Implementing Sorting in Database Systems

GOETZ GRAEFE

Microsoft

Most commercial database systems do (or should) exploit many sorting techniques that are publicly known, but not readily available in the research literature. These techniques improve both sort performance on modern computer systems and the ability to adapt gracefully to resource fluctuations in multiuser operations. This survey collects many of these techniques for easy reference by students, researchers, and product developers. It covers in-memory sorting, disk-based external sorting, and considerations that apply specifically to sorting in database systems.

Categories and Subject Descriptors: E.5 [**Data**]: Files—*Sorting/searching*; H.2.2 [**Database Management Systems**]: Access Methods; H.2.4 [**Database Management**]: Systems—*Query processing; relational databases*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Key normalization, key conditioning, compression, dynamic memory resource allocation, graceful degradation, nested iteration, asynchronous read-ahead, forecasting, index operations

1. INTRODUCTION

Every computer science student learns about $N \log N$ in-memory sorting algorithms as well as external merge-sort, and can read about them in many text books on data structures or the analysis of algorithms (e.g., Aho et al. [1983] and Cormen et al. [2001]). Not surprisingly, virtually all database products employ these algorithms for query processing and index creation. While these basic approaches to sort algorithms are widely used, implementations of sorting in commercial database systems differ substantially from one another, and the same is true among prototypes written by database researchers.

These differences are due to "all the clever tricks" that either are exploited or not. Many of these techniques are public knowledge, but not widely known. The purpose of this survey is to make them readily available to students, researchers, and industrial software developers. Rather than reviewing everything published about internal and external sorting, and providing another overview of well-published techniques, this survey focuses on techniques that seem practically useful, yet are often not well-understood by researchers and practitioners.

Author's address: G. Graefe, Microsoft, Inc., One Microsoft Way, Redmond, WA 98052-6399; email: goetzg@microsoft.com.

In order to be practically useful in a commercial database product, a sorting technique must be reasonably simple to maintain as well as both effective and robust for a wide range of workloads and operating conditions, since commercial database systems employ sorting for many purposes. The obvious purposes are for user-requested sorted query output, index creation for tables and materialized views, and query operations. Query operations with efficient sort-based algorithms include duplicate removal, verifying uniqueness, rank and top operations, grouping, roll-up and cube operations, and merge-join. Minor variations of merge-join exist for outer join, semijoin, intersection, union, and difference. In addition, sorting can be used for logical consistency checks (e.g., verifying a referential or foreign key constraint that requires each *line-item* row to indeed have an associated *order* row) and for physical consistency checks (e.g., verifying that rows in a table and records in a redundant index precisely match up) because both are essentially joins. Similarly, sorting may speed-up fetch operations following a nonclustered 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 mapping logical object ids to physical object ids (locations on disk) are forms of internal joins that may benefit from sorting. In a database system for graphs, unstructured data, or XML, sorting and sort-based algorithms can be used to match either nodes and edges or elements and relationships among elements. A specific example is the multipredicate merge-join [Zhang et al. 2001]. Finally, sorting can be used when compressing recovery logs or replication actions, as well as during media recovery while applying the transaction log. Many of these sort applications within relational database systems were well-known as early as the System R project [Härder 1977], and many were employed in database systems even before then. In spite of these many different uses, the focus here is on query processing and maintenance of B-tree indexes, since these two applications cover practically all the issues found in the others.

This survey is divided into three parts. First, in-memory sorting is considered. Assuming the reader knows and understands quicksort and priority queues implemented with binary heaps, techniques to speed in-memory sorting are discussed, for example, techniques related to CPU caches or speeding-up individual comparisons. Second, external sorting is considered. Again assuming that external merge-sort is well-known, variations of this theme are discussed in detail, for example, graceful degradation if the memory size is almost, but not quite, large enough for in-memory sorting. Finally, techniques are discussed that uniquely apply to sorting in the context of database query execution, for example, memory management in complex query plans with multiple pipelined sort operations or nested iteration. Query optimization, while obviously very important for database query performance, is not covered here, except for a few topics directly related to sorting.

The assumed database environment is a typical relational database. Records consist of multiple fields, each with its 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. It may include all fields, but typically does 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, in some places an ascending sort is assumed, but adaptation to descending sort or multiattribute mixed-sort is quite straightforward and not discussed further. Similarly, "stability" of sort algorithms is also ignored, that is, the guarantee that input records with equal keys appear in the output in the same sequence as in the input, since any sort algorithm can be made stable by appending a "rank" number to each key in the input.

2. INTERNAL SORT: AVOIDING AND SPEEDING COMPARISONS

Presuming that in-memory sorting is well-understood at the level of an introductory course in data structures, algorithms, or database systems, this section surveys only a few of the implementation techniques that deserve more attention than they usually receive. After briefly reviewing why comparison-based sort algorithms dominate practical implementations, this section reviews normalized keys (which speed comparisons), order-preserving compression (which shortens keys, including those stretched by normalization), cache-optimized techniques, and algorithms and data structures for replacement selection and priority queues.

2.1. Comparison-Based Sorting versus Distribution Sort

Traditionally, database sort implementations have used comparison-based sort algorithms, such as internal merge-sort or quicksort, rather than distribution sort or radix sort, which distribute data items to buckets based on the numeric interpretation of bytes in sort keys [Knuth 1998]. However, comparisons imply conditional branches, which in turn imply potential stalls in the CPU's execution pipeline. While modern CPUs benefit greatly from built-in branch prediction hardware, the entire point of key comparisons in a sort is that their outcome is not predictable. Thus, a sort that does not require comparisons seems rather attractive.

Radix and other distribution sorts are often discussed because they promise fewer pipeline stalls as well as fewer faults in the data cache and translation look-aside buffer [Rahman and Raman 2000, 2001]. Among the variants of distribution sort, one algorithm counts value occurrences in an initial pass over the data and then allocates precisely the right amount of storage to be used in a second pass that redistributes the data [Agarwal 1996]. Another variant moves elements among linked lists twice in each step [Andersson and Nilsson 1998]. Fairly obvious optimizations include stopping when a partition contains only one element, switching to an alternative sort method for partitions with only a few elements, and reducing the number of required partitions by observing the minimal and maximal actual values in a prior partitioning step.

Despite these optimizations of the basic algorithm, however, distribution-based 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 and the data contains duplicate keys, many passes over the data may be needed. For variable-length keys, the maximal length must be considered. If key normalization (explained shortly) is used, lengths might be both 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 of steps) before using a comparison-based sort algorithm on each resulting bucket [Arpaci-Dusseau et al. 1997].

Second, radix sort is most effective if data values are 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 byte, which implies uniform distribution. Of course, to achieve the correct sort order, the compression must be order-preserving. Third, if input records are nearly sorted, the keys in each memory load in a large external sort are similar in their leading bytes, rendering the initial passes of radix sort rather ineffective. Fourth, while a radix sort might reduce the number of pipeline stalls due to poorly predicted branches, cache efficiency might require

very small runs (the size of the CPU's cache, to be merged into initial disk-based runs), for which radix sort does not offer substantial advantages.

2.2. Normalized Keys

The cost of in-memory sorting is dominated by two operations: key comparisons (or other inspections of the keys, e.g., in radix sort) and data movement. Surrogates for data records, for example, pointers, typically address the latter issue—we will provide more details on this later. The former issue can be quite complex due to multiple columns within each key, each with its own type, length, collating sequence (e.g., case-insensitive German), sort direction (ascending or descending), etc.

Given that each record participates in many comparisons, it seems worthwhile to reformat each record both before and after sorting if the alternative format speeds-up the multiple operations in between. For example, when sorting a million records, each record will participate in more than 20 comparisons, and we can spend as many as 20 instructions to encode and decode each record for each instruction saved in a comparison. Note that each comparison might require hundreds of instructions if multiple columns, as well as their types, lengths, precision, and sort order must be considered. International strings and collation sequences can increase the cost per comparison by an order of magnitude.

The format that is most advantageous for fast comparisons is a simple binary string such that the transformation is both order-preserving and lossless. In other words, the entire complexity of key comparisons can be reduced to comparing binary strings, and the sorted output records can be recovered from the binary string. Since comparing two binary strings takes only tens of instructions, relational database systems have sorted using normalized keys as early as System R [Blasgen et al. 1977; Härder 1977]. Needless to say, hardware support is much easier to exploit if key comparisons are reduced to comparisons of binary strings.

Let us consider some example normalizations. Whereas these are just simple examples, alternative methods might add some form of compression. If there are multiple columns, their normalized encodings are simply concatenated. Descending sorts simply invert all bits. *NULL* values, as defined in SQL, are encoded by a single 0-bit if *NULL* values sort low. Note that a leading 1-bit must be added to non-*NULL* values of fields that may be *NULL* in some records. For an unsigned integer in binary format, the value itself is the encoding after reversing the byte order for high-endian integers. For signed integers in the usual B-1 complement, the first (sign) bit must be inverted-Floating-point numbers are encoded using first their overall (inverted) sign bit, then the exponent as a signed integer, and finally, the fractional value. The latter two components are placed in descending order if the overall sign bit is negative. For strings with international symbols, single and double-byte characters, locale-dependent sort orders, primary and secondary weights, etc., many modern operating systems or programming libraries provide built-in functions. These are usually controlled with large tables and can produce binary strings that are much larger than the original text string, but amenable to compression. Variable-length strings require a termination symbol that sorts lower than any valid data symbol in order to ensure that short strings sort lower than longer strings with the short string as the prefix. Creating an artificial termination symbol might force variable-length encodings.

Figure 1 illustrates the idea. The initial single bit indicates whether the leading key column contains a valid value. If this value is not null, it is stored in the next 32 bits. The following single bit indicates whether the second column contains a valid value. This value is shown here as text, but really ought to be stored binary, as appropriate for the desired international collation sequence. A string termination symbol marks

Integer Column	*String Column*	*Normalized Key*
2	"foo"	1 0…0 0000 0000 0010 1 "foo"\0
3	"bar"	1 0…0 0000 0000 0011 1 "bar"\0
1024	Null	1 0…0 0100 0000 0000 0
Null	""	0 1 \0

Fig. 1. Normalized keys.

the end of the string. If the string termination symbol can occur as a valid character in some strings, the binary representation must offer one more symbol than the alphabet contains. Notice the difference in representations between an empty string and a null in a string column.

Reformatting applies primarily to the key because it participates in the most frequent and costly operations. This is why this technique is often 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 for 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, particularly for lossless normalization and some international collation sequences, thus increasing the requirements for both memory and disk, space and bandwidth.

There are some remedies, however. If it is known *a priori* that some fields will never participate in comparisons, for example, because earlier fields in the sort key form a unique key for the record collection being sorted, the normalization for these fields does not need to preserve order; it just needs to enable fast copying of records and the recovery of original values. Moreover, a binary string is much easier to compress than a complex record with individual fields of different types—we will present more on order-preserving compression shortly.

In the remainder of this survey, normalized keys and records are assumed, and any discussion about applying the described techniques to traditional multifield records is omitted.

2.3. Order-Preserving Compression

Data compression can be used to save space and bandwidth in all levels of the memory hierarchy. Of the many compression schemes, most can be adapted to preserve the input's sort order, typically with a small loss in compression effectiveness. For example, a traditional Huffman code is created by successively merging two sets of symbols, starting with each symbol forming a singleton set and ending with a single set containing all symbols. The two sets to be merged are those with the lowest rates of occurrence. By restricting this rule to sets that are immediate neighbors in the desired sort order, an order-preserving compression scheme is obtained. While this algorithm fails to produce optimal encoding in some cases [Knuth 1998], it is almost as effective as the optimal algorithm [Hu and Tucker 1971], yet much simpler. Order-preserving Huffman compression compresses somewhat less effectively than traditional Huffman compression, but is still quite effective for most data.

As a very small example of order-preserving Huffman compression, assume an alphabet with the symbols 'a,' 'b,' and 'c,' with typical frequencies of 10, 40, and 20, respectively. Traditional Huffman code combines 'a' and 'c' into one bucket (with the same leading bit) such that the final encodings will be "00," "1," and "01," respectively. Order-preserving Huffman code can only combine an immediate neighbor, in this case

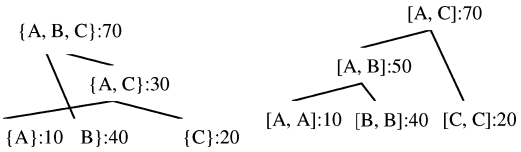

Fig. 2. Ordinary and order-preserving Huffman compression.

