# Implementation of Sorting in Database Systems

Goetz Graefe

## Abstract

It has often been said that sorting algorithms are very instructional in their own right and representative of a variety of computer algorithms, and that the performance of sorting is indicative of the performance of a variety of other data management tasks. Therefore, there is a fair amount of literature about the theory of sorting as well as about specific benchmark results. On the other hand, most commercial implementations of sorting do (or should!) exploit a number of techniques that are publicly known but not widely discussed in the research literature. This survey collects them for easy reference by students, researchers, and product developers. This paper does not contain novel algorithmic techniques and does not experimentally evaluate the effectiveness of any one individual technique; instead, it gathers and organizes such techniques in order to organize, stimulate, and focus future research.

## 1 Introduction

Every computer science student learns (or definitely should learn!) about *N log N* in-memory sorting algorithms as well as about external merge sort, and virtually all database products employ these algorithms for query processing and index creation. However, the implementations of sorting in commercial database systems differ greatly from one another, and the same is true among database research prototypes. These differences are due to "all the little tricks" that are either exploited or not. Many of these techniques are public knowledge but not widely known. The purpose of this paper is to survey them in order to make them readily available to students, researchers, and industrial software developers. Another purpose is to add structure to discussions of real-world sort implementations.

Commercial database systems employ sorting in many places. The obvious ones are user-requested sorted query output, index creation, and query operations such as grouping, duplicate removal, verifying uniqueness, rank and top, and merge join, including minor variations of merge join for outer join, semi join, intersection, union, and difference. In addition, sorting can be used for logical consistency checks (e.g., verifying a referential foreign key constraint that requires that each *line item* indeed have an associated *order*) and physical consistency checks (e.g., verifying rows in a table and entries in a non-clustered index precisely match up), because both consistency checks are essentially joins. Similarly, sorting may speed up fetch operations following a non-clustered index scan, because fetching records from a clustered index or heap file is tantamount to joining a stream of pointers to a disk. In an object-oriented database system, fetching multiple object components as well as looking up the physical locations for multiple logical object id's are forms of internal joins that may benefit from sorting. Finally, sorting can be used when compressing recovery logs or replication actions as well as during media recovery while applying the transaction log. Nonetheless, in spite of these many different uses, we focus here on query processing and on index maintenance since these two applications cover all issues found in the other ones, and we let the reader apply the techniques to the other use cases.

There are many interesting relationships between sorting and B-trees. Most obviously, building a large B-tree is much faster if the future B-tree entries are sorted first and the B-tree is built bottom-up, rather than using random top-down insertions. Similarly, sorting can be employed to optimize large inserts that touch many or all of the B-tree leaves. As a more interesting example, one can write run files in external merge sort in B-tree structure: it imposes a small overhead but includes in the run sufficient information for accurate read-ahead planning when merging runs from multiple striped disks. In general, it seems that a lot of techniques that improve one can be adapted for the other, and researchers interested in either B-trees or in sorting might benefit from studying known techniques for the other.

This survey is divided into three parts. First, in-memory sorting is considered. Presuming that quicksort and basic priority heaps are well known and understood by the reader, techniques to speed basic in-memory sorting are discussed, for example techniques related to CPU caches or to faster comparisons. Second, external sorting is considered. Again presuming that basic external merge sort is well known and understood, variations on this basic theme are discussed in detail, for example graceful degradation if the memory size is almost but not quite large enough for in-memory sorting – the sorting equivalent to hybrid hash join. Finally, sorting techniques that uniquely apply to sorting within a query execution plan are discussed, for example memory management in complex query plans with multiple pipelined sort operations or with nested iteration. Query optimization, as important as it is for database query performance, is not covered here, except for a few topics directly related to sorting.

In this paper, we make assumptions that represent typical database environments. Records consist of multiple fields, each with their own type and length. Some fields are of fixed length, others of variable length. The sort key includes some fields in the record, but not necessarily the leading fields, and possibly all fields but typically not. Memory is sizeable, but often not large enough to hold the entire input. CPUs are fast, to some extent through the use of caches, and there are more disk drives than CPUs. For brevity or clarity, we presume an ascending sort in some places, but adaptation to descending sort or multi-attribute mixed sort is very straightforward.

## 2 Internal sort: avoiding and speeding comparisons

Given that in-memory sorting is well understood in general, this section only surveys the less known or less used implementation techniques, their complexities and costs, and their benefits.

### *2.1 Comparison-based sorting vs. distribution sort*

Traditionally, database sort implementations have used comparison-based sort algorithms, e.g., internal merge sort or quicksort, rather than distribution sort or radix sort, which distribute data items to buckets based on numeric interpretation of bytes in sort keys. However, comparisons imply conditional branches, which imply potential stalls in the CPU's execution pipeline. While modern CPUs benefit greatly from built-in branch prediction hardware, the whole point of key comparisons in a sort is that their outcome is not predictable. Therefore, a sort that does not require comparisons seems very attractive.

Radix sort and other distribution sorts are often discussed, because they promise fewer pipeline stalls on modern processors. One key argument is that the nature of comparisons in a sort algorithm is that their outcome is not predictable; therefore, branch prediction hardware necessarily performs poorly for key comparisons. An interesting variant of distribution sort counts value occurrences in a first pass over the data and then allocates precisely the right amount of storage to be used in a second pass that redistributes the data [A 96]. However, these sort algorithms have not supplanted comparison-based sorting in database systems. Implementers have been hesitant because these sort algorithms suffer from several shortcomings. First and most importantly, if keys are long, many passes over the data are needed. For variable-length keys, the maximal length must be considered. If key normalization (explained shortly) is used, lengths might be variable and fairly long, even longer than the original record. If key normalization is not used, managing field types, lengths, sort orders, etc. makes distribution sort fairly complex, and typically not worth the effort. A promising approach, however, is to use one partitioning step (or a small number) before using a comparison-based sort algorithm on each resulting bucket [AAC 97]. Second, radix sort is most effective if data values are very uniformly distributed. This cannot be presumed in general, but may be achievable if compression is used, because compression attempts to give maximal entropy to each bit and each byte, which implies uniform distribution. Of course, to achieve the correct sort order, the compression must be order preserving. Effective compression requires that the data distribution be known in advance, which typically would require an additional pass over intermediate query results, which might cost more than radix sort saves.

## 2.2 Normalized keys

The cost of in-memory sorting is dominated by two operations: key comparisons (or other inspection of the keys, e.g., in radix sort) and data movement. Creating and moving representatives for data records, e.g., pointers, typically address the latter issue - more details on this later. The former issue can be very complex, due to multiple columns within each key, each with their own type, length, collating sequence (e.g., case-insensitive German), sort direction (ascending or descending), etc. Given that each record participates in many comparisons (*2 log N* comparisons of 2 records for a *N log N* sort), it seems worthwhile to reformat each record before and after sorting if the alternative format speeds up multiple operations in between. A typical alternative format is a simple binary string, in such a way that the transformation is both order-preserving and loss-less. In other words, the entire complexity of key comparisons can be reduced to comparing binary strings, and the final records can be recovered from the binary string.

