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