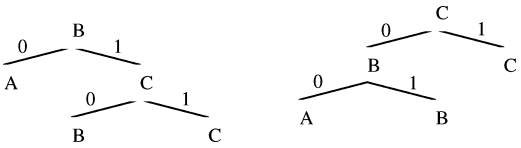

Fig. 3. Tree rotation in adaptive order-preserving Huffman coding.

'b,' with one of its neighbors. Thus, 'a' and 'b' will form the first bucket, with the final encodings "00," "01," and "1." For a string with frequencies as assumed, the compressed length is $10 \times 2 + 40 \times 1 + 20 \times 2 = 100$ bits in traditional Huffman coding and $10 \times 2 + 40 \times 2 + 20 \times 1 = 120$ bits in order-preserving Huffman coding, compared to $(10 + 40 + 20) \times 2 = 140$ uncompressed bits.

Figure 2 illustrates the two code-construction algorithms. Each node in the tree is labeled with the symbols it represents and their cumulative frequency. In ordinary Huffman compression, each node represents a set of symbols. The leaf nodes represent singleton sets. In order-preserving Huffman compression, each node in the tree represents a range. One important difference between the two trees is that the one on the right is free of intersecting lines when the leaf nodes are sorted in the desired order.

While the dynamic (adaptive) Huffman codes described in the literature do not preserve order [Lelewer and Hirschberg 1987; Vitter 1987], adaptive order-preserving Huffman coding is also easily possible based on order-preserving node rotations in a binary tree that is used for both encoding and decoding. Each leaf node contains a weight that captures how frequently or recently the symbol has been processed. Internal tree nodes aggregate the weight from their children. During encoding, the tree is traversed using key comparisons, while during decoding, branch selection is guided by bits in the compressed code. Both encoding and decoding recursively descend the tree, adjust all nodes' weights, and rotate nodes as suggested by the weights.

Consider, for example, the two encoding trees in Figure 3. The leaf nodes represent symbols and the root-to-leaf paths represent encodings. With a left branch encoded by a 0 and a right branch by a 1, the symbols "A," "B," and "C" have the encodings "0," "10," and "11," respectively. The internal nodes of the tree contain separator keys that are very similar to separator keys in B^+-trees. The left tree in Figure 3 is designed for relatively frequent "A" symbols. If the symbol "C" is particularly frequent, the encoding tree can be rotated into the right tree such that the symbols "A," "B," and "C" have encodings "00," "01," and "1," respectively. The rotation from the left tree in Figure 3 to the right tree is worthwhile if the accumulated weight in leaf node C is higher than that in leaf node A, that is, if the effective compression of leaf node C is more important than that of leaf node A. Note that the frequency of leaf node B irrelevant and unaffected by the rotation, and that this tree transformation is not suitable for minimizing the path to node B or the representation of B.

Encoding or decoding may start with an empty tree. In each key range that permits the addition of symbols, a new symbol reserves an encoding that indicates that a new

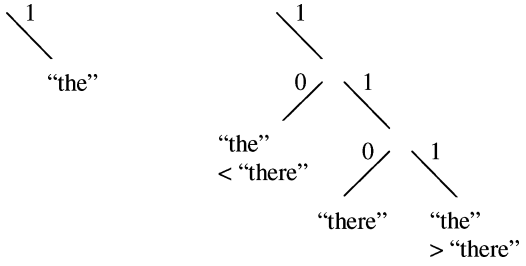

Fig. 4. Order-preserving dictionary compression.

symbol has been encountered for the first time. Alternatively, encoding or decoding may start with a tree constructed for static order-preserving Huffman compression based on a fixed sample of text. Hybrids of the two approaches are also possible, that is, starting with a nonempty tree and developing it further if necessary. Similarly, a binary tree with leaves containing strings rather than single characters can be used for order-preserving dynamic dictionary encoding. A separate parser must cut the input into encoding units, which are then encoded and decoded using a binary tree.

When run-length encoding and dictionary compression are modified to be order-preserving, the symbols following the substituted string must be considered. When a new string is inserted into the dictionary, the longest preexisting prefix of a new string must be assigned two encodings, rather than only one [Antoshenkov et al. 1996]. For example, assume that a dictionary already contains the string "the" with an appropriate code, and the string "there" is to be inserted into the dictionary with its own code. In this case, the string "the" must be assigned not one, but two codes: one for "the" strings followed by a string less than "re," and one for "the" strings followed by a string greater than "re." The encoding for "there" might be the value 124, and the two encodings for "the" are either 123 or 125, depending on its continuation. Using these three codes, the strings "then," "therefore," and "they" can be compressed based on the encodings. The prefix "the" within "then" requires code 123, whereas "the" within "they" requires code 125 such that "then," "therefore," and "they" can be sorted correctly.

Figure 4 illustrates the idea and combines it with earlier concepts about adaptive order-preserving Huffman compression. At some point, the string "the" has an encoding or bit pattern assigned to it in the example ending in "1." When the string "there" is introduced, the leaf node representation of "the" is expanded into a small subtree with 3 leaf nodes. Now, the compression of "the" in "then" ends in "10" and of "the" in "they" ends in "111." The compression of "there" in "therefore" ends in "110," which sorts correctly between the encodings of "then" and "they." The newly created subtree in Figure 4 is right-deep based on the assumption that future text will contain more occurrences of "the" sorting lower than "there" than occurrences sorting higher than "there." Subsequent tree rotations may optimize the compression scheme further.

Dictionary compression is particularly effective for long strings of padding characters (e.g., white space in fixed-size string fields) and for default values. Of course, it is also related to the normalization of *NULL* values, as described earlier. A useful extension uses multiple bits to compress *NULL*, default, and otherwise frequent values. For example, 2 bits (instead of only 1 bit for *NULL* values) permit one value for *NULL* values ("00") that sort lower than all valid data values, one for the default value ("10"), and two for actual values that are smaller or larger than the default value ("01" and "11"). For example, the value 0 is a frequent value in many numeric columns, so the 2-bit combination "10" may indicate the column value 0, which does not need to be stored explicitly, "01" indicates negative values, and "11" indicates positive values. If multiple frequent values are known *a priori*, say 7 values in addition to *NULL*, then

Integer Column	*Normalized Key*
2	10 0010
3	10 0011
1024	11 0…0 0100 0000 0000
Null	00

Fig. 5. Compression of integers using numeric ranges.

Value	*Code*	*Modified*	*Value*	*Code*	*Value*	*Code*
abc	255,a	254,b; 253,c	aaa	255,a	aaa	255,a
abd	253,d		aba	254,b	aba	254,b
ad	254,d		ae	254,e	abc	253,c
…	…		…	…	abd	253,d
					ad	254,d
					…	…

Fig. 6. Merge efficiency with offset-value coding.

twice as many encodings are required, say 16 encodings using 4 bits, such that half the encodings can serve for specific frequent values and half for the values in the intervals.

A related compression method applies specifically to integer, columns in which large values must be supported for exceptional situations, but in which most values are small. For example, if most actual values can be represented in a single-byte integer, but some very few values require eight-byte integers, then leading bits "00" may indicate a *NULL* value, "01" an eight-byte integer less than −128, "10" a single-byte positive or negative integer, and "11" a positive eight-byte integer value greater than 127. Obviously, such variable-length integers can be used in many variations, for example, if values are sure to be nonnegative, if more than two different sizes should be supported, if specific small ranges of large values are particularly frequent, or if specific individual values are particularly frequent. Figure 5 illustrates the point. In this example, the code "10" indicates a 4-bit integer in the range of 0 to 15. These values require only 6 bits, whereas all other values require 66 bits, except for null, which requires 2 bits.

Another compression method that is exploited effectively in commercial sort packages relies not on key encoding, but on key truncation (next-neighbor prefix truncation). Note that prefix truncation and order-preserving compression can be applied one after the other, in either order. In an internal or external merge-sort, each record is compared to its predecessor in the same run, and leading bytes that are equal to the preceding record are replaced by their count. For the first record in any run, there is an imagined leading record of zero length. The offset of the first difference is combined with the actual value at this location into a single-integer value, which motivates the name *offset-value coding* [Conner 1977]. In a merge of two inputs, offset-value codes are compared before any data bytes are compared, and suitable prefix lengths or offsets for the merge output can be computed efficiently from those in the merge inputs. Actually, during the merge process, the offsets and values used in comparisons capture the difference not with the prior record from the same merge input, but with the most recent output record, whichever input it may have come from. A merge of more than two runs can be implemented in a binary heap structure as multiple binary merges. Note, however, that offset-value codes are maintained separately for each binary merge within a multiway merge.

For a small example of offset-value coding, consider Figure 6. On the left and in the center are two sorted input streams, and the output is on the right. For each record in every run, both the original complete record is shown, as well as the offset-value

code. During the merge process, the code may be modified. These modifications are also shown in a separate column. The first record in each run has zero overlap with the preceding imaginary record of zero length. Thus, the highest possible byte count is assigned. In each subsequent record, the code is 255 minus the length of the overlap or the offset of the first difference.

After the first comparison finds that the two codes "255,a" and "255,a" are equal, the remaining parts of the strings are compared, and "aa" is found to be lower than "bc." Hence, "aaa" with "255,a" is produced as output. The code for "abc" is modified to "254,b" in order to reflect the decisive character comparison, which is also the correct code relative to the most recent output. Now, the leading records in both merge inputs have code "254,b," and again, a comparison of the remaining strings is required, that is "c" versus "a." Then, "aba" is moved to the merge output, and the code for "abc" becomes "253,c." In the next comparison, "abc" is found to be lower than "ae," based on the codes alone. In fact, the next three comparisons move records from the left input without any further string comparisons based on code comparisons alone. After these comparisons, there is no need to recompute the loser's offset-value code. Modifications of the codes are required only after comparisons that could not be decided based on the codes alone. The offset modification reflects the number of bytes that needed to be compared.

The net effect of offset-value coding, as can be observed in the example, is that any symbol within any string participates in—at most—one comparison during a binary merge, no matter how long the strings are and how much duplication of the leading prefixes exists in the two merge inputs. For successive keys that are completely equal, another optimization is discussed later in this article in the context of duplicate elimination during database query processing. Alternatively, a special offset-value code could be used to indicate that two keys have no difference at all.

Applying this idea not to merge-sort, but to quicksort requires using a single reference value for an entire partitioning step. This reference value ought to be the minimal value in the original partition. Otherwise, offsets and values must accommodate negative differences by using negative values [Baer and Lin 1989]. A further generalization uses not only truncation, but for numeric keys, subtraction from a base value called a *frame of reference* [Goldstein et al. 1998; Kwan and Baer 1985]. For example, if a given key column only contains values between 1,020 and 1,034, intermediate records can be slightly smaller and the sort slightly faster if 1,020 is subtracted from all values prior to the sort and added back after the sort is complete. This idea can be applied either per page or for an entire dataset. The former choice is particularly effective when applied to sorted data, such as runs. Note that columns with only a single constant value in the entire dataset will automatically be compressed to zero bits, that is, they will be eliminated from all comparisons.

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, compression other than truncation has generally not been used in database sorting, although it seems worthwhile to explore this combination, particularly if it is integrated with key normalization. Note that an order-preserving method is required only for the key. One of the obvious difficulties is that the same compression scheme has to be used for all records, and the distribution of values within an intermediate query result is often not known accurately when a sort operation 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 system software. Modern microprocessors can

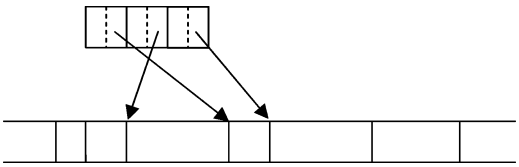

Fig. 7. In-memory runs from cache-sized sort operations.

theoretically complete as many as 9 instructions in a single cycle (although 1–3 instructions per cycle are more typical in practice due to various stalls and delays), and a single cache fault in the level-2 cache can cost 50 or even 100 CPU cycles, that is, the equivalent of up to hundreds of instructions. Thus, it is no surprise that performance engineers focus at least as much on cache faults as on instruction-path length. For example, reducing the instruction count for a comparison by 100 instructions is less effective than avoiding a single cache fault per comparison.