Some implementations apply such reformatting only to the key, because it participates in the most operations and the most costly operations. This is why this technique is usually called *key normalization* or *key conditioning*. Even computing only the first few bytes of the normalized key is beneficial if most comparisons will be decided by the first few bytes alone. However, copying is also expensive, and treating an entire record as a single field reduces overheads for space management and allocation as well as address computations. Thus, normalization can be applied to the entire record. The disadvantage of reformatting the entire record is that the resulting binary string might be substantially larger than the original record, in particular for loss-less normalization and for some international collation sequences, thus increasing the requirements for both memory and disk, both space and bandwidth. There are some remedies, however. If it is known a priori that some fields will never participate in comparisons, e.g., because earlier fields in the sort key form a unique key for the record collection being sorted, the normalization for those fields does not need to preserve order; it just needs to enable fast copying of records and recovery of original values. Moreover, a binary string is much easier to compress than a complex records with individual fields of different types – more on order-preserving compression shortly. In the remainder of this survey, we presume that normalized keys and records are used, and do not detail how to apply the described techniques to traditional multi-field records.

## 2.3 Order-preserving compression

Data compression can be used to save space and bandwidth in all levels of the memory hierarchy. There are many compression schemes, and many of them can be adapted to be order preserving, typically with a small loss in compression effectiveness and possibly with some additional complexity in the comparison method. For example, optimal Huffman code is created by successively merging two sets of symbols, starting with each symbol forming a singleton set and

[2] It is also amazing how very simple indexing heuristics result in a well-indexed database. One heuristic simply builds indexes on primary keys, foreign keys, dates, and all columns in equality and “in” predicates; if clustered indexes are an option, the primary key is also the clustering key. A variation includes all primary and foreign keys in all non-clustered indexes in order to enable index-only navigation through multiple tables. An alternative heuristic builds the same indexes on primary keys and possibly on foreign keys, and additional indexes only if, for an actual query, the new index retrieves 10 times fewer records than any existing index. Not surprisingly, these two heuristics will build almost the same indexes for most databases and workloads.

ending with a single set containing all symbols. The two sets to be merged are the ones with the lowest combined rate of occurrence; by restricting this rule to sets that are immediate neighbors in the desired sort order, an order-preserving compression scheme is obtained. Order-preserving Huffman compression, by imposing a restriction on the creation process, is somewhat less effective than non-order-preserving Huffman compression, but it is still quite effective for most data.

When dictionary compression is modified to be order preserving, it might require that the symbol following the substituted string be considered [ALM 96]. The dictionary includes all individual symbols plus strings of symbols that are found to occur frequently. In ordinary dictionary compression, inserting a string into the dictionary does not affect the encoding of other strings. In order-preserving dictionary compression, the longest pre-existing prefix of a newly inserted string must be assigned two rather than one encoding. For example, when the string “th” is inserted into the dictionary with its own code, the longest prefix that already exists in the dictionary, in this case “t”, must be assigned not one but two codes in the dictionary, one for “t” strings followed by a letter less than “h” and one for “t” strings followed by a symbol greater than “h”. The encoding for “th” might be the value 24, and the encoding for “t” is 23 if the “t” is followed by one of “a” through “g” or it is 25 if followed by “i” through “z”. Using these three codes, the strings “ta”, “th”, and “tz” can be compressed and sorted correctly based on their encodings.

Compression has been used in database systems to preserve disk and memory space, and more importantly to better exploit available disk and memory bandwidth. However, it has generally not been used in database sorting, although it seems worthwhile to explore this combination, in particular if it is integrated with key normalization. Note that only the key (possibly including the input sequence number – see above) needs to be compressed using an order-preserving method. One of the issues is that the same compression scheme has to be used for all records, and the distribution of values is typically not known when the sort operations starts.

## 2.4 Cache-optimized techniques

Given today’s hardware as well as foreseeable trends, CPU caches must be considered and exploited in high-performance software. Beyond the obvious, e.g., aligning data structures and records to cache line boundaries, there are two principal sources of ideas. First, one can attempt to leave the full records in main memory (i.e., not access them) and use only record surrogates in the cache. Second, one can attempt to adapt and re-apply to CPU caches any and all techniques used in external sort to ameliorate the distance between memory and disk.

Typically, in-memory sorting uses an array of pointers, simply to avoid moving variable-length records. Heavily used, these pointers typically end up in the cache. Similarly, the first few bytes of each key are fairly heavily used, and it seems advantageous to design data structures and algorithms such that those, too, are likely to be in the cache. One such design includes a fixed-size prefix of each key with each pointer, such that the array of pointers becomes an array of structures with pointer and key prefix. Alternatively, a three-level scheme could be used, but probably wouldn’t be beneficial. We call fixed-size prefixes of normalized keys used in this way *poor man’s normalized keys*. They can be very effective if keys typically differ in the first few bytes. If, however, the first few bytes are typically equal and the comparisons of poor man’s normalized keys will all have the same result, these comparisons are virtually free, as the comparison is compiled into the sort code and the branch prediction logic of modern CPUs will ensure that such useless predictable comparisons will not stall the CPU pipeline.

An alternative design for order-preserving fixed-size fixed-type keys starts with a fixed key value and represents each actual key value using the difference from that one reference key. The fixed-size key is obtained by combining the count of leading equal symbols and the first differing symbol [BL 89]. Depending on whether the actual key is smaller or larger than the reference key, the count and the difference are expressed in positive or negative numbers, such that the result is indeed order-preserving. This design works particularly well if many actual keys share a prefix with the reference key, and probably works best with partitioning-based sort algorithms such as quicksort, in particular if a new reference key is used for each partitioning step [BL 89]. Moreover, if multiple reference keys are used for disjoint key ranges and the fixed-size fixed-type key encodes the range in its highest-order bits, such reference key encoding might also speed up comparisons while merging runs.

Among the techniques that can be adapted from external sorting to cache-optimized in-memory sorting, the most obvious one is to create runs the size of the cache and then to merge multiple such runs in memory before producing the final output (in-memory sort) or writing the result as a base level run to disk (external sort). Poor man's normalized keys and cache-size runs are two principal techniques exploited in the AlphaSort [NBC 95]. An additional promising technique is to run internal activities not one record at a time but in groups of records, as this may reduce cache faults for instructions and for global data structures. Candidate activities include writing a record to a run, obtaining an input record, inserting a record into the in-memory data structures, etc.

