of these operations eliminates the burden of choosing an algorithm for many 'map' and 'reduce' operations. Thus, g-join and g-distinct are much better suited for modern execution frameworks than the troika of traditional join algorithms and the troika of traditional aggregation algorithms.

While g-join and g-distinct are based on merging sorted runs and thus akin to traditional merge sort, an alternative family of algorithms can be based on distribution sort. It might be surprising that a second such family of algorithms exists that may replace all other join algorithms and grouping algorithms. The well-known duality of merge sort and distribution sort, however, suggests a pair of algorithm families. For lack of imagination, we call the resulting algorithms h-join and h-grouping. The suggested implementation of distribution sort focuses on order-preserving hash functions in hybrid hash join, hybrid hash aggregation, histogram-guided recursive partitioning, and hash teams [36]. Role reversal, bit vector filtering, asynchronous read-ahead and write-behind, zigzag merge join (skipping forward instead of scanning forward, in particular when joining ordered indexes), etc. all remain relevant. Our future research will also explore this opportunity.

**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, Stefan Krompass, and Wey Guy provided excellent feedback on the algorithm, including test cases for robustness and general stress tests.

## References

1. Antoshenkov G, Ziauddin M (1996) Query processing and optimization in Oracle Rdb. VLDB J 5(4):229–237
2. Avnur R, Hellerstein JM (2000) Eddies: continuously adaptive query processing. In: SIGMOD, pp 261–272
3. Ballinger C, Fryer R (1997) Born to be parallel: why parallel origins give Teradata an enduring performance edge. IEEE Data Eng Bull 20(2):3–12
4. Bayer R, Unterauer K (1977) Prefix B-Trees. ACM Trans Database Syst 2(1):11–26
5. Bitton D, DeWitt DJ (1983) Duplicate record elimination in large data files. ACM Trans Database Syst 8(2):255–265
6. Bizarro P, Babu S, DeWitt DJ, Widom J (2005) Content-based routing: different plans for different data. In: VLDB, pp 757–768
7. Bratbergsengen K (1984) Hashing methods and relational algebra operations. In: VLDB, pp 323–333
8. Carey MJ, Kossmann D (1997) On saying "enough already!" in SQL. In: SIGMOD, pp 219–230
9. Carey MJ, Kossmann D (1997) Processing top n and bottom n queries. IEEE Data Eng Bull 20(3):12–19