Cache faults for instructions are as expensive as cache faults for data. Thus, reducing code size is important, especially the code within the "inner" loops, such as comparisons. Normalized keys substantially reduce the amount of comparison code, whereas the code required for traditional complex comparisons might well exceed the level-1 instruction cache, particularly if international strings and collation sequences are involved. For data, beyond the obvious, for example, aligning data structures and records to cache-line boundaries, there are two principal sources of ideas. First, we can attempt to leave the full records in main memory (i.e., not access them) and use only record surrogates in the cache. Second, we can try to adapt and reapply to CPU caches any and all techniques used in the external sort to ameliorate the distance between memory and disk.

In order to avoid moving variable-length records and the ensuing memory management, most implementations of in-memory sorting use an array of pointers. Due to their heavy use, these pointers typically end up in the CPU 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 these, 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, each with a pointer and a key prefix. Moreover, if the type of prefix is fixed, such as an unsigned integer, prefix comparisons can be compiled into the sort code, instead of relying entirely on interpretive comparison functions. Since key normalization has been restricted to the first few bytes, these fixed-size prefixes of normalized keys have been called *poor man's normalized keys* [Graefe and Larson 2001].

These prefixes can be very effective if the 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. This is because the comparison of poor man's normalized keys is compiled into a few instructions of machine code, and the branch prediction hardware of modern CPUs will ensure that such useless predictable comparisons will not stall the CPU pipeline.

Figure 7 illustrates the two-level scheme used in AlphaSort [Nyberg et al. 1995]. An array with a fixed number of pairs containing a poor man's normalized key and a pointer, with the array sized to fit into the CPU cache, is sorted to form an in-memory run within a workspace filled with variable-length records. After multiple such sections of the workspace have been sorted into runs, they are merged and the result forms an initial on-disk run. The type of key-pointer pairs in the array is fixed for all sort operations, and can therefore be compiled into the sort code. Type, size, collation sequence, etc., are

considered when the poor man's normalized keys are extracted from the data records and assigned to array elements.

Alternative designs for order-preserving fixed-size, fixed-type keys use offset-value coding [Conner 1977] or one of its variants. One such variant starts with an arbitrarily chosen key value and represents each actual key value using the difference from that reference key [Baer and Lin 1989]. As in offset-value coding, the fixed-size key for each record is composed of two parts: first, the length of the reference key minus the length of the prefix equal in the actual and reference keys, then the value of the first symbol following this prefix in the reference key minus the corresponding value in the actual key. If the actual key sorts lower than the reference key, the first part is made negative. For example, if the chosen reference key is "acid," the actual key "ad" is encoded as $(+3, +1)$, since *length ("acid") − length ("a")* $= 3$ and *'d' − 'c'* $= 1$. Similarly, "ace" is encoded as $(-2, -5)$. This design works particularly well if many, but not all, actual keys share a sizable prefix with the reference key, and probably works best with partitioning-based sort algorithms such as quicksort [Baer and Lin 1989]. Moreover, if multiple reference keys are used for disjoint key ranges and the fixed-size, fixed-type key encodes the chosen reference key in its highest-order bits, such reference key encoding might also speed-up comparisons while merging runs, although traditional offset-value coding might still be more efficient for merge-sorts.

Among the techniques that can be adapted from external sorting to cache-optimized in-memory sorting, the most obvious is to create runs that are the size of the CPU cache and then to merge multiple such runs in memory before writing the result as a base-level run to disk. Poor man's normalized keys and cache-sized runs are two principal techniques exploited in AlphaSort [Nyberg et al. 1995]. Alternatively, Zhang and Larson [1997] proposed a method that is simple, adaptive, and cache-efficient: Sort each incoming data page into a minirun, and merge miniruns (and remove records from memory) as required to free space for incoming data pages or competing memory users. An additional promising strategy is to run internal activities not one record at a time, but in batches, as this may reduce cache faults instructions and global data structures [Padmanabhan et al. 2001]. Candidate activities include writing a record to a run, obtaining an input record, inserting a record into the in-memory data structures, etc.

2.5. Priority Queues and Binary Heaps

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 costs. From this point of view, using a single data structure for many purposes is a good idea. A single implementation of a priority queue may be used for many functions, for example, run generation, merging cache-sized runs in memory, merging disk-based runs, forecasting the most effective read-ahead I/O, planning the merge pattern, and virtual concatenation (the last three issues will be discussed shortly). A single module that is used so heavily must be thoroughly profiled and optimized, but it also offers great returns for any tuning efforts.

For example, a traditional belief holds that run generation using replacement selection in a priority queue requires twice as many comparisons as run generation using quicksort. A second customary belief holds that these comparisons are twice as expensive as comparisons in quicksort because any comparison of keys during replacement selection must be preceded by an analysis of the tags that indicate to which run the keys are assigned. Careful design, however, can belie both of these beliefs.

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 $2i + 1$, or a minor variation of this scheme. Many implementations use a

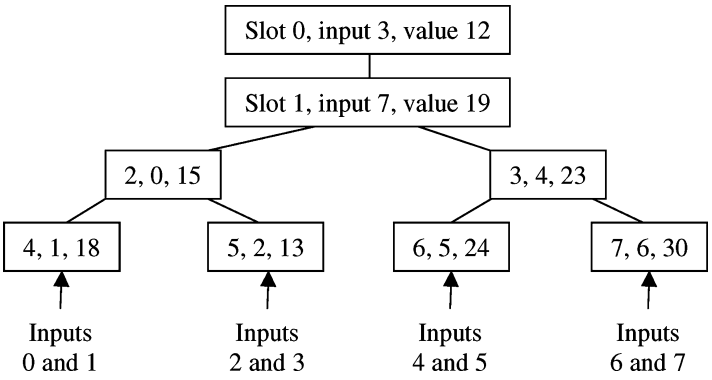

Fig. 8. A tree of losers.

tree of winners [Knuth 1998], 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, for example, records in the workspace during run generation or input runs during a merge step. In a tree of losers [Knuth 1998], no two nodes contain the same entry. There is a special root that has only one child, whereas all internal nodes have two children. Each leaf represents two runs, if necessary, by adding a dummy run. The invariants are that any leaf contains the larger key of the two runs represented by the leaf, that any internal node contains the larger among the smallest keys from each of its two subtrees, and that the tree's single-child root contains the smallest key in the entire tree. Note that the second invariant refers to one key from each subtree. Thus, an internal node does not necessarily contain the second-smallest key from the sub-tree rooted at the node.

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 to a leaf requires two comparisons per tree-level because an entry must exchange places with the smaller of its two children. The first comparison determines which of the two children is smaller, and the second compares that child with the parent. 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, with only one comparison per level.

Figure 8 illustrates a tree of losers. The slot indicates the index when the tree is embedded in an array. Slot values count level-by-level and left-to-right. The input indicates the record identifier within a workspace or the input number in a merge-sort. The values are example sort keys. The leftmost leaf is the entry point for inputs 0 and 1; the rightmost leaf is the entry point for inputs 6 and 7. As the key of input 3 is the smallest key in the tree, it is at the root. The other input with which input 3 shares the entry point (input 2) was the loser in its comparison at the leaf node, and remained there as the loser at this location. Since the root node originated in the left half of the tree, the topmost binary node must have originated from the right half of the tree, in this case from input 7. Input 6, therefore, remained as the loser at that leaf node. Note that the key value in the root of a subtree is not required to be smaller than all other values in that subtree. For example, value 19 in slot 1 is larger than value 13 in slot 5.

Either kind of priority heap is a variant of a binary tree. When the nodes of a binary tree are fitted into larger physical units, for example, disk pages or cache lines, entire units are moved within the memory hierarchy, but only a fraction of every unit is truly exploited in every access. For disk-based search trees, B-trees were invented. B-trees with nodes that are equal to cache lines have shown promise in some experiments.

Priority heaps can similarly be adapted to employ nodes the size of cache lines [Nyberg et al. 1995], with some additional space in each node to point to the node's parent or with additional complexity to compute the location of a node's child or parent. However, it is not clear whether it is more effective to generate runs using such modified priority heaps or to limit the size of the entire priority heap to that of the cache, thus creating cache-sized runs in memory and later merging such cache-sized runs into a single memory-sized run while writing to disk.

In order to make comparisons in the priority heap quickly, heap entries can employ poor man's normalized keys. In fact, these keys can be more than simply the first few bytes of the normalized key, with the result that poor man's normalized keys eliminate all comparison logic, except when two valid records must be compared on their entire keys.

An example will shortly clarify the following ideas. First, priority heaps may contain invalid entries, indicating, for example, that during a merge step an input run has been exhausted. This is also how a dummy run, if necessary, is represented. In order to save the analysis, regardless of whether both entries in a comparison represent valid records, invalid heap entries can have special values as their poor man's normalized keys, called *sentinel* values hereafter. It is useful to have both early and late sentinel values for invalid entries, that is, values that compare either lower or higher than all poor man's normalized keys for valid entries. Second, in order to simplify the logic after two poor man's normalized keys are found to be equal, two sentinel values in the priority heap should never compare as equal. To safeguard against this, each possible heap entry (each record slot in the workspace during run generation or each input run during a merge step) must have its own early and late sentinel values. Third, during run generation, when the heap may simultaneously contain records designated for the current output run as well as those for the next output run, the poor man's normalized key can also encode the run number of valid entries those 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.

For example, assume the priority heap's data structure supports normalized keys of 16 bits or 2^{16} possible (nonnegative) values, including sentinel values. Let the heap size be 2^{10} entries, that is, let the priority heap support sorting up to 1,024 records in the workspace or merging up to 1,024 runs. The lowest 2^{10} possible values and highest 2^{10} possible 16-bit values are reserved as sentinels, a low and high sentinel 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, we might use only 2^{15} values (effectively, 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, we might want to use only 12 of these 15 bits, leaving 3 bits to represent run numbers.

Thus, when the normalized key for an input record contains the value 47 in its leading 12 bits and the record is assigned to run 5, its poor man's normalized key in the priority heap is $2^{10} + 5 \times 2^{12} + 47$. The first term skips over low sentinel values, second captures the run number, which is suitably shifted such that it is more important than the record's actual key value, and the third term represents the record's actual sort key. Note that for every 7 runs ($2^3 - 1$, due to using 3 bits for the run number), a quick pass over the entire heap is required to reduce all such run numbers by 7. In other words, after 7 runs have been written to disk, all valid records remaining in the heap belong to run 7, and therefore, their normalized keys are at least $2^{10} + 7 \times 2^{12}$ and less than $2^{10} + 8 \times 2^{12}$. This pass inspects all 2^{10} entries in the priority heap and reduces each normalized key that is not a sentinel value by 7×2^{12}.

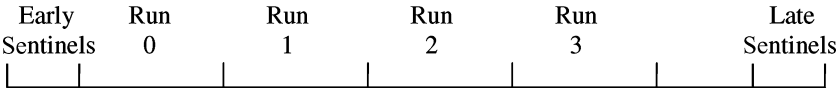

Fig. 9. Ranges of keys in a priority queue during run generation.

Figure 9 illustrates these ranges of normalized keys. The lowest and highest values are sentinel values, one per entry in the priority queue. Between them, there are several runs. Each run has a dedicated range of key values. The more runs are created in the priority queue without resetting the key values, the fewer distinct values can be used per run, that is, more comparisons need to go beyond the poor man's normalized key values and access the data records.

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 a pass over the heap after every run (every $2^1 - 1$ runs). The traditional design uses priority queues for run generation and also employs a single bit to separate the current output run from its successor, without sweeping over after each run all the items currently in memory, but with substantially more complex logic in each comparison because this one bit is not order-preserving and thus cannot be part of a poor man's normalized key.

Prior to forming the special poor man's normalized key for use in the priority heap, a prefix of the key can be used to speed several decisions for which slightly conservative approximations suffice. For example, during run generation, the poor man's normalized key alone might determine whether an input record is assigned to the current run or the next run. Note that an input record must be assigned to the next initial run, not only if its poor man's normalized key is less than that of the record most recently written to the current run, but also if it is equal to the prior poor man's normalized key—a tradeoff between quick decisions and small losses in decision accuracy and run length. Similarly, when replacing a record in the priority heap with its successor, we might want to repair the heap either by a root-to-leaf or leaf-to-root pass, depending on the incoming key, the key it replaces, and the key in the appropriate leaf of the priority heap.