## 3 External sort: reducing and managing I/O

If the sort input is too large for an in-memory sort, external sorting is needed. This section discusses a number of techniques to improve the performance of external sorts, including in-memory data structures, space allocation heuristics for disks, and many other issues.

One might presume that all that needs to be done is to improve the I/O performance during merging. While I/O bandwidth is important, there is much more to fast external sort. Since I/O bandwidth can often be improved simply by adding more striping runs (e.g., the temp space in a database system) over more disk drives, a well-implemented external sort also employs a variety of techniques to reduce CPU consumption, both CPU instructions and cache faults.

### 3.1 Partitioning vs. merging

Just as internal sorting can be based either on distribution or on merging, the same is true for external sorting. An external sort based on partitioning is actually quite similar to hash join (or actually hash aggregation or duplicate removal, given that there is only one input), except that the hash function must be order-preserving and that output from the final in-memory partitions must be sorted before it is produced. The same set of variants that apply to hash operations apply with pretty much the same effect, in particular hybrid hashing for graceful degradation when memory is almost but not quite large enough, but also dynamic de-staging to deal with unpredictable input sizes, bucket tuning to deal with input skew, etc. [KNT 89]. Surprisingly, one of the complex real-world difficulties of hash operations, namely "bail-out" execution strategies in case partitioning does not produce buckets smaller than the allocated memory due to a frequent value, could be neatly integrated into sort operations based on partitioning – if a partition cannot be split further, this partition can be produced immediately as sort output.

Interestingly, if two inputs need to be sorted for a merge join, they could be sorted in a single interleaved operation if that sort operation is based on partitioning. Techniques used in hash join could then readily be adapted, e.g., bit vector filtering on each partitioning or merging level. Even hash teams [GBC 98] for more than 2 inputs could be adapted to sorting and merge-based set operations. Nonetheless, partition-based sorting has not been explored much in database sorting, partially because it hasn't been done before. However, in light of all the techniques that have been developed for hash operations, it seems to be worth a new thorough analysis.

### 3.2 Run generation

Quicksort and internal merge sort, when used to generate initial runs for an external merge sort, produce runs the size of the sort's memory allocation. Alternatively, replacement selection based on a priority heap produces runs about twice as large. One of the very desirable side benefits of replacement selection is that the entire run generation process is continuous, alternately consuming input pages and producing run pages, rather than cyclic with distinct read, sort, and write phases. In a database query processor, this steady behavior not only permits running concurrently the input query plan and the disks for temporary run files but also has very desirable effects in parallel query plans and parallel sorting, as will be discussed later.

There are two related issues with replacement selection that don't apply to quicksort and internal merge sort. First, quicksort and internal merge sort read and write data one entire memory load at a time. Thus, run generation can be implemented such that each record is copied within memory only once, when assembling sorted run pages. Second, memory management for quicksort and internal merge sort is straightforward, even for inputs with variable length records. Replacement selection, on the other hand, because it replaces records in the in-memory work space one at a time, free space management for variable length records requires as well as at least two copy operations per record (to and from the workspace). Fortunately, fairly simple techniques seem to permit memory utilization around 90% without any additional copy steps due to memory management [LG 98]. However, such memory management schemes imply that the number of records in the workspace and the number of valid entries in the priority heap may fluctuate. Thus, it is required that the heap implementation can accommodate efficiently temporary absence of entries, if possible without requiring multiple root-to-leaf passes over the priority heap.

Runs should be written to disk only as necessary to create space for additional input records. In database query processing, before execution it often is not known precisely how many pages, runs, etc. will be needed in a sort operation. For example, if the input is only slightly larger than memory, only very few records and pages should be written to disk. In general, the second-to-last run should only partially be written to disk. The remainder of the input should remain in memory and not incur any I/O costs. This idea is well known as hybrid hashing or dynamic de-staging for hash-based algorithms [DKO 84, NKT 88], and is equally applicable to sorting. In order to manage memory effectively for such on-demand de-staging of initial runs on disk, the same techniques can be used that are effective for memory management during replacement selection.

There are several ways to accommodate or exploit CPU caches during run generation. One of them was mentioned earlier: creating multiple cache-size runs in memory and merging them into memory-size initial run on disk. The cache-size runs can be created using a load-sort-write (to memory) algorithm or (cache-size) replacement selection. If poor man's normalized keys are employed, it is probably sufficient if the indirection array with pointers and poor man's normalized keys fits into the cache, because record accesses should be rare. Actually, any size is satisfactory if each small run as well as the priority heap for merging these runs into a single on-disk run fit in the cache – a single page or I/O unit might be a convenient size [ZL 97].

Another way to reduce cache faults on code and on global data structures runs the various activities not for each record but in bursts of records. Such activities include obtaining (pointers to) new input records, finding space in the workspace, copying records into the workspace, inserting new keys into the priority heap used for replacement selection, etc. This technique can reduce cache faults in both instruction and data caches, and is applicable to other modules in the database server, e.g., lock manager, buffer manager, log manager, output formatting, network interaction, etc. However, batched processing is probably not a good idea for key replacement in priority heaps, because those are typically implemented in data structures that favor replacement of keys over separate deletion and insertion.

## 3.3 Priority heaps

After runs have been created, they must be merged, and it important to design algorithms and data structures that make efficient use of instructions and CPU caches. Since the total size of the code affects the number of faults in the instruction cache, code reuse is also a performance issue in addition to software engineering and development cost. A single implementation of a priority heap may be used for many functions, e.g., run generation, merging cache-size runs in memory, merging disk-based runs, forecasting the most effective read-ahead I/O, planning the merge pattern, and virtual concatenation (the latter three issues will be discussed shortly). A single module used so heavily must be thoroughly profiled and optimized, but it also probably improves the effectiveness of the instruction cache.

When merging runs, most implementations use a priority heap implemented as a binary tree embedded in an array. For a given entry, say at array index *i*, the children are at *2i and at 2i+1*, or a minor variation of this scheme. Many implementations use a *tree of winners* [K 98], with the invariant that any node contains the smallest key of the entire tree rooted at that node. Thus, the tree requires twice as many nodes as it contains actual entries, e.g., records in the workspace during run generation or input runs during a merge step. In a *tree of losers* [K 98], each node contains a different entry. In this tree, there is a special root that has only one child, whereas all other nodes have two children. The invariant is that any node (except the special root) contains the second-smallest key of the entire tree rooted at that node; the smallest key is further towards the root or in the special root. To make this work for the tree's leaves, each leaf represents two merge runs.