10. Chandrasekaran S, Franklin MJ (2002) Streaming queries over streaming data. In: VLDB, pp 203–214
11. Chaudhuri S, Shim K (1994) Including group-by in query optimization. In: VLDB, pp 354–366
12. Cole RL, Graefe G (1994) Optimization of dynamic query evaluation plans. In: SIGMOD, pp 150–160
13. Dean J, Ghemawat S (2004) MapReduce—simplified data processing on large clusters. In: OSDI, pp 137–150
14. DeWitt DJ, Gerber RH (1985) Multiprocessor hash-based join algorithms. In: VLDB, pp 151–164
15. DeWitt DJ, Katz RH, Olken F, Shapiro LD, Stonebraker M, Wood DA (1984) Implementation techniques for main memory database systems. In: SIGMOD, pp 1–8
16. DeWitt DJ, Gerber RH, Graefe G, Heytens ML, Kumar KB, Muralikrishna M (1986) GAMMA—a high performance dataflow database machine. In: VLDB, pp 228–237
17. DeWitt DJ, Naughton JF, Burger J (1993) Nested loops revisited. In: PDIS, pp 230–242
18. Dittrich J-P, Seeger B, Taylor DS, Widmayer P (2002) Progressive merge join: a generic and non-blocking sort-based join algorithm. In: VLDB, pp 299–310
19. Freytag JC, Goodman N (1989) On the translation of relational queries into iterative programs. ACM Trans Database Syst 14(1):1–27
20. Fushimi S, Kitsuregawa M, Tanaka H (1986) An overview of the system software of a parallel relational database machine GRACE. In: VLDB, pp 209–219
21. Gassner P, Lohman GM, Schiefer KB, Wang Y (1993) Query optimization in the IBM DB2 family. IEEE Data Eng Bull 16(4):4–18
22. Graefe G (1993) Query evaluation techniques for large databases. ACM Comput Surv 25(2):73–170
23. Graefe G (2000) Dynamic query evaluation plans: some course corrections? IEEE Data Eng Bull 23(2):3–6
24. Graefe G (2003) Sorting and indexing with partitioned B-trees. In: CIDR
25. Graefe G (2003) Executing nested queries. In: BTW, pp 58–77
26. Graefe G (2006) Implementing sorting in database systems. ACM Comput. Surv. 38(3)
27. Graefe G (2006) B-tree indexes, interpolation search, and skew. In: DaMoN, p 5
28. Graefe G (2007) Master-detail clustering using merged indexes. Inform Forsch Entwickl 21(3–4):127–145
29. Graefe G (2010) A survey of B-tree locking techniques. ACM Trans Database Syst
30. Graefe G (2010) Robust sorting (in preparation)
31. Graefe G, Cole RL (1995) Fast algorithms for universal quantification in large databases. ACM Trans Database Syst 20(2):187–236
32. Graefe G, DeWitt DJ (1987) The Exodus optimizer generator. In: SIGMOD, pp 160–172
33. Graefe G, Kuno HA (2010) Self-selecting, self-tuning, incrementally optimized indexes. In: EDBT, pp 371–381
34. Graefe G, Ward K (1989) Dynamic query evaluation plans. In: SIGMOD, pp 358–366
35. Graefe G, Linville A, Shapiro LD (1994) Sort versus hash revisited. IEEE Trans Knowl Data Eng 6(6):934–944
36. Graefe G, Bunker R, Cooper S (1998) Hash joins and hash teams in Microsoft SQL Server. In: VLDB, pp 86–97
37. Graefe G, Kuno HA, Wiener JL (2009) Visualizing the robustness of query execution. In: CIDR
38. Haas LM, Carey MJ, Livny M, Shukla A (1997) Seeking the truth about ad-hoc join costs. VLDB J 6(3):241–256
39. Hagmann RB (1986) An observation on database buffering performance metrics. In: VLDB, pp 289–293
40. Hellerstein JM (1998) Optimization techniques for queries with expensive methods. ACM Trans Database Syst 23(2):113–157
41. Helmer S, Westmann T, Moerkotte G (1998) Diag-join: an opportunistic join algorithm for 1:N relationships. In: VLDB, pp 98–109
42. Hoare CAR (1962) Quicksort. Comput J 5(1):10–15
43. Idreos S, Kersten ML, Manegold S (2007) Database cracking. In: CIDR, pp 68–78
44. Ioannidis YE, Kang YC (1990) Randomized algorithms for optimizing large join queries. In: SIGMOD, pp 312–321
45. Keller T, Graefe G, Maier D (1991) Efficient assembly of complex objects. In: SIGMOD, pp 148–157
46. Kemper A, Kossmann D, Wiesner C (1999) Generalised hash teams for join and group-by. In: VLDB, pp 30–41
47. Kim W (1980) A new way to compute the product and join of relations. In: SIGMOD, pp 179–187
48. Kitsuregawa M, Nakayama M, Takagi M (1989) The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In: VLDB, pp 257–266
49. Knuth DE (1973) The art of computer programming, vol III: sorting and searching. Addison-Wesley, Reading
50. Larson P-Å (2003) External sorting: run formation revisited. IEEE Trans Knowl Data Eng 15(4):961–972
51. Larson P-Å, Graefe G (1998) Memory management during run generation in external sorting. In: SIGMOD, pp 472–483
52. Li G (2010) On the design and evaluation of a new order-based join algorithm. MS-CS thesis, UC Irvine
53. Li Q, Shao M, Markl V, Beyer KS, Colby LS, Lohman GM (2007) Adaptively reordering joins during query execution. In: ICDE, pp 26–35
54. Lohman GM (1988) Grammar-like functional rules for representing query optimization alternatives. In: SIGMOD, pp 18–27
55. Markl V, Lohman GM, Raman V (2003) LEO: An autonomic query optimizer for DB2. IBM Syst J 42(1):98–106
56. Mohan C, Narang I (1992) Algorithms for creating indexes for very large tables without quiescing updates. In: SIGMOD, pp 361–370
57. Mohan C, Haderle DJ, Wang Y, Cheng JM (1990) Single table access using multiple indexes: optimization, execution, and concurrency control techniques. In: EDBT, pp 29–43
58. Moss JEB (1992) Working with persistent objects: to swizzle or not to swizzle. IEEE Trans Softw Eng 18(8):657–673
59. Muralikrishna M, DeWitt DJ (1988) Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In: SIGMOD, pp 28–36
60. Nakayama M, Kitsuregawa M, Takagi M (1988) Hash-partitioned join method using dynamic destaging strategy. In: VLDB, pp 468–478
61. Pang H, Carey MJ, Livny M (1993) Memory-adaptive external sorting. In: VLDB, pp 618–629
62. Salzberg B (1989) Merging sorted runs using large main memory. Acta Inform 27(3):195–215
63. Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1979) Access path selection in a relational database management system. In: SIGMOD, pp 23–34
64. Shapiro LD (1986) Join processing in database systems with large main memories. ACM Trans Database Syst 11(3):239–264
65. Shekita EJ, Carey MJ (1990) A performance evaluation of pointer-based joins. In: SIGMOD, pp 300–311
66. Simmen DE, Shekita EJ, Malkemus T (1996) Fundamental techniques for order optimization. In: SIGMOD, pp 57–67
67. Youssefi K, Wong E (1979) Query processing in a relational database management system. In: VLDB, pp 409–417
68. Zeller H, Gray J (1990) An adaptive hash join algorithm for multiuser environments. In: VLDB, pp 186–197
69. Zhang W, Larson P-Å (1998) Buffering and read-ahead strategies for external mergesort. In: VLDB, pp 523–533

**Goetz Graefe** is an HP Fellow researching indexing and query processing. Prior to joining Hewlett-Packard Laboratories in 2006, Goetz spent 12 years as software architect in product development at Microsoft, mostly in database management. Both query optimization and query execution of Microsoft's 1990s reimplementation of SQL Server are based on his designs. Goetz's research credentials include numerous original publications as well as surveys published by ACM Computing Surveys and ACM Transactions on Database Systems. His original publications span query optimization, query execution, indexing, and concurrency control. His work has been honored by the ACM SIGMOD 2000 "test of time" award for work on parallel query execution, by the IEEE ICDE 2005 "influential paper" award for work on extensible query execution, and by the 2009 ACM "software systems" award for participation in the Gamma database machine research project.