Typically, the size and depth of the priority heap are chosen to be as small as possible. However, while merging runs, particularly runs of very different sizes, it might be useful to use a larger priority heap and to reserve multiple entry points in it for each run, although only one of these points will actually be used. The objective is 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 that are decided entirely based on sentinel values. In particular, the many records from a large run participate in fewer comparisons per record. For example, when merging one run of 100,000 records and 127 runs of 1,000 records each, the typical heap with 128 entries requires 7 comparisons for each of the 227,000 records, or a total of 1,589,000 comparisons. A heap with 256 entries permits records from the large run to participate in only one comparison, while records from the remaining runs must participate in 8 comparisons each, resulting in $100{,}000 \times 1 + 127{,}000 \times 8 = 1{,}116{,}000$ comparisons, a savings of about 30%.

Figure 10 illustrates this point with a traditional merge tree on the left and an optimized merge tree on the right. In the optimized merge tree, the numerous records in the largest input participate in the least number of comparisons, whereas records from the smallest inputs participate in more. A promising algorithm for planning this

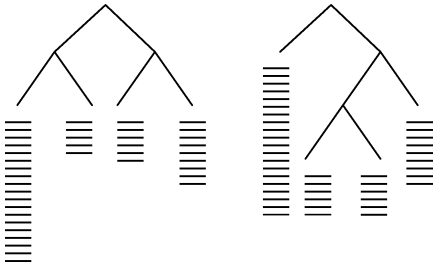

Fig. 10. Merge tree with unbalanced input sizes.

optimization is similar to the standard one for constructing a Huffman code. In both cases, the maximal depth of the tree might be higher than the depth of a balanced tree with the same number of leaves, but the total number of comparisons (in a merge tree) or of bits (in the Huffman-compressed text) is minimized.

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 does not need to participate in the priority heap at all. Instead, each key resulting from merging all other runs can be located among the remaining records of the large run, for example, using a binary search. The effect is that many records from the large run do not participate in any comparisons at all. For example, assume one run of 1,000 records has been created with about $N \log_2 N$ or 10,000 comparisons, and another run of 1,000,000 records with about 20,000,000 comparisons. A traditional merge operation of these two runs would require about 1,001,000 additional comparisons. However, theoretically, a run of 1,001,000 records could be created using only about 20,020,000 comparisons, that is, the merge step should require only $20{,}020{,}000 - 20{,}000{,}000 - 10{,}000 = 10{,}000$ comparisons. This is much less than a traditional merge step would cost, leading us to look for a better way to combine these two input runs of very different sizes. 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 a straightforward binary search and probably much less using an interpolation search—close to the number suggested by applying the $N \log_2 N$ formula to the three run sizes.

2.6. Summary of In-Memory Sorting

In summary, comparison-based sorting has been preferred over distribution sorts in database systems, and this is likely to remain the case despite some advantageous characteristics of distribution sorts. For comparison-based sorting, there are numerous techniques that speed, up each comparison, such as preparing records for fast comparison using normalization, using poor man's normalized keys, and exploiting CPU caches with carefully designed in-memory data structures.

3. EXTERNAL SORT: REDUCING AND MANAGING I/O

If the sort input is too large for an internal sort, then external sorting is needed. Presuming the reader knows about basic external merge-sort, this section discusses a number of techniques to improve the performance of external sorts. While we might believe that all we need do is improve I/O performance during merging, for example, by effective asynchronous read-ahead and write-behind, there is much more to fast external sorting. For example, since striping runs over many disk drives often improves I/O bandwidth beyond that of the CPU processing bandwidth, a well-implemented external

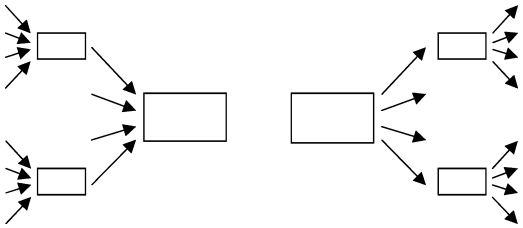

Fig. 11. Merging and partitioning.

sort also employs a variety of techniques to reduce CPU effort, both in terms of CPU instructions and cache faults.

After a brief review of external distribution-sort versus merge-sort, the discussion covers the generation of initial on-disk run files and graceful degradation in the case of inputs that are slightly larger than the available memory. Merge optimizations include merge patterns for multiple merge steps and I/O optimizations.

3.1. Partitioning Versus Merging

Just as internal sorting can be based either on distribution or merging, the same is true for external sorting. An external sort that is based on partitioning is actually quite similar to a hash-join (more accurately, it is similar to a hash aggregation or hash-based duplicate removal, as there is only one input). The main differences are that the distribution function must be order-preserving and that output from the final in-memory partitions must be sorted before being produced. As in hash-based algorithms, partitioning stops when each remaining partition fits into the available memory.

Figure 11 illustrates the duality of external merge-sort (on the left) and partitioning (on the right). Merge fan-in and partitioning fan-out, memory usage and I/O size, merge-levels and recursion depth all have duals. Not surprisingly, the same set of variants and improvements that apply to hash operations also apply to external sorting based on partitioning, with essentially the same effect. A typical example is hybrid hashing for graceful degradation when the memory is almost, but not quite, large enough. Other examples include dynamic destaging to deal with unpredictable input sizes, bucket tuning to deal with input skew, etc. [Kitsuregawa et al. 1989], although spilling or overflow ought to be governed by the key order in a distribution sort, not by partition size, as in a hash-based operation. Surprisingly, one of the complex real-world difficulties of hash operations, namely, "bail-out" execution strategies [Graefe et al. 1998] where partitioning does not produce buckets smaller than the allocated memory due to a single extremely frequent value, can neatly be integrated into sort operations based on partitioning. If a partition cannot be split further, this partition must contain only one key, and all records can be produced immediately as sorted output.

Interestingly, if two inputs need to be sorted for a merge-join, they could be sorted in a single interleaved operation, especially if this sort operation is based on partitioning. The techniques used in hash-join could then readily be adapted, for example bit vector filtering on each partitioning level. Even hash teams [Graefe et al. 1998] for more than two inputs could be adapted to implement sorting and merge-based set operations. Nonetheless, partition-based sorting hasn't been extensively explored for external sorts in database systems, partially because this adaptation hasn't been used before. However, in light of all the techniques that have been developed for hash operations, it might well be worth a new, thorough analysis and experimental evaluation.

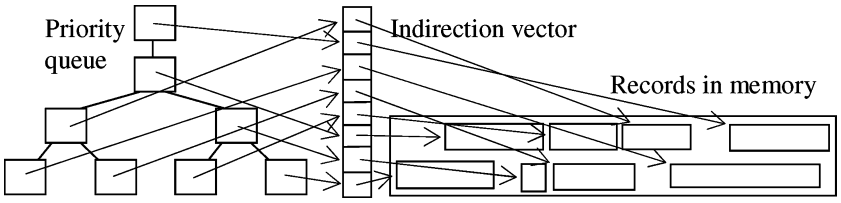

Fig. 12. Memory organization during run generation.

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, or even less if multiple runs are read, sorted, and written concurrently in order to overlap CPU and I/O activity [Nyberg et al. 1995]. Alternatively, replacement selection based on the priority heap produces runs about twice as large as the memory allocation [Knuth 1998]. Depending on the nature and extent of the disorder in the input, runs are at least as large as the in-memory workspace, but can be much longer—see [Estivil-Castro and Wood, 1992] for a discussion of the many alternative metrics of disorder and sorting algorithms that adapt to and exploit nonrandom input sequences. In general, replacement selection can defer an item from its position in the input stream for a very long interval, but can move it forward only by the size of the workspace. For example, if the in-memory workspace holds 1,000 records, the 7,815th input item can be arbitrarily late in the output sequence, but cannot be produced in the output sequence earlier than the 6,816th position.

Memory management techniques and their costs differ considerably among internal sorting methods. 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, and memory management for quicksort and internal merge-sort is straightforward, even for inputs with variable-length records. On the other hand, because replacement selection replaces records in the in-memory workspace one at a time, it requires free space management for variable-length records, as well as at least two copy operations per record (to and from the workspace).

Figure 12 illustrates memory organization during run generation. A priority queue, implemented using a tree of losers, refers to an indirection vector that is reminiscent of the one used to manage variable-length records in database pages. The slots in the indirection vector point to the actual records on the sort operation's workspace. The indirection vector also determines which identifiers the records can be referred to in the priority queue and the records' entry points into the tree of losers. Omitted from Figure 12 are poor man's normalized keys and free space management. The former would be integrated into the indirection vector. The latter interprets the gaps between valid records as records in their own right and merges neighboring "gap records" whenever possible. In other words, free space management maintains a tree of gap records ordered by their addresses as well as by some free lists for gap sizes.

Fortunately, fairly simple techniques for free space management seem to permit memory utilization around 90%, without any additional copy steps due to memory management [Larson and Graefe 1998]. However, such memory management schemes imply that the number of both records in the workspace and valid entries in the priority heap may fluctuate. Thus, the heap implementation is required to efficiently accommodate a temporary absence of entries. If growing and shrinking the number of records in the priority heap is expensive, forming miniruns (e.g., all the records from one input page) prior to replacement selection eliminates most of the cost [Larson 2003]. Note that this

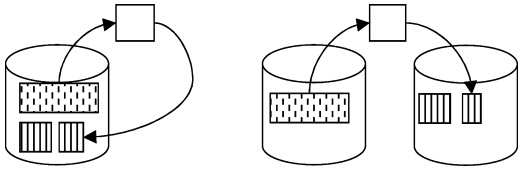

Fig. 13. Run generations with a single disk or dual disks.

method is not a simple merge of internal (page-or cache-sized) runs. Instead, by continuously replacing miniruns in the priority heap like replacement selection continuously replaces records, this method achieves runs substantially longer than the allocated memory.

Internal merge-sort can exploit sort order preexisting in the input quite effectively. In the simplest algorithm variant, the merge-sort is initialized by dividing the input into initial "runs" of one record. If, however, multiple successive input records happen to be in the correct order, they can form a larger initial run, thus saving merge effort. For random input, these runs will contain two data elements, on average. For presorted input, these runs can be considerably longer. If initial runs can be extended at both ends, initial runs will also be long for input presorted in reverse order. Quicksort, on the other hand, typically will not benefit much from incidental ordering in the input.

One of the desirable side effects of replacement selection is that the entire run generation process is continuous, alternately consuming input pages and producing run pages, rather than cycling through distinct read, sort, and write phases. In a database query processor, this steady behavior not only permits concurrently running the input query plan and the disks for temporary run files, but also has desirable effects on both parallel query plans and parallel sorting, as will be discussed later.

Figure 13 shows a single disk storing both input and intermediate runs, and alternatively, two disks serving these functions separately. Load-sort-store run generation is appropriate for the lefthand configuration, whereas continuous run generation is not. The righthand configuration always has one disk idle during load-sort-store run generation, but excels in continuous run generation due to a small number of required disk seeks.

There are several ways to accommodate or exploit CPU caches during run generation. One was mentioned earlier: creating multiple cache-sized runs in memory and merging them into initial on-disk runs. The cache-sized runs can be created using a load-sort-write (to memory) algorithm or (cache-sized) 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 fit into the cache because record accesses most likely will 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, fits in the cache—a single page or I/O unit might be a convenient size [Zhang and Larson 1997].

Another way to reduce cache faults on code and global data structures is to run various activities not for each record, but in bursts of records [Harizopoulos and Ailamaki 2003]. Such activities include obtaining (pointers to) new input records, finding space in and 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 many modules in the database server, for example, the 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 these are typically implemented such that they favor replacement of keys over separate deletion and insertion.

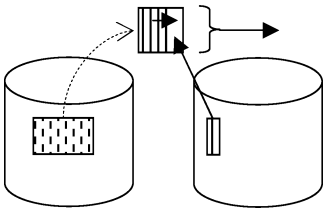

Fig. 14. Merging an in-memory run with on-disk runs.

3.3. 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 a little too large to be sorted in memory, many sort implementations spill the entire input to disk. A better policy is to spill only as much as absolutely necessary by writing data to runs only so as to make space for more input records [Graefe 1993], as previously described in the section on run generation. For example, if the input is only 10 pages larger than an available sort memory of 1,000 pages, only about 10 pages need to be spilled to disk. The total I/O to and from temporary files for the entire sort should be about 20 pages, as opposed to 2,020 pages in some 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, hash, and bitmap operations. If the sort operation's cost function has a stark discontinuity, a surprising amount of special-case code in the memory management policy must be designed to reliably avoid fairly small (but relatively expensive) sorts. This problem is exacerbated by inevitable inaccuracies in cardinality estimation during compile-time query optimization. If, on the other hand, the cost function is smooth because both CPU and I/O loads grow continuously, implementing an effective memory allocation policy is much more straightforward. Note that special cases require not only development time, but also substantial testing efforts, as well as explanations for database users who observe such surprisingly expensive sort operations and queries.