When inserting, deleting, or replacing keys in the tree, many implementations employ passes from the tree's root to one of its leaves. Note that a pass from the root requires two comparisons per level, because an entry must exchange places with the smaller one of its two children. Passes from the leaves to the root, on the other hand, require only one comparison per tree level. In trees of losers, leaf-to-root passes are the usual technique. In trees of winners, most implementation use both root-to-leaf and leaf-to-root passes, although it is possible to rely entirely on leaf-to-root passes by defining a fixed mapping from entries (such as input runs during a merge step) to leaves.

Either kind of priority heap is a variant of a binary tree. When the nodes of a binary tree is fitted into larger physical units, e.g., disk pages or cache lines, entire units are moved in the memory hierarchy but only a fraction of the unit is truly exploited in each access. For disk-based search trees, B-trees were invented. B-trees with nodes equal to cache lines have shown promise in some experiments, where each node contains $2^n$ or $2^n$-1 entries. Priority heaps can similarly be adapted to employ nodes the size of cache lines [NBC 95], with some additional space in each node to point to the node's parent or with some additional complexity to compute the location of a node's child or parent. However, it isn't clear whether run generation is more effective using such modified priority heaps or limiting the size of the entire priority heap to the cache size, thus creating cache-size runs in memory and merging such cache-size runs into a single memory-size run while writing it to disk.

In order to make comparisons in the priority heap fast, heap entries should contain poor man's normalized keys, with the goal that poor man's normalized keys eliminate all comparison logic except when two valid records must be compared on their entire keys. In other words, the poor man's normalized key is more than simply the first few bytes of the normalized key. For fast comparisons between valid and invalid entries in the heap, invalid entries should have special "sentinel" values as their poor man's normalized keys. It is practical to have both early and late sentinel values, i.e., values that compare lower or higher than all poor man's normalized keys for valid entries. In order to simplify the comparison logic, two invalid entries in the priority heap, i.e., two sentinel values, should never compare as equal. Thus, each possible heap entry (each record slot in the workspace during run generation or each input run during a merge step) should have its own early and its own late sentinel value. During run generation, when record designated for two runs can be in the heap at the same time, the poor man's normalized key should also encode the run number of valid entries, such that records designated for different runs compare correctly based solely on their poor man's normalized keys. Note that the run number can be determined when a record is first inserted into the priority heap, which is when its poor man's normalized key value to be used in the priority heap is determined.

Let us consider an example. Presume a data structure that allows 16 bits or $2^{16}$ possible (non-negative) values for poor man's normalized keys, including sentinels etc. Let the heap size be $2^{10}$ entries, i.e., records in the workspace or runs to be merged. The lowest $2^{10}$ possible values and highest $2^{10}$ possible values are reserved as sentinels, one for each record or input run. Thus, $2^{16}$-$2^{11}$ values can be used as poor man's normalized keys for valid records, although pragmatically one might use only $2^{15}$ values or 15 bits from each actual key value in the poor man's normalized keys within the priority heap. If the priority heap is used to generate initial runs of an external sort, one might want to use only 12 bits, leaving the remaining 3 bits to represent run numbers. Thus, when the normalized key for an input record contains the value 47 in the leading 12 bits and the record is assigned to run 5, its poor man's normalized key in the priority heap is $2^{10}$ + 5 * $2^{12}$ + 47. The first term skips over the low sentinel values, the second term captures the run number suitably shifted such that the run number is more important than the record's actual key value, and the third term represents the record's actual sort key. Note that every 7 runs ($2^3$-1), a quick sweep through the entire heap would be required to reduce all such run numbers by 7. An alternative design employs only one bit to indicate a record's designated run, capturing more bits from the record keys in the poor man's normalized keys but requiring sweeps through the heap after every run.

Prior to forming the special poor man's normalized key for use in the priority heap, a prefix of a record's key can be used to speed several decisions for which conservative approximations suffice. For example, during run generation, the poor man's normalized key alone might determine whether an input record belongs into the current or the next initial run. Note that an input record must be assigned to the next initial run if its poor man's normalized key is equal to that of the record most recently written to the current run – a tradeoff between fast decisions and a small loss in accuracy and run length. Similarly, when replacing a record in the priority heap with its successor, one might want to repair the heap either by a root-to-leaf pass or a leaf-to-root pass, depending on the incoming key, the key it replaced, and the key in the appropriate leaf of the priority heap. Finally, poor man's normalized keys might also be used for planning virtual concatenation (discussed shortly).

Typically, the size of the priority heap is chosen as small as possible, e.g., the heap size is equal to the number of runs being merged. However, while merging runs, in particular runs of very different sizes, it might be useful to use a large priority heap and reserve multiple entry points in the priority heap for each run, although only one of these points is actually used, with the objective to minimize the number of key comparisons for the many keys in the largest runs. For example, the number of entry points reserved for each run might be proportional to the run's size. The effect of this policy is to balance the number of key comparisons that each run participates in, not counting the inexpensive comparisons decided entirely based on sentinel values. In particular, the many records from a large run participate in fewer comparisons per record.

A special technique can be exploited if one of the runs is so large that its size is a multiple of the sum of all other runs in the merge step. In fact, this run should not participate in the priority heap at all. Instead, each key resulting from merging all other runs should be located among the next records of the large run, e.g., using a binary search. The effect should be that many records from the large run do not participate in any comparisons at all. For example, presume one run of 1,000 records has been created with $N \log_2 N$ or about 10,000 comparisons, and another run of 1,000,000 records with about 20,000,000 comparisons. A traditional merge operation of these two would require about 1,001,000 comparisons, for a total of 21,011,000 comparisons. However, theoretically a run of 1,001,000 records could be created using only about 20,020,000 comparisons, i.e., the merge step should have required only 20,020,000 – 20,000,000 – 10,000 = 10,000 comparisons. Creating the merge output by searching for 1,000 correct positions among 1,000,000 records can be achieved with about 20,000 comparisons using straightforward binary search and probably much less using interpolation search – close to the number suggested by applying the $N \log_2 N$ formula to the three run sizes.

## 3.4 Merge patterns and merge optimizations

It is a widely held belief that given today's memory sizes, all external sort operations use only a single merge step. Therefore, optimizing merge patterns seems no more than an academic exercise. If a sort operation is used to create an index, the belief is justified except for the largest data warehouse tables. However, if the sort operation is part of a complex query plan that pipes data among multiple sort or hash operations all competing for memory at the same time, and in particular if nested queries employ sorting, e.g., for grouping or duplicate removal, multi-level merging is not uncommon. This is particularly true if a query processor employs eager or semi-eager merging (discussed in detail in [G 93]), which interleave merge steps with run generation. The problem with eager merging is that the operations producing the sort input may compete with the sort for memory, thus forcing merge operations with less than all memory.