In database query processing, it often is not known before execution precisely how many pages, runs, etc., will be needed in a sort operation. In order to achieve graceful degradation, the last two runs generated must be special. In general, runs should be written to disk only as necessary to create space for additional input records. The last run remains in memory and is never written to a run file, and the prior run 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 the writing of run files can stop as soon as reading input reaches the end of the input.

Figure 14 illustrates the merge step required for an input that is only slightly larger than memory. The disk on the left provides the input, whereas the disk on the right holds any intermediate run files. Only a fraction of the input has been written to an initial on-disk run, much of the input remaining in memory as a second run, and the binary merge operation produces the sort operation's final output from these two runs.

3.4. Merge Patterns and Merge Optimizations

It is widely believed 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

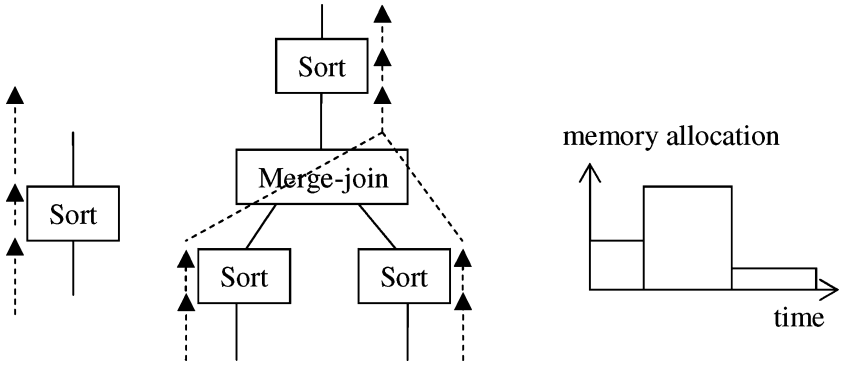

Fig. 15. Operator phases, plan phases, and a memory allocation profile.

academic exercise. If a sort operation is used to create an index, the belief might be justified, except for very large tables in a data warehouse. However, if the sort operation is part of a complex query plan that pipes data among multiple sort or hash operations, all of them competing for memory, and in particular, if nested queries employ sorting, for example, for grouping or duplicate removal, multilevel merging is not uncommon.

Figure 15 shows a sort operation taken from a larger query execution plan. The sort algorithm's operator phases are indicated by arrows, namely, the input phase with run generation, the merge phase with all intermediate merge steps, and the output phase with the final merge step. It also shows a larger plan segment with a merge-join fed by two sort operations and feeding another sort operation on a different sort key. The three sort operations' operator phases define seven plan phases. For example, the intermediate merge phase of the lefthand sort operation itself defines an entire plan phase, and may therefore use all memory allocated to the query execution. Concurrent with the merge-join, however, is the output phase of two sort operations' plus another sort operation's input phase, all of which compete for memory. Thus, we may expect each of the lower sort operations to have a memory allocation profile similar to the one shown at the righthandside, namely, a moderate memory allocation during the input phase (depending on the source of the input), full allocation during intermediate merge steps, and a relatively small memory allocation during the output phase.

Multilevel merging is particularly common if a query processor employs eager or semieager merging [Graefe 1993], which interleaves merge steps with run generation. The problem with eager merging is that the operations producing the sort input may compete that have the sort for memory, thus forcing merge operations that have less than all available query memory.

One reason for using eager and semieager merging is to limit the number of runs existing at one time, for example, because this permits managing all existing runs with a fixed amount of memory, such as one or two pages. Probably a better solution is to use 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 sufficiently reduced such that the remaining descriptors fit on a single page.

The goal of merge optimizations is to reduce the number of runs to one, yet to perform as few merge steps and move as few records as possible while doing so. Thus, an effective heuristic is to always merge the smallest existing runs. All merge steps except the first ought to use maximal fan-in. However, if not all runs are considered during merge planning (e.g., because some merge steps precede the end of the input

or the directory of run descriptors exceeds the memory dedicated to merge planning), alternative heuristics may be better, for example, when merging runs most similar in size, independent of their absolute size. This latter heuristic attempts to ensure that any merge output run of size N requires no more sorting and merge effort than $N \log N$ comparisons.

Merge planning should also attempt to avoid merge steps altogether. If the records in two or more runs have nonoverlapping key ranges, these runs can be combined into a single run [Härder 1977]. Rather than concatenating files by moving pages on-disk, it is sufficient to simply declare all these files as a single "virtual" run and to scan all files that make up a virtual run when actually merging runs. Planning such *virtual concatenation* can be implemented relatively easily by retaining low and high keys in each run descriptor and using a priority heap that sorts all available low and high keys, that is, twice as many keys as there are runs. Instead of long or variable-length keys, poor man's normalized keys might suffice, with only a moderate loss of effectiveness. If choices exist on how to combine runs into virtual runs, both the combined key range and combined run size should be considered.

While virtual concatenation is not very promising for random inputs because most runs will effectively cover the entire key range, it is extremely effective for inputs that are almost sorted, which particularly includes inputs sorted on only a prefix of the desired sort key, as well as for reverse-sorted inputs. Another example application is that of a minor change to an existing sort order, for example, a conversion from case-insensitive English to case-sensitive German collation.

The idea of virtual concatenation can be taken further, although the following ideas have not been considered in prior research or practice (to the best of our knowledge). The essence is to combine merging and range partitioning, and to exploit information gathered while writing runs to optimize the merge process. Instead of merging or concatenating entire runs, fractions of runs or ranges of keys could be merged or concatenated. For example, consider a run that covers the entire range of keys and therefore cannot participate in virtual concatenation as previously described. However, assume that most of the records in this run have keys that sort lower than some given key, and that only a few keys are high. For the lower key range, this run appears to be large, whereas for the high key range, it appears to be small. Therefore, the lower key range ought to be merged with other large runs, and the higher key range with other small runs. If there is not one, but maybe a dozen such "partition" keys, are if all runs are partitioned into these ranges and the key distributions differ among runs, merging range-by-range ought to be more efficient than merging run-by-run. Starting a merge at such a partition key within a run on-disk is no problem if runs are stored on-disk in B-trees, as will be proposed next.

As a simple example, consider an external merge-sort with memory for a fan-in of 10, and 18 runs remaining to be merged with 1,000 records each. The keys are strings with characters 'a' to 'z'. Assume both these keys occur in all runs, so traditional virtual concatenation does not apply. However, assume that in 9 of these 18 runs, the key 'm' appears in the 100th record, whereas in the others, it appears in the 900th record. The final merge step in all merge strategies will process all 18,000 records, with no savings possible. The required intermediate merge step in the standard merge strategy first chooses the smallest 9 runs (or 9 random runs, since they all contain 1,000 records), and merges these at a cost of 9,000 records that are read, merged, and written. The total merge effort is $9{,}000 + 18{,}000 = 27{,}000$ records. The alternative strategy proposed here merges key ranges. In the first merge step, 9 times 100 records with the keys 'a' to 'm' are merged, followed by 9 times 100 records with keys 'm' to 'z'. These 1,800 records are written into a single output run. The final merge step merges these 1,800 records with 9 times 900 records with keys 'a' to 'm,' followed by another 9 times 900

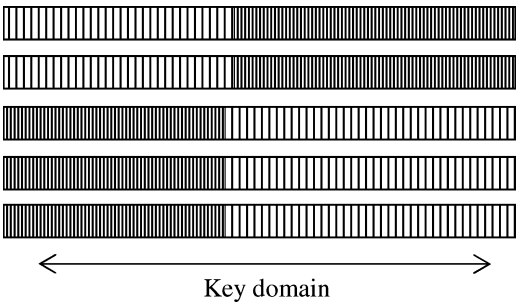

Fig. 16. Key distributions in runs.

records with keys 'm' to 'z'. Thus, the total merge effort is 1,800 + 18,000 = 19,800 records; a savings of about 25% in this (admittedly extreme) example.

Figure 16 illustrates this example. Assume here that the maximal merge fan-in is three such that two merge steps are required for five runs. In this case, the most efficient strategy merges only sparsely populated key ranges in the first merge step, leaving densely populated ranges to the second and final step. In Figure 16, the optimal merge plan consumes the first two runs plus one of the other runs for the low key range, and the last three runs for the high key range.

3.5. I/O Optimizations

Finally, there are I/O optimizations. Some of them are quite obvious, but embarrassingly, not always considered. For example, files and file systems used by database systems typically should not use the file system buffer [Stonebraker and Kumar 1986], virtual device drivers (e.g., for virus protection), compression provided by the file system, and the like. Sorting might even bypass the general-purpose database buffer. Network-attached storage (NAS) does not seem ideal for sorting. Rather, the network-attached device ought to perform low-level functions such as sorting or creating and searching B-tree indexes, possibly using normalized keys or portable code (compiled just-in-time) for comparisons, copying, scan predicates, etc. Finally, updates to run files should not be logged (except space allocations to enable cleanup, e.g., after a system failure), and using redundant devices (e.g., RAID) for run files seems rather wasteful in terms of space, processing, and bandwidth. If disk arrays are used, for example, RAID 5 [Chen et al. 1994], read operations can blithely read individual pages, but write operations should write an entire stripe at a time, with obvious, effects on memory management and merge fan-in. These recommendations may seem utterly obvious, but they are violated nonetheless in some implementations and installations.

It is well-known that sequential I/O achieves much higher disk bandwidth than random I/O. Sorting cannot work with pure sequential I/O because the point of sorting is to rearrange records in a new, sorted order. Therefore, a good compromise between bandwidth and merge fan-in is needed. Depending on the specific machine configuration and I/O hardware, 1 MB is typically a reasonable compromise on today's server machines, and even has been for a while, especially if it does not increase the number of merge steps [Salzberg 1989]. If CPU processing bandwidth is not the limiting resource, the optimal I/O unit achieves the maximal product of the bandwidth and the logarithm of the merge fan-in. This is intuitively the right tradeoff because it enables the maximal number of useful key comparisons per unit of time and the number of comparisons in an entire sort is practically constant for all (correctly implemented) merge strategies.

Table 1. Effect of Page Size on the Rate of Comparisons

Page Size	IOs/sec	Records/sec	Merge Fan-In	Heap Depth	Comparisons/sec
16 KB	111	17,760	4,093	12	213,120
64 KB	106	67,840	1,021	10	678,400
256 KB	94	240,640	253	8	1,925,120
1 MB	65	665,600	61	5.9	3,927,040
4 MB	29	1,187,840	13	3.7	4,395,008

Table I shows the effect of page size on the number of comparisons per second, including some intermediate results aiding the calculation. These calculated values are based on disk performance parameters found in contemporary SCSI disks offered by multiple manufacturers: 1 ms overhead for command execution, 5 ms average seek time, 10,000 rpm or 3 ms rotational latency, and 160 MB/sec transfer rate. The calculations assume 40 records per page of 4 KB, 64 MB of memory available to a single sort operator within a single thread of a single query, and 3 buffers required for merge output and asynchronous I/O against a single disk. While different assumptions and performance parameters change the result quantitatively, they do not change it qualitatively, as can easily be verified using a simple spreadsheet or experiment.

In Table I, larger units of I/O always result in higher I/O bandwidth, more comparisons per second, and thus faster overall sort performance. Of course, if I/O bandwidth is not the bottleneck, for example, because the CPU cannot perform as many comparisons as the disk bandwidth permits, reducing the merge fan-in is counterproductive. Up to this point, however, maximizing the merge fan-in is the wrong heuristic, whereas maximizing I/O bandwidth is more closely correlated to optimal sort performance. Perhaps a reasonable and robust heuristic is to choose the unit of I/O such that the disk access time equals the disk transfer time. In the example, the access time of 9 ms multiplied by the transfer bandwidth of 160 MB/sec suggests a unit of I/O of about 1 MB.

Of course, there are reasons to deviate from these simple heuristics, particularly if merge input runs have different sizes and 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, is proportional to the square root of the run size. If the disk layout is known, larger I/O operations can be planned by anticipating the page consumption sequence among all merge input runs, even variable-sized and batches of multiple moderate-sized I/Os [Zhang and Larson 1997, 1998; Zheng and Larson 1996]. Using key distributions saved for each run while writing it, the consumption sequence can be derived either before a merge step or dynamically as the merge progresses.

Even if each I/O operation moves a sizeable amount of data that is, contiguous on-disk, say 1 MB, it is not necessary that this data is contiguous in memory. In fact, even if it is contiguous in the virtual address space, it probably is not contiguous in physical RAM. Scatter/gather I/O (scattering read and gathering write) can be exploited in an interesting way [Zhang and Larson 1998]. Records must be stored in smaller pages, for example, 8 KB, such that each large I/O moves multiple self-contained pages. When a sufficient number of pages has been consumed by the merge logic, say 8, from any input runs, a new asynchronous read request is initiated. The important point is that individual pages may come from multiple input runs, and will be reassigned such that they all serve as input buffers for one run that is, selected by the forecasting logic. In the standard approach, each input run requires memory equal to a full I/O unit, for example, 1 MB, in addition to the memory reserved for both the output buffer and asynchronous I/O. In this modified design, on the other hand, each input run might require a full I/O unit in the worst case, but only one-half of the last large I/O remains at any point in

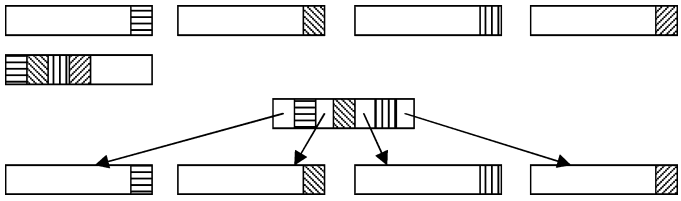

Fig. 17. Run file and boundary page versus B-tree.

time. Thus, the modified design permits an earlier and more asynchronous read-ahead or a higher fan-in, the latter with some additional logic to cope with the temporary contention among the runs.

Given that most database servers have many more disk drives than CPUs, typically by roughly an order of magnitude, either many threads or an asynchronous I/O needs to be used to achieve full system 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 most beneficial run to read from. A single asynchronous read can be forecasted correctly by comparing the highest keys in all current input buffers [Knuth 1998]. If, as is typical, multiple disk drives are to be exploited, multiple reads must be forecasted, roughly as many as there are disk access arms (or half as many if both reading and writing share the disk arms). A possible simple heuristic is to extend the standard single-page forecast to multiple pages, although the resulting forecasts may be wrong, particularly if data distributions are skewed or merge input runs differ greatly in size and therefore multiple pages from a single large run ought to be fetched. Alternatively, the sort can retain key values at all page boundaries in all runs, either in locations separate from the runs or as part of the runs themselves.

Note that such runs and the key values extracted at all page boundaries strongly resemble the leaves and their parents in a B^+-tree [Comer 1979]. Rather than designing a special storage structure and writing special code for run files, we might want to reuse the entire B-tree code for managing runs [Graefe 2003]. The additional run-time cost of doing so ought to be minimal, given that typically, 99% of a B-tree's allocated pages are leaves and 99% of the remaining pages are immediate parents, leaving only 0.01% of unused overhead. We might also want to reuse B-tree code because of its cache-optimized page structures, poor man's normalized keys in B-trees [Graefe and Larson 2001] and of course, multileaf read-ahead directed by the parent level implemented for ordinary index-order scans.

Appending new entries must be optimized as in ordinary B-tree creation. If a single B-tree is used for all runs in an external merge-sort, the run number should be the first key column in the B-tree. Comparisons in merge steps must skip this first column, runs probably should start on page boundaries (even if the prior leaf page is not full), sorted bulk-insertion operations must be optimized similarly to append operations during B-tree creation, and the deletion of an entire range of keys must be optimized, possibly recycling the freed pages for subsequent runs.

The top half of Figure 17 shows a run file with 4 pages, together with a separate data structure containing boundary keys, that is, the highest key extracted from each page in the run file. The figure does not show the auxiliary data structures needed to link these pages together, although they are of course necessary. The bottom half of Figure 17 shows an alternative representation of the same information, structured as a B-tree.

While higher RAID levels with redundancy are a bad idea for sort runs, disk striping without redundancy is a good idea for sorting. The easiest way to exploit many disks is

simply to stripe all runs similarly over all disks, in units that are either equal to or a small multiple of the basic I/O unit, that is, ½ MB to 4 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 the most effective.

3.6. Summary of External Sorting

In summary, external merge-sort is the standard external sort method in contemporary database systems. In addition to fairly obvious I/O optimizations, especially very large units of I/O, there are numerous techniques that can improve the sort performance by a substantial margin, including optimized merge patterns, virtual concatenation, and graceful degradation. In order to minimize the volume of code, both for maintenance and for efficiency in the CPU's instruction cache, internal and external sorting should exploit normalized keys and priority queues for multiple purposes.

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 additionally considers sorting in database query processors and its role in processing complex *ad hoc* queries.

In the architecture of database systems, sorting is often considered a function of the storage system, since it is used to create indices and its performance depends so much on I/O mechanisms. However, sorting can also be designed and implemented as a query operation with query processor interfaces for streaming input and output. The advantage of this design is that sort operations integrate more smoothly into complex query plans, whether these answer a user query or create an index on a view.

While it is obvious that the creation of an index on a view benefits from query planning, the same is true for traditional indices on tables. For example, if a new index requires only a few columns that are already indexed in other ways, it might be slower to scan a stored table with very large records than to scan two or three prior indices with short records and to join them on their common row identifier. Index creation also ought to compete with concurrent large queries for memory, processing bandwidth (parallelism), temporary space, etc., all of which suggests that index creation (including sorting, in particular) should be implemented as part of the query processor.

4.1. Sorting to Reduce the Data Volume

Sorting is one of the basic methods to group records by a common value. Typical examples include aggregation (with grouping) and duplicate removal, but most of the considerations here also apply to "top" operations, including grouped top operations. An example of the latter is the query to find the top salespeople in many regions—one reasonable implementation sorts them by their region and sales volume. Performing the desired operation (top) not after, but during the sort operation can substantially improve performance. The required logic can be invoked while writing run files, both initial and intermediate runs, and while producing the final output [Bitton and DeWitt 1983; Härder 1977]. The effect is that no run can be larger than the final output of the aggregation or top operation. Thus, assuming randomly distributed input keys, early aggregation is effective if the data reduction factor due to aggregation or top is larger than the merge fan-in of the final merge step [Graefe 1993]. If even the sizes of initial

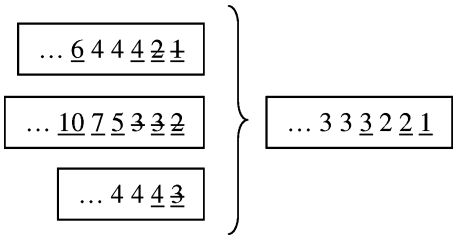

Fig. 18. Avoiding comparisons without duplicate elimination.

runs are affected in a top operation, the highest key written in prior runs can also be used to filter out incoming records immediately.

Even if a sort operation does not reduce the data volume, there is a related optimization that applies to all sort operations. After two specific records have been compared once and found to have equal keys, they can form a *value packet* [Kooi 1980]. Each value packet can move through all subsequent merge steps as a unit, and only the first record within each value packet participates in the merge logic. Thus, the merge logic of any sort should never require more comparisons than a sort with duplicate removal. If there is a chance that records in the same run file will compare as equal, value packets can be formed as the run is being written. A simple implementation is to mark each record, using a single bit, as either a head of a value packet or a subsequent member of a value packet. Only head records participate in the merge logic while merging in-memory runs into initial on-disk runs and merging multiple on-disk runs. Member records bypass the merge logic and are immediately copied from the input to the output.

Figure 18 illustrates the point in a three-way merge. The underlined keys are heads of a value packet in the merge inputs and merge output. Values 1, 2, and 3 are struck out in the merge inputs because they have already gone through the merge logic. In the inputs, both copies of value 2 are marked as heads of a value packet within their runs. In the output, only the first copy is marked, whereas the second one is not, so as to be exploited in the next merge level. For value 3, one copy in the input is already not marked and thus did not participate in the merge logic of the present merge step. In the next merge level, two copies of the value 3 will not participate in the merge logic. For value 4, the savings promise to be even greater: Only two of six copies will participate in the merge logic of the present step, and only one in six in the next merge level.

4.2. Pipelining Among Query-Evaluation Iterators

A complex query might require many operations to transform stored data into the desired query result. Many commercial database systems pass data within a single evaluation thread using some variant of iterators [Graefe 1993]. The benefit of iterators is that intermediate results are written to disk only when memory-intensive stop-and-go operations such as sort or hash-join exceed their memory allocation. Moreover, all such I/O and files are managed within a single operation, for example, run files within a sort operation or overflow files within a hash aggregation or hash-join.

Iterators can be data-driven or demand-driven, and their unit of iteration can be a single or group of records. Figure 19 illustrates the two prototypical query execution plans that benefit from demand- or data-driven dataflow. In the lefthand plan, the merge-join performs most efficiently if it controls the progress of the two sort operations, at least during their final merge steps. In the righthand plan, the spool operation

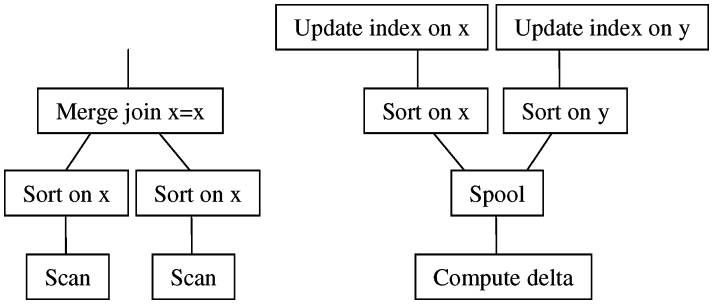

Fig. 19. Demand-driven and data-driven dataflow.

performs with the least overhead if it never needs to save intermediate result records to disk and instead can drive its two output operations, at least during their run generation phases. In general, stop-and-go operations such as sort can be implemented to be very tolerant about running in demand or data-driven dataflow.

If an iterator's unit of progress is a group of records, it can be defined its data volume or a common attribute value, the latter case called *value packets* [Kooi 1980]. Obviously, B-tree scans, sort operations, and merge-joins are good candidates for producing their output in value packets rather than in individual records. For example, a B-tree scan or sort operation passing its output into a merge-join as value packets may save a large fraction of the key comparisons in the merge-join.

Recently, an additional reason for processing multiple records at a time has emerged, namely, CPU caches [Padmanabhan et al. 2001]. For example, if a certain selection predicate requires the expression evaluator as well as certain large constants such as long strings, it might be advantageous to load the expression evaluator and these large constants into the CPU caches only once for every few records, rather than for every single record. Such batching can reduce cache faults both for instructions and for global data structures.

More generally, the multithreaded execution of a large program such as a database server can be organized around shared data structures, and activities can be scheduled for spatial and temporal locality [Larus and Parkes 2001]. As mentioned earlier, this technique can also be exploited for activities within a sort operation, for example, 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 replacing a record. In other words, strict alternation between removal and insertion leads to optimal efficiency. 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 reestablished first after each deletion and then after each insertion. Moreover, these passes might 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.

4.3. Nested Iteration

In a well-indexed database, the optimal query execution plan frequently relies entirely on index navigation rather than on set-oriented operations such as merge-join, hash-join, and their variants. In databases with concurrent queries and updates, index-to-index navigation is often preferable over set-oriented operations because the former locks individual records or keys rather than entire tables or indexes. It could even be

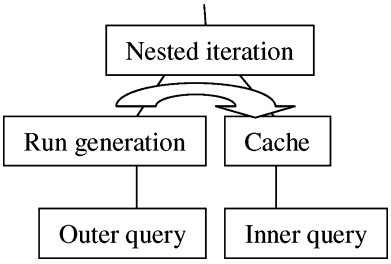

Fig. 20. Nested iteration with optimizations.

argued that index navigation is the only truly scalable query execution strategy because its run-time grows only logarithmically with data size, whereas sorting and hash-join grow, at best, linearly. Typically, such a plan is not simply a sequence of index lookups, but a careful assembly of more or less complex nested iterations, whether or not the original query formulation employed nested subqueries. Sorting, if carefully designed and implemented, can be exploited in various 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 (outer) binding values to (inner) query results [Graefe 2003b]. This cache will require less 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 binding values. An opportunistic variant of this technique does not perform a complete sort of the outer rows, but only an in-memory run generation to improve the chances of locality either in this cache or even in the indices searched by the inner query block. This variant can also be useful in object-oriented databases for object id resolution and object assembly [Keller et al. 1991].