One reason for using eager and semi-eager merging is to limit the number of runs existing at one time, for example because that permits managing all existing runs with a fixed amount of memory, e.g., one or two pages. Probably a better solution uses a file of run descriptors, with as many pages as necessary. For planning, only two pages full of descriptors are considered, and an additional page of run descriptors is brought into memory only when the number of runs has been reduced sufficiently that the remaining descriptors fit on a single page.

The goal of merge optimizations is to perform as few merge steps as possible and to move as few records as possible while doing so. In other words, one very effective heuristic is to always merge the smallest existing runs. All merge steps should use maximal fan-in, except in the first merge step. However, if not all existing runs are considered during merge planning (e.g., because some merge steps precede the end of the input or because the directory of runs exceeds the memory dedicated to merge planning), alternative heuristics may be better, for example to always merge the runs of the most similar size, independent of their absolute size. This latter heuristic attempts to ensure that any merge output run of size *N* has required no more sorting and merge effort than *N log N*.

The merge planning effort should also attempt to avoid merge steps altogether. If the records in two or more runs have non-overlapping key ranges, these runs can be combined into a single run without any effort. Rather than concatenating files by moving pages on disk, it is sufficient to simply declare all those files a virtual run and to scan all files that make up a virtual when actually merging runs. Planning such *virtual concatenation* can be implemented relatively easily by retaining low and high (poor man's normalized) keys in each run descriptor and using a priority heap that sorts all available low and high keys, i.e., twice as many keys as there are runs. While virtual concatenation is not very promising for random inputs, it is extremely effective for almost sorted inputs, in particular inputs sorted on only a prefix of the desired sort key, as well as for reverse sorted inputs.

## 3.5 Graceful degradation

One of the most important merge optimizations applies not to the largest inputs but to the smallest external merge sorts. If an input is just barely too large to be sorted in memory, many sort implementations spill the entire input to disk. Better implementations spill only as much as absolutely necessary. For example, if the input is only 10 pages larger than the available sort memory of 1,000 pages, only about 10 pages should be spilled to disk. The total I/O for the entire sort should be about 20 pages, not 2,020 pages as it is in many existing implementations.

Obviously, for inputs just a little larger than the available memory, this represents a substantial performance gain. Just as important but often overlooked, however, is the effect on resource planning in query evaluation plans with multiple memory-intensive sort and hash operations. If the sort operation's cost function has a stark discontinuity, a fair amount of special-case code in the memory management policy must be designed to reliably avoid fairly small but fairly expensive sorts. If, on the other hand, the cost function is smooth because both CPU load and I/O load grow continuously, implementing an effective memory allocation policy is must more straightforward.

In order to achieve graceful degradation, the last two runs generated must be special. The last run remains in memory and is never written to a run file, and the run prior to that is cut short when enough memory is freed for the first (or only) merge step. If run generation employs a read-sort-write cycle, the read and write phases must actually be a single phase with interleaved reading and writing, such that writing run files can stop as soon as reading input reaches the end of the input.

## 3.6 I/O optimizations

Finally, there are I/O optimizations. Some of them are quite obvious but embarrassingly not always considered – for example, a fair amount of time can be wasted if disk space is zeroed before it is used for run files, although it might be necessary to do so after the sort for security or privacy reasons. Similarly, files and file systems used by database systems typically should not use the file system buffer (in addition to the database buffer) and should not use virtual device drivers (e.g., for virus protection) or the like. Finally, updates to run files should not be logged, and using RAID devices for run files seems rather wasteful in space, processing, and bandwidth. These issues might seem utterly obvious to many but these guidelines are violated in many implementations and installations nonetheless.

It is well known that sequential I/O achieves much higher disk bandwidth than random I/O. Sorting cannot progress with pure sequential I/O; therefore, large I/O units are a good compromise between bandwidth and merge fan-in. Depending on the specific machine configuration and I/O hardware, 64 KB is typically a reasonable compromise on today's server machines, although somewhat larger units of I/O may also make sense. If CPU processing bandwidth is not an issue, the optimal I/O unit achieves the maximal product of bandwidth and the logarithm of the merge fan-in. Intuitively this is the right tradeoff because it enables the maximal number of useful key comparisons per unit of time.

Of course, there are reasons to deviate from this simple heuristic, in particular if merge input runs have different sizes and if the disk layout is known. It appears from a preliminary analysis that the minimal number of disk seeks for runs of different sizes is achieved if the I/O unit of each run as well as the number of seek operations per run are proportional to the square root of the run size. If the disk layout is known, larger I/O operations can be planned either before a merge step based on key distributions saved for each run or dynamically as the merge progresses [ZL 98].

Given that most database servers have many more disk drives than CPUs, typically by roughly one order of magnitude, either many threads or asynchronous I/O needs to be used to achieve full performance. Asynchronous write-behind while writing run files is fairly straightforward, thus half the I/O activity can readily exploit asynchronous I/O. However, effective read-ahead requires forecasting the right run to read from. A single asynchronous read can be forecasted correctly by comparing the highest keys in all current input buffers. If, as is typical, multiple disk drives are to be exploited, multiple reads must be forecasted. A possible simple heuristic is to extend the standard single-page forecast to multiple pages, although the resulting forecasts may be wrong, in particular if the merge input runs differ greatly in size and multiple pages from a single run ought to be fetched. Alternatively, the sort can retain key information about all pages in all runs, either in locations separate from the runs or as part of the runs themselves. If the latter, note that such runs strongly resemble the leaf level and the first parent level of a B-tree. Rather than designing a special storage structure and writing special code for run files, one might want to reuse the entire B-tree code for managing runs. The additional cost of doing so should be minimal given that typically 99% of a B-tree's allocated pages are leaves, and 99% of the remainder are immediate parents. One might also want to reuse B-tree code because of its cache-optimized page structures, use of poor man's normalized keys in B-trees [GL 01] and, of course, multi-page read-ahead directed by the parent level implemented for ordinary index-order scans.

While higher RAID levels with redundancy are a bad idea for sort runs, disk striping without redundancy is a good idea for sorting. The simplest way to exploit many disks is to simply stripe all runs similarly over all disks, in units equal to or a small multiple of the basic I/O unit, i.e., 64 KB to 1 MB. Larger striping units dilute the automatic load balancing effect of striping. Such simple striping is probably very robust and offers most of the achievable performance benefit. Note that both writing and reading merge runs ought to exploit striping and I/O parallelism. If, however, each run is assigned to a specific disk or to a specific disk array among many, forecasting per disk or disk array is probably most effective.

# 4 Sorting in context: database query processing

The techniques described so far apply to any large sort operation, whether in a database system or not. This section adds considerations for sorting in database query processors and its role in processing complex ad-hoc queries.