Figure 20 shows a query execution plan with a nested iteration, including some optimizations for the nested iteration. First, the correlation or binding values from the outer input are sorted in order to improve locality in indices, caches, etc., in the inner query. However, the cost of an external sort is avoided by restricting the sort operation to its run generation logic, that is, it produces runs for the nested iteration operation in hope that successive invocations of the inner query have equal or similar binding values. Second, the cache between the nested iteration operation and the inner query execution plan may avoid execution of the inner query in the case of binding values that are equal to prior values. Depending on the cache size and organization, this might mean the single most recent or frequent bindings, or any previous binding.

Note that the cache of results from prior nested invocations might not be an explicit data structure and operation specifically inserted into the inner query plan for this purpose. Rather, it might be the memory contents built-up by a stop-and-go operator, for example, a hash-join or sort. In this case, the sort operation at the root of the inner query must not “recycle” its memory and final on-disk runs, even if a sort implementation by default releases its memory and disk space as quickly as possible. Retaining these resources enables the fast and cheap “rewind” of sort operations by restarting the final merge. While the CPU cost of merging is incurred for each scan over the sort result, the additional I/O cost for a separate sorted spool file is avoided. Unless there are very many rewind operations, merging repeatedly is less expensive than spooling a single file to disk.

Just as a query plan might pipeline intermediate results by multiple rows at a time, it can be a good idea to bind multiple outer rows to the inner query plan—this and closely related ideas have been called *semijoin reduction*, *sideways information passing* or *magic* over the years [Bernstein and Chio 1981; Seshadri et al. 1996]. Multiple executions of the inner query plan are folded into one, with the resulting rows often in a sort order less than ideal for combining them with the outer query block. In this case, the outer rows can be tagged with sequence numbers, the sequence numbers 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.

4.4. 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 between consuming its input and producing the final output. These three sort phases—run generation while consuming input, intermediate merges, and the final merge producing output—and similar operator phases in other stop-and-go operators define plan phases in complex query execution plans. For example, in an *ad hoc* query with two sort operators feeding data into a merge-join that 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.

Operators within a single query plan 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 to 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 page or other sort operations having to perform a "one-way" final merge, that is, the last "intermediate" merge step gathers all output into a single run and the final step simply scans this 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 modified heuristics, and the same is true if bitmap filtering employs large bitmaps. 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, and thus a more sophisticated memory allocation policy. Finally, some sort operations are not stop-and-go. Only the last of these issues is considered here because it is the most specific to sorting.

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 can be 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 in-memory, 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. During all intermediate merge steps, it can employ all available memory. Note that the

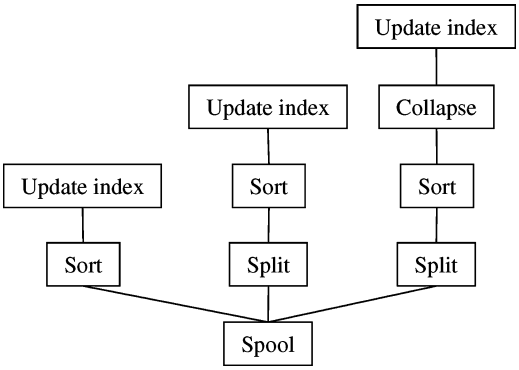

Fig. 21. Optimized index maintenance plan.

same situation that enables major-minor sort also makes virtual concatenation very effective, such that even the largest input may require no intermediate merge steps. The difference is mostly in the plan phases: A stop-and-go sort with virtual concatenation separates two plan phases, whereas major-minor sort enables fast initial query results.

There are numerous proposals for dynamic memory adjustment during sorting, for example, Pang et al. [1993], Zhang and Larson [1997], and Graefe [2003]. The proposed policies differ in adjusting memory based on only a single sort or multiple concurrent sort operations, while generating or only between runs, and during a single merge step or only between them. Proposed mechanisms include adjusting the I/O unit or merge fan-in. Adding or dropping a merge input halfway through a merge step actually seems practical if virtual concatenation is employed, that is, the merge policy can deal with partial remainders of runs. Note that it is also possible to increase the merge fan-in in an on-going merge step, especially if virtual concatenation of ranges is considered and runs are stored in B-trees and therefore permit starting a scan at a desired key.

4.5. Index Creation and Maintenance

One very important purpose of sorting in database systems is the fast creation of indices, most often some variant of B-trees, including hierarchical structures ordered by hash values. In addition, sorting can be used in a variety of ways for B-tree maintenance. Consider, for example, updating a column with a uniqueness constraint and the B-tree index used to enforce it. Assume that the update makes multiple rows "exchange" their key values—say, the original values are 1 and 2 and the update is "set value = 3 − value." Simply updating a unique index row-by-row using a delete and insert for each row will detect false (temporary) violations.

Instead, after N update actions have been split into delete and insert actions, the resulting $2N$ actions can be sorted on the column value (i.e., the index entry they affect) and then applied in such a way that there will be no false violations. As a second example, when updating many rows such that numerous index leaves will be affected multiple times, sorting the insert or delete set and applying B-tree changes in the index order can result in a substantial performance gain, just like building a B-tree index bottom-up in sort order is faster than using random top-down insertions.

Figure 21 shows a part of a query execution plan, specifically those parts relevant to nonclustered index maintenance in an update statement. Not shown below the spool

operation is the query plan that computes the delta to be applied to a table and its indices. In the left branch, no columns in the index's search key are modified. Thus, it is sufficient to optimize the order in which changes are applied to existing index entries. In the center branch, one or more columns in the search key are modified. Thus, index entries may move within the index, or alternatively, updates are split into deletion and insertion actions. In the right branch, search key columns in a unique index are updated. Thus, there can be at most one deletion and one insertion per search key in the index, and matching deletion and insertion items can be collapsed into a single update item. In spite of the differences among the indices and how they are affected by the update statement, their maintenance benefits from sorting, ideally data-driven sort operations.

For a large index, for example, in a data warehouse, index creation can take a long time, possibly several days. In data warehouses used for data mining and business intelligence, which include many existing databases larger than a terabyte, it is not unusual that half of all disk space is used for a single "fact" table, and half of this is a clustered index of that table. If the database system fails during a sort to create such an index, it might be desirable to resume the index creation work rather than to restart it from the beginning. Similarly, load spikes might require pausing and resuming a resource-intensive job such as creating an index desirable, particularly if the index creation is online, that is, concurrent transactions query and update the table, even while an index creation is ongoing or paused.

In order to support pausing and resuming index operations, we can checkpoint the scan, sort, and index creation tasks between merge steps, but it is also possible to interrupt halfway through building runs as well as halfway through individual large merge steps. The key requirement, which introduces a small overhead to a large sort, is to take checkpoints that can serve as restart points [Mohan and Narang 1992]. Representing runs as B-trees [Graefe 2003], as well as dynamic virtual concatenation, can greatly improve the efficiency of such restart operations, with minimal code specific to pausing and resuming large sorts.

Another possible issue for large indices is that there might not be enough temporary space for all the run files, even if they or the individual pages within them are "recycled" as soon as the merge process has consumed them. Some commercial database systems therefore store the runs in the disk space designated for the final index, either by default or as an option. During the final merge, pages are recycled for the index being created. If the target space is the only disk space available, there is no alternative to using it for the runs, although an obvious issue with this choice is that the target space is often on mirrored or redundant RAID disks, which does not help sort performance, as discussed earlier. Moreover, sorting in the target space might lead to a final index that is, rather fragmented because the pages are recycled from merge input to merge output effectively in random order. Thus, an index-order scan of the resulting index, for examples, a range query, 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 final index creation can attempt to allocate large contiguous disk space from there. However, unless the allocation algorithm's search for contiguous free space is very effective, most of the allocations will be of the same small size in which space is recycled in the merge. Second, space can be recycled from initial 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 this multiple is 8, disk space that does not exceed 8 times the size of memory might be held for such deferred group recycling, which is typically an acceptable overhead when creating large indices. The benefit is that a full scan of the completed index requires 8 times fewer seeks in large ordered scans.

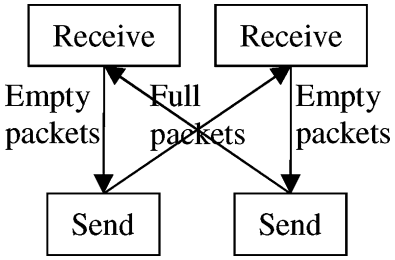

Fig. 22. Deadlock in order-preserving data exchange.

4.6. Parallelism and Threading

There is a fair amount of literature on parallel sorting, both internal and external. In database systems, parallel sorting is most heavily used in data loading and index creation [Barclay et al. 1994]. One key issue is data skew and load balancing, especially when using range partitioning [Iyer and Dias 1990; Manku et al. 1998]. The partition boundaries are typically determined prior to the sort to optimize both the final index and its expected access pattern, and range partitioning based on these boundaries followed by local sorts is typically sufficient. In query processing, however, hash partitioning is often the better choice because it works nicely with most query operations, including merge-join and sort-based duplicate removal, and user-requested sorted output can be obtained with a simple final merge operation that is, typically at least as fast as the application program consuming the output. Note that in parallel query processing based on hash partitioning, the hash value can also be used as a poor man's normalized key in sorting, merge-join, and grouping, even if it is not order-preserving.

Perhaps a more important issue for parallel sort-based query execution plans is the danger of deadlocks due to unusual data distributions and limited buffer space in the data-exchange mechanism. These cases occur rarely, but they do exist in practice and must be addressed in commercial products. Some simple forms of deadlocks are described in Graefe [1993], but more complex forms also exist, for example, 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. The easiest of these solutions, artificial keys, works only within a single data exchange step unless all sort-sensitive query operations are modified to pass through artificial records, for example, merge-join and stream aggregation.

Figure 22 illustrates the deadlock among four query execution threads participating in a single data exchange operation. Due to flow control, the two send operations wait for empty packets, that is, for permission to produce and send more data, whereas the two receive operations wait for data. A deadlock may arise if, for example, the lefthand send operation directs all its data to the lefthand receive operation, the righthand send operation directs all its data to the righthand receive operation, and neither receive operation obtains input from all its sources such that the merge logic in the order-preserving data exchange can progress, consume data, and release flow control by sending empty packets to the send operations.

Parallel query plans work best, in general, if all the data flow is steady and balanced among all parallel threads and over time. Thus, a 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 is to divide memory into three sections of equal size and create separate threads rather than distinct phases. A third alternative creates small runs in memory and merges these runs into a memory-sized disk-based run file on demand as memory is needed for new input records—an algorithm noted earlier for its efficient use of CPU caches.

If parallel sorting is employed for grouping or duplicate removal, and data needs to be repartitioned from multiple scan to multiple sort threads, and data transfer between threads is not free, the scan threads might form initial runs 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 repartitioning, thus saving transfer costs. Of course, the receiving sort threads have to merge runs and continue to remove 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. Incidentally, the "run generation" iterator is precisely what is needed to opportunistically sort outer correlation values in nested iteration (as discussed earlier), as well as to complement hash partitioning that uses an order-preserving hash function to realize a disk-based distribution sort. Similarly, the "merge" iterator is useful to produce sorted output from a partitioned B-tree index. For example, a B-tree index on columns (A, B) can be exploited to produce output sorted on B by interpreting values in the leading column, A, as partition identifiers and merging them in one or more merge steps. Whether or not this query execution plan is advantageous depends on the amount of data for each distinct value of A.

4.7. Query Planning Considerations

While all prior sorting techniques apply during the execution of a sort operation, some considerations ought to be included in the compile-time and optimization of queries and update operations. Some are briefly considered in this survey's final section.

As mentioned in the introduction, it has long been recognized that sorting can assist in grouping, duplicate removal, joins, set operations, retrieval from disk, nested iteration, etc., for example, in System R [Härder 1977]. In update plans, the Halloween problem [Gassner et al. 1993; McJones 1997] and false constraint violations may reliably be avoided by a stop-and-go operation such as a sort. Thus, sort operations and sort-based query execution plans should be considered during the compilation of these kinds of database requests.

While optimizing a query plan that includes a sort operation, there are a number of simplifications that ought to be considered. 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 levels, although mutual bit vector filtering might be much easier to implement in partition-based sorting than in merge-sort.

If an order-by list contains keys, the functional dependencies within it can be exploited [Simmen et al. 1996]. Specifically, a column can be removed from the order-by list if it is functionally dependent on columns constructed earlier within it. Incidentally, this technique also applies to partitioning in parallel query plans and column sets in hash-based algorithms. A constant column is assumed 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 or any other equality predicate, equivalence classes of columns 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 an outright removal of columns, it may pay to reorder the order-by list. For example, if the sort operation's purpose is to form groups or remove duplicates, the order-by list can be treated as a set, that is, the sequence of columns is irrelevant to the grouping operation, although it might matter if the query optimizer considers *interesting orderings* [Selinger et al. 1979] for subsequent joins or output to the user. Note that in hash-based partitioning and in grouping operations, the columns always form a set rather than a list. Thus, in cases in which both sorting and hashing are viable algorithms, the following 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 subsequently compressed, 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 or offset-value coding is used. In addition to reordering the order-by list, it is even possible to add an artificial column at the head of the list, for example, an integer computed by hashing other columns in the order-by list—of course, this idea is rather similar to both hash-based operations and poor man's normalized keys, which have been discussed earlier.

4.8. Summary of Sorting in Database Systems

In summary, there are numerous techniques that improve sort operations specifically in the context of database systems. Some of these improve performance, such as, simplifying and reordering the comparison keys, whereas others improve the dynamic and adaptive behaviors of large sort operations when complex sort operations are processed concurrently. Adaptive sorting techniques, as well as policies and mechanisms for resource management in complex query plans with nested iteration, is a challenging research area with immediate practical applications and benefits.

5. SUMMARY, CONCLUSIONS, AND OUTLOOK

In summary, it has long been known that sorting can be used in all types of database management systems for a large variety of tasks, for example, query processing, object assembly and record access, index creation and maintenance, and consistency checks. There are numerous techniques that can substantially improve the performance of sort operations. Many of these have been known for a long time, whether or not they have been widely adopted. Some techniques, however, have only been devised more recently, such as those for exploiting CPU caches at the top end of the storage hierarchy. Further improvements and adaptations of sorting algorithms, and in general, database query evaluation algorithms, might prove worthwhile with respect to further advances in

computing hardware, for example, with the imminent prevalence of multiple processing cores within each CPU chip.

In addition, storage hierarchy is becoming more varied and powerful (and thus more complex and challenging) also at the bottom end, for example, with the advent of intelligent disks and network-attached storage. Quite likely, there will be another wave of new sorting techniques (or perhaps mostly adaptations of old techniques) to exploit the processing power built into new storage devices for sorting and searching in database systems. For example, new techniques may distribute and balance the processing load between the main and storage processors and integrate activities in the latter with the data formats and transaction semantics of the database system running on the main processors. Modern portable programming languages, with their just-in-time compilers and standardized execution environments, might enable novel techniques for function shipping and load distribution in heterogeneous system architectures.

While a few of the techniques described in this survey require difficult tradeoff decisions, most are mutually complementary. In their entirety, they may speed-up sorting and sort-based query evaluation plans by a small factor or even by an order of magnitude. Perhaps more importantly, there are now many adaptive techniques to cope with or even exploit skewed key distributions, selectivity estimation errors in database query processing, and fluctuations in available memory and other resources, if necessary by pausing and efficiently resuming large sort operations. These new techniques provide a strong motivation to rethink and reimplement sorting in commercial database systems. Some product developers, however, are rather cautious about dynamic techniques because they expand the test matrix and can create challenges when reproducing customer concerns. Research into robust policies and appropriate implementation techniques could provide valuable guidance to developers of commercial data management software.

ACKNOWLEDGMENTS

A number of friends and colleagues have contributed many insightful comments to earlier drafts of this survey, including David Campbell, Bob Gerber, Wey Guy, James Hamilton, Theo Härder, Ian Jose, Per-Åke Larson, Steve Lindell, Barb Peters, and Prakash Sundaresan. Craig Freedman suggested identifying heads of value packets within runs using a single bit per record.

REFERENCES

AGARWAL, R. C 1996. A super scalar sort algorithm for RISC processors. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) Conference*. 240–246.

AHO, A., HOPCROFT, J. E., AND ULLMAN, J. D. 1983. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

ANDERSSON, A. AND NILSSON, S. 1998. Implementing radixsort. *ACM J. Experimental Algorithms 3*, 7.

ANTOSHENKOV, G., LOMET, D. B., AND MURRAY, J. 1996. Order-Preserving compression. In *Proceedings of the IEEE International Conference on Data Engineering*. 655–663.

ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. A. 1997. High-Performance sorting on networks of workstations. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) Conference*. 243–254.

BAER, J.-L. AND LIN, Y.-B. 1989. Improving quicksort performance with a codeword data structure. *IEEE Trans. Softw. Eng. 15*, 5, 622–631.

BARCLAY, T., BARNES, R., GRAY, J., AND SUNDARESAN, P. 1994. Loading databases using dataflow parallelism. *ACM SIGMOD Rec. 23*, 4, 72–83.

BERNSTEIN, P. A. AND CHIU, D.-M. W. 1981. Using semi-joins to solve relational queries. *J. ACM 28*, 1, 25–40.

BITTON, D. AND DEWITT, D. J. 1983. Duplicate record elimination in large data files. *ACM Trans. Database Syst. 8*, 2, 255–265.

BLASGEN, M. W., CASEY, R. G., AND ESWARAN, K. P. 1977. An encoding method for multifield sorting and indexing. *Comm. ACM 20*, 11, 874–878.

CHEN, P. M., LEE, E. L., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. 1994. RAID: High-performance, reliable secondary storage. *ACM Comp. Surv. 26*, 2, 145–185.

COMER, D. 1979. The ubiquitous B-Tree. *ACM Comp. Surv. 11*, 2, 121–137.

CONNER, W. M. 1977. Offset value coding. *IBM Technical Disclosure Bulletin 20*, 7, 2832–2837.

CORMEN, T. H., LEISERSON, C. E., RIVEST, AND R. L., STEIN, C. 2001. *Introduction to Algorithms, 2nd ed*. Cambridge, MA. MIT Press.

ESTIVILL-CASTRO, V. AND WOOD, D. 1992. A survey of adaptive sorting algorithms. *ACM Comp. Surv. 24*, 4, 441–476.

GASSNER, P., LOHMAN, G. M., SCHIEFER, K. B., AND WANG, Y. 1993. Query optimization in the IBM DB2 family. *IEEE Data Eng. Bulletin 16*, 4, 4–18.

GOLDSTEIN, J., RAMAKRISHNAN, R., AND SHAFT, U. 1998. Compressing relations and indexes. In *Proceedings of the IEEE International Conference on Data Engineering*, 370–379.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Comp. Surv. 25*, 2, 73–170.

GRAEFE, G. 2003. Sorting and indexing with partitioned B-Trees. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. Asilomar, CA.

GRAEFE, G. 2003b. Executing nested queries. In *Proceedings of the Datenbanksysteme für Business, Technologie und Web (BTW) Conference*. Leipzig, Germany, 58–77.

GRAEFE, G. AND LARSON, P.-A. 2001. B-Tree indexes and CPU caches. In *Proceedings of the IEEE International Conference On Data Engineering*. Heidelberg, Germany. 349–358.

GRAEFE, G., BUNKER, R., AND COOPER, S. 1998. Hash joins and hash teams in microsoft SQL server. In *Proceedings of the Conference on Very Large Databases (VLDB)*. 86–97.

HÄRDER, T. 1977. A Scan-driven sort facility for a relational database system. In *Proceedings of the Conference on Very Large Databases (VLDB)*. 236–244.

HARIZOPOULOS, S. AND AILAMAKI, A. 2003. A case for staged database systems. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. Asilomar, CA.

HU, T. C. AND TUCKER, A. C. 1971. Optimal computer search trees and variable-length alphabetic codes. *SIAM J. Appl. Math. 21*, 4, 514–532.

IYER, B. R. AND DIAS, D. M. 1990. System issues in parallel sorting for database systems. In *Proceedings of the IEEE International Conference on Data Engineering*. 246–255.

KELLER, T., GRAEFE, G., AND MAIER, D. 1991. Efficient assembly of complex objects. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) Conference*. 148–157.

KITSUREGAWA, M., NAKAYAMA, M., AND TAKAGI, M. 1989. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proceedings of the Conference on Very Large Databases (VLDB) Conference*. 257–266.

KNUTH, D. E. 1998. *The Art of Computer Programming: Sorting and Searching*. Addison Wesley Longman.

KOOI, R. 1980. The optimization of queries in relational databases, Ph.D. thesis, Case Western Reserve University.

KWAN, S. C. AND BAER, J.-L. 1985. The I/O performance of multiway mergesort and tag sort. *IEEE Trans. Comput. 34*, 4, 383–387.

LARUS, J. R. AND PARKES, M. 2001. Using cohort scheduling to enhance server performance. Microsoft Research Tech. Rep. 39.

LARSON, P.-L. 2003. External sorting: Run formation revisited. *IEEE Trans. Knowl. Data Eng. 15*, 4, 961–972.

LARSON, P.-A. AND GRAEFE, G. 1998. Memory management during run generation in external sorting. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) Conference*. 472–483.

LELEWER, D. A. AND HIRSCHBERG, D. S. 1987. Data compression. *ACM Comp. Surv. 19*, 3, 261–296.

MANKU, G. S., RAJAGOPALAN, S., AND LINDSAY, B. G. 1998. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) Conference*. 426–435.

MCJONES, P. ED. 1997. The 1995 SQL reunion: People, projects, and politics. SRC Tech. Note 1997-018, Digital Systems Research Center. Palo Alto, CA.

MOHAN, C. AND NARANG, I. 1992. Algorithms for creating indexes for very large tables without quiescing updates. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) Conference*. 361–370.

NYBERG, C, BARCLAY, T., CVETANOVIC, Z., GRAY, J., AND LOMET, D. B. 1995. AlphaSort: A cache-sensitive parallel external sort. *VLDB J. 4*, 4, 603–627.

PADMANABHAN, S., MALKEMUS, T., AGARWAL, R. C., AND JHINGRAN, A. 2001. Block-Oriented processing of relational database operations in modern computer architectures. In *Proceedings of the IEEE International Conference on Data Engineering*. 567–574.

PANG, H., CAREY, M. J., AND LIVNY, M. 1993. Memory-adaptive external sorting. In *Proceedings of the Conference on Very Large Databases (VLDB)*. 618–629.

RAHMAN, N. AND RAMAN, R 2000. Analysing cache effects in distribution sorting. *ACM J. Experimental Algorithms 5*, 14.

RAHMAN, N. AND RAMAN, R 2001. Adapting radix sort to the memory hierarchy. *ACM J. Experimental Algorithms 6*, 7.

SALZBERG, B. 1989. Merging sorted runs using large main memory. *Acta Informatica 27*, 3, 195–215.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) Conference*. 23–34.

SESHADRI, P., HELLERSTEIN, J. M., PIRAHESH, H., LEUNG, T. Y. C., RAMAKRISHNAN, R., SRIVASTAVA, D., STUCKEY, P. J., AND SUDARSHAN, S. 1996. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) Conference*. 435–446.

SIMMEN, D. E., SHEKITA, E. J., AND MALKEMUS, T. 1996. Fundamental techniques for order optimization. In *Proceedings of the Extending Database Technology Conference*. 625–628.

STONEBRAKER, M. AND KUMAR, A. 1986. Operating system support for data management. *IEEE Database Eng. Bulletin 9*, 3, 43–50.

VITTER, J. S. 1987. Design and analysis of dynamic Huffman codes. *J. ACM 34*, 4, 825–845.

ZHANG, W. AND LARSON, P.-A. 1997. Dynamic memory adjustment for external mergesort. In *Proceedings of the Conference on Very Large Databases (VLDB)*. 376–385.

ZHANG, W. AND LARSON, P.-A. 1998. Buffering and read-ahead strategies for external mergesort. In *Proceedings of the Conference on Very Large Databases (VLDB)*. 523–533.

ZHANG, C., NAUGHTON, J. F., DEWITT, D. J., LUO, Q., LOHMAN, G. M. 2001. On supporting containment queries in relational database management systems. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) Conference*. 425–436.

ZHENG, L. AND LARSON, P.-A. 1996. Speeding Up external mergesort. *IEEE Trans. Knowl. Data Eng. 8*, 2, 322–332.

Received March 2005; revised January 2006; accepted May 2006