## *4.1 Sorting to reduce the data volume*

Sorting is often used to group records by a common value. Typical examples include aggregation (with prior grouping) and duplicate removal, but most of the considerations here also apply to "top" or "rank" operations, in particular grouped "top" and "rank" operations. An example for the latter is the query to find the top sales people in each region – one reasonable implementation sorts sales people by their region and sales volume. Performing the desired operation ("top") not only after but also during the sort operation can substantially speed up all these operations. The required logic can be invoked while writing run files, both initial runs and intermediate runs, and while producing the final output [BD 83]. The effect is that no intermediate run can be larger than the final output of the aggregation or "top" operation; thus, (presuming randomly distributed input keys) it is effective if the data reduction factor due to aggregation or "top" is larger than the merge fan-in of the final merge step.

Even if the sort operation does not reduce the data volume, records that have once been compared as equal should never be compared again. For example, if multiple records in a run file compare as equal, they should move through the next merge step together. Thus, the merge logic should never be more expensive than it is for sorting with duplicate removal. Thus, if there is a chance that records in the same run file will compare as equal, such groups of records should be identified as the file is being written, whether or not duplicate records are actually retained or removed.

## *4.2 Pipelining among query evaluation iterators*

A complex query might require many operations to transform stored data into the desired and correct query result. Commercial database systems pass data within a single evaluation thread using some variant of iterators [G 93]. Iterators can be data-driven or demand-driven, and their unit of iteration can be a single record or a group of records. Such a group defined by their data volume, the record count, and a common attribute value – the latter concept is known as *value packets* [K 80]. Obviously, the sort operation is a prime candidate to produce its output in value packets.

More recently, an additional reason for using multiple records at a time has emerged, namely CPU caches. For example, if a certain selection predicate requires the expression evaluator as well as certain constants, it might be advantageous to load the expression evaluator and those constants into the CPU caches only once every few records rather than for every single record.

Such batching will reduce cache faults both for instructions and for global data structures. As mentioned earlier, this technique can also be exploited for activities within a sort operation, e.g., inserting variable-length records into the sort operation's workspace. However, it probably is not a good idea to batch record replacement actions in the priority heap, since the leaf-to-root pass through a tree of losers is designed to repair the heap after deletion of a record as well as include a new record. If this activity is batched, multiple passes through the priority heap will replace each such leaf-to-root pass, since the heap must be repaired and the heap invariants re-established after each deletion and insertion. Moreover, these passes probably include a root-to-leaf pass, which is more expensive than a leaf-to-root pass since each level in the binary tree requires two comparisons rather than one.

A sort operation, a stop-and-go operation that consumes its entire input before producing output, can be implemented equally well using data-driven or demand-driven dataflow. Demand-driven dataflow is superior for single-threaded query execution, in particular for binary operations such as joins, whereas data-driven dataflow is often superior in multi-threaded execution and for operation with multiple outputs. For example, consider optimizing a large update of a table with multiple indexes. After a read-only query plan determines the affected rows as well as old values and new values, multiple branches in the query plan consume the same intermediate query result, sort it suitably for each index, and then apply all required changes in a single index-order pass over each index, thus ensuring that each index leaf page is touched at most once. In this case, rather than storing the unsorted intermediate result and scanning it multiple times, it might be more efficient to push the intermediate result rows immediately into the various plan branches and therefore into the various sort operations, immediately producing initial run files, albeit smaller ones than they could be, due to memory contention among the sort operations.

In a well-indexed database, it is amazing how often the optimal or at least some near-optimal query plan relies entirely on index navigation[2] rather than set-oriented operations such as merge join, hash join, and their variants. Typically, such a plan is not simply a sequence of index lookups but a clever assembly of more or less complex nested iterations, whether the original query formulation employed nested subqueries or not. Interestingly, sorting can be exploited in a variety of ways to improve the performance of nested iterations.

Most obviously, if the binding (correlation variables) from the outer query block or iteration loop is not unique, the inner query block might be executed multiple times with identical values. One improvement is to insert at the root of the inner query plan a caching iterator that retains the mapping from binding values to (inner) query results. This cache is particularly effective, and less likely to require disk space and disk I/O, if all outer rows with the same binding value occur in immediate sequence – in other words, if the outer rows are grouped or sorted by their bindings values. An opportunistic variant of this technique does not perform a complete sort of the outer rows, but only performs in-memory run generation to improve the chances of locality either in this cache or even in the indexes searched by the inner query block. This variant could also be very useful in object-oriented databases for object id resolution and for object assembly. In some situations, e.g., merge-joining two inputs of very different sizes, directly processing the initial runs rather than producing a completely sorted join input can beat hash join even for unsorted input [G 91].

Note that the cache of results from prior nested invocations might not be an explicit data structure and operation specifically inserted into the query plan for this purpose. Instead, it might be the memory built up by a stop-and-go operator, e.g., a hash join or a sort. In that case, the sort operation must not "recycle" its memory and its final on-disk runs, even if the sort implementation usually deallocates its memory and disk space as quickly as possible. Retaining these resources enables fast and cheap "rewind" actions for sort operation.

Just as pipelining multiple rows up a query plan can be advantageous, it can be a good idea to bind multiple outer rows to the inner query plan – in the extreme case, the result is complete side-ways information passing or semi-join reduction [BC 81, SHP 96]. If this idea is used, multiple executions of the inner query plan are folded into one, with the result rows often in an order less than ideal for combining it with the outer query block; in that case, the outer rows can be tagged with sequence numbers, the sequence number made part of the bindings, and the outer and inner query plan results combined using an efficient merge join on the sequence number after sorting the entire inner result on that sequence number.

## 4.3 Memory management

Sorting is typically a stop-and-go operator, meaning it consumes its entire input before producing its first output. In addition to the input and output phases, it may perform a lot of work after consuming its input and before producing the final output. These three operator phases – run generation while consuming input, intermediate merges, and final merge producing output – and similar operator phases in other stop-and-go operators define plan phases. For example, in an ad-hoc query with two sort operators feeding data into a merge join which in turn feeds a third sort operator, one of the plan phases includes two final merge steps, the merge join, and one initial run generation step.

Quite obviously, operators compete for memory and other resources only if they participate in a common plan phase, and only with some of their operator phases. In general, it makes sense to allocate memory to competing sort operations proportional their input data volume, even if (in extreme cases) this policy results in some sort operations having to produce initial runs with only a single buffer or other sort operations having to perform a “one-way” final merge, i.e., the last “intermediate” merge step gathers all output into a run and the final step simply scans that run.

This general heuristic needs to be refined for a number of cases. First, hash operations and query plans that combine sort and hash operations require different heuristics. Second, since sequential activation of query operators in a single thread may leave some sort operators dormant for extended periods of time, their memory must be made available to other operators that can use it more effectively. A typical example is a merge join with two sort operations for its input, where the input sorted first is actually small enough that it could be kept in memory if sorting the other sort does not require the same memory. Third, complex query plans using nested iteration need a more sophisticated model of operator and plan phases. Finally, some sort operations are not stop-and-go operations. We will consider only the last of these issues here.

If an input is almost sorted in the desired order, for example if it is sorted on the first few but not all desired sort attributes, it is often more efficient to run the core sort algorithm multiple times for segments of the input data rather than once for the entire data set. Such a *major-minor sort* is particularly advantageous if each of the single-segment sorts can be an in-memory sort whereas the complete sort cannot. On the other hand, if the input segments are so large that each requires an external sort, segment-by-segment sorting might not be optimal, because the sort competes with the sort operation’s producer and consumer operations for memory. A stop-and-go sort operation competes during its input phase with its producer and during its output phase with its consumer; but during the intermediate merge steps, it can employ all available memory. Note that the same situation that enables major-minor sort also makes virtual concatenation very effective, such that even the largest input might require no intermediate merge steps.

There are multiple proposals for dynamic memory adjustment during sorting, e.g., [PCL 93, ZL 97]. Techniques differ in adjusting memory based on only a single sort or multiple concurrent sort operations, while generating runs or only between runs, during a single merge step or only between merge steps, and by means of adjusting the I/O unit or the merge fan-in. Dropping or adding a merge input half-way through a merge step actually seems promising if virtual concatenation is employed, i.e., the merge policy can deal effectively with partial remainders of runs. In general, however, product developers tend to shy away from dynamic techniques because they expand the test matrix and create challenges when reproducing customer concerns.

## *4.4 Index creation and index maintenance*

One very important purpose of sorting in database systems is fast creation of indexes, typically some variant of B-trees. In addition, sorting can be used in a variety of ways for B-tree maintenance. For example, when updating a column with a uniqueness constraint and a B-tree index to enforce it, simply updating a unique index row by row, using a delete and an insert operation for each row, might detect false (temporary) violations if multiple rows “exchange” their column value. Instead, *N* update actions can be split into delete and insert actions, all *2N* actions sorted on the column value, and then applied in such a way that there will be no false violations. For a second example, when updating so many rows that many index leaves will be affected multiple times, sorting the insert or delete set and applying B-tree changes in index order can be substantial performance gain, just like building a B-tree index in sort order is faster than using random insertions.

For large indexes, e.g., in a data warehouse, there might not be enough temporary space for all the run files, even if runs files or even individual pages in run files are “recycled” as soon as the merge process has consumed them. Some database systems therefore use the target space for runs, either by default or as an option. During the final merge, pages are recycled for the index being created. An obvious issue with this is that the target space is often on mirrored disks or on RAID disks, which does not help sort performance as discussed earlier. However, if the target space is the only space available, there might be no alternative to using it for the runs.

Another issue with sorting in the target space is that the final index might be very fragmented because pages are recycled effectively in random order. Thus, an index-order scan of such an index would incur many disk seeks. There are two possible solutions. First, the final merge can release pages to the global pool of available pages, and the index creation attempts to allocate large contiguous disk space from there. However, unless the allocation algorithm successfully searches for contiguous free space, most of the allocations will be in the same small size in which space is recycled from the merge. Second, space is recycled from initial runs to intermediate runs, among intermediate runs, and to the final index in larger units, typically a multiple of the I/O unit. For example, if the factor is 8, disk space equal to at most 8 times the size of memory might be held for such deferred group recycling, which is typically an acceptable overhead when creating large indexes. Fortunately, the two solutions can be combined, and should result in index-order scans with acceptable amounts of seek operations.

## *4.5 Parallelism & threading*

There is a fair amount of literature that covers parallel sorting, both internal and external parallel sorts. One key issue is load balancing when using range partitioning. In practice, however, hash partitioning is typically the better choice, because it works nicely with merge join and sort-based duplicate removal, and user-requested sorted output can be obtained with a simple final merge operation that is typically as fast as or faster than the application program consuming the output. For index creation, the partition boundaries are typically determined prior to the sort and based on considerations beyond of the final index, not the creation process, and partitioning based on those boundaries followed by local sorts is typically sufficient.

Maybe a more important issue with parallel sort-based query execution plans is the danger of deadlocks due to unusual data distributions and limited buffer in the data exchange mechanism. Some simple forms of deadlocks are described in [G 93], but more complex forms also exist, e.g., over multiple groups of sibling threads and multiple data exchange steps. Typical deadlock avoidance strategies include alternative query plans, artificial keys that unblock the merge logic on the consumer side of the data exchange, and unlimited (disk-backed) buffers in the data exchange mechanism.

Parallel query plans work best, in general, if all data flow is balanced among all parallel threads and steady over time. Thus, a core sort algorithm is more suitable to parallel execution if it consumes and produces its output in steady flows. Merge sort naturally produces its output in a steady flow, but alternative run generation techniques result in different patterns of input consumption. Consider a parallel sort where a single, possibly parallel scan produces the input for all parallel sort threads. If these sort threads employ an algorithm with distinct read, sort, and write phases, each thread stops accepting input during its sort and write phases. If the data exchange mechanism on the input side of the sort uses a bounded buffer, one thread's sort phase can stop the data exchange among all threads and thus all sort threads. One alternative to distinct read, sort, and write phases is replacement selection. Another alternative is to divide memory into thirds and create separate threads rather than distinct phases. A third alternative creates very small runs in memory as soon as a suitable number of new input records are available and merges those runs into a memory-size disk-based run file on demand as memory is needed for new input records – an algorithm noted earlier for its CPU cache efficiency.

If parallel sorting is employed for grouping or duplicate removal, and if data needs to be re-partitioned to from multiple scan threads to multiple sort threads, and if data transfer between threads is not free, it might be useful to form initial runs in the scan threads, as part of the data pipeline leading to the data exchange operation. If duplicates are detected in these runs, they can be removed prior to re-partitioning, thus saving transfer costs. Of course, the receiving sort threads have to merge runs and continue removing duplicates. This idea of "local and global aggregation" is well known for hash-based query plans, but typically not used in sort-based query plans, because most sort implementations do not permit splitting run generation from merging. It might be interesting to separate run generation and merging into two separate iterators; note also that the "run generation" iterator is precisely what is needed to complement hash-partitioning using an order-preserving hash function to realize a disk-based distribution sort.

## *4.6 Query planning techniques*

While planning a query plan that includes a sort operation, there are a number of simplifications that ought to be considered, even if they are often ignored in today's products. For example, bit vector filtering applies not only to hash-based or parallel query plans but to any query plan that matches rows from multiple inputs and employs stop-and-go operations such as sort and hash join. In fact, if two inputs are sorted for a merge join, it might even be possible to apply bit vector filtering in both directions by coordinating the two sort operations' merge phases and merge levels.

If an order-by list contains keys, and if constraints are enforced in the database from which the data are sorted, functional dependencies within the order-by list can be exploited [SSM 96]. Specifically, a column can be removed from the order-by list if it is functionally dependent on the columns that appear earlier in the order-by list. Incidentally, this technique also applies to hash-based algorithms and to partitioning in parallel query plans. A constant column is presumed to be functionally dependent on the empty set, and can therefore always be removed. If the sort input is the result of a join operation, equivalence classes of column ought to be considered. Note that, in addition to primary key constraints on stored tables, functional dependencies also exist for intermediate results. For example, a grouping or distinct operation creates a new key for its output, namely the group-by list.

In addition to outright removal of columns, it may pay to reorder columns in the order-by list. For example, if the sort operation's purpose is to form groups or to remove duplicates, the order-by list can be treated as a set, i.e., the sequence of columns does not matter for the grouping operation, although it might matter if the query optimizer considers "interesting orderings" [SAC 79]. Note that in hash-based algorithms, the hashing columns always form a set, not a list. Thus, in cases in which both sorting and hashing are viable algorithms, these optimizations apply. The goal of reordering the column list is to move columns to the front that are inexpensive to compare, are easily mapped to poor man's normalized keys, and have many distinct values. For example, if the first column in the order-by list is a long string with a complex collation sequence and very few distinct values (known from database statistics or from a referential constraint to a small table), a lot of time will be spent comparing equal bytes within these keys, even if normalized keys are used. In addition to reordering the order-by list, it is even possible to add an artificial column at the head of the list, e.g., an integer computed by hashing other columns in the order-by list – of course, this idea is very similar poor man's normalized keys, which has been discussed before.

## 5 Summary and conclusions

In summary, sorting can be used in all kinds of database systems for all kinds of tasks, e.g., query processing, index creation, and consistency checks. There is a large and varied set of techniques that can substantially improve the performance of sort operations. Many of these techniques are complementary to each other. In their entirety, they can speed up sorting and sort-based query plans by an order of magnitude in many cases. Hopefully, this survey will make these techniques more widely known and commonly exploited in research prototypes and in commercial products.

## Acknowledgements

A number of friends and colleagues have contributed many insightful comments to earlier drafts, particularly David Campbell, Ian Jose, Per-Åke Larson, and Prakash Sundaresan.

## References

[A 96] Ramesh C. Agarwal: A Super Scalar Sort Algorithm for RISC Processors. SIGMOD Conf. 1996: 240-246.

[AAC 97] Andrea C. Arpaci-Dusseau, Remzi Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, David A. Patterson: High-Performance Sorting on Networks of Workstations. SIGMOD Conf. 1997: 243-254.

[ALM 96] Gennady Antoshenkov, David B. Lomet, James Murray: Order Preserving Compression. ICDE 1996: 655-663.

[BC 81] Philip A. Bernstein, Dah-Ming W. Chiu: Using Semi-Joins to Solve Relational Queries. JACM 28(1): 25-40 (1981).

[BD 83] Dina Bitton, David J. DeWitt: Duplicate Record Elimination in Large Data Files. TODS 8(2): 255-265 (1983).

[BL 89] Jean-Loup Baer, Yi-Bing Lin: Improving Quicksort Performance with a Codeword Data Structure. TSE 15(5): 622-631 (1989).

[DKO 84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood: Implementation Techniques for Main Memory Database Systems. SIGMOD Conf. 1984: 1-8.

[G 91] Goetz Graefe: Heap-Filter Merge Join: A New Algorithm For Joining Medium-Size Inputs. TSE 17(9): 979-982 (1991).

[G 93] Goetz Graefe: Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25(2): 73-170 (1993).

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

[GL 01] Goetz Graefe, Per-Åke Larson: B-Tree Indexes and CPU Caches. ICDE 2001.

[K 80] Robert Kooi, The Optimization of Queries in Relational Databases, Ph.D. thesis, Case Western Reserve University, 1980.

[K 98] Donald E. Knuth, The Art of Computer Programming: Sorting and Searching. Addison Wesley Longman (1998).

[KNT 89] Masaru Kitsuregawa, Masaya Nakayama, Mikio Takagi: The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. VLDB 1989: 257-266.

[LG 98] Per-Åke Larson, Goetz Graefe: Memory Management During Run Generation in External Sorting. SIGMOD Conference 1998: 472-483.

[NBC 95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, David B. Lomet: AlphaSort: A Cache-Sensitive Parallel External Sort. VLDB Journal 4(4): 603-627 (1995).
[NKT 88] Masaya Nakayama, Masaru Kitsuregawa, Mikio Takagi: Hash-Partitioned Join Method Using Dynamic Destaging Strategy. VLDB 1988: 468-478.
[PCL 93] HweeHwa Pang, Michael J. Carey, Miron Livny: Memory-Adaptive External Sorting. VLDB 1993: 618-629.
[SAC 79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD Conf. 1979: 23-34.
[SHP 96] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, S. Sudarshan: Cost-Based Optimization for Magic: Algebra and Implementation. SIGMOD Conf. 1996: 435-446.
[SSM 96] David E. Simmen, Eugene J. Shekita, Timothy Malkemus: Fundamental Techniques for Order Optimization. EDBT 1996: 625-628.
[ZL 97] Weiye Zhang, Per-Åke Larson: Dynamic Memory Adjustment for External Mergesort. VLDB 1997: 376-385.
[ZL 98] Weiye Zhang, Per-Åke Larson: Buffering and Read-Ahead Strategies for External Mergesort. VLDB 1998: 523-533.

Additional references suggested by PALarson
Vinay S. Pai, Peter J. Varman: Prefetching with Multiple Disks for External Mergesort: Simulation and Analysis. ICDE 1992: 273-282.
Betty Salzberg: Merging Sorted Runs Using Large Main Memory. Acta Informatica 27(3): 195-215(1989).
LuoQuan Zheng, Per-Åke Larson: Speeding up External Mergesort. TKDE 8(2): 322-332(1996).
Vladimir Estivill-Castro, Derick Wood: A Survey of Adaptive Sorting Algorithms. ACM Computing Surveys 24(4): 441-476 (1992).
Andersson, A. and Nilsson, S., A new efficient radix sort, Proc. 35th Annual IEEE Symposium on Foundations of Computer Science (1994), pp. 714-721.
Arne Andersson, Stefan Nilsson: Implementing Radixsort. ACM Journal of Experimental Algorithms 3: 7 (1998).
Bentley, J. L. and McIlroy, M. D., Engineering a sort function, Software - Practice and Experience 23, 11 (1993), pp. 1249-1265.
McIlroy, P. M., Bostic, K., and McIlroy, M. D., Engineering radix sort, Computing Systems 6, 1 (1993), pp. 5-27.