Sorting in Parallel Database Systems

David Taniar
Department of Computer Science
Royal Melbourne Institute of Technology
GPO Box 2476V, Melbourne 3001, Australia
taniar@cs.rmit.edu.au

J. Wenny Rahayu
Department of Comp. Sc. and Comp. Eng.
La Trobe University
Bundoora Vic 3823, Australia
wenny@cs.latrobe.edu.au

Abstract

Sorting in database processing is frequently required through the use of Order By and Distinct clauses in SQL. Sorting is also widely known in computer science community at large. Sorting in general covers internal and external sorting. Past published work has extensively focused on external sorting on uni-processors (serial external sorting), and internal sorting on multi-processors (parallel internal sorting). External sorting on multi-processors (parallel external sorting) has received surprisingly little attention; furthermore, the way current parallel database systems do sorting is far from optimal in many scenarios. In this paper, we present a taxonomy for parallel sorting in parallel database systems, which covers five sorting methods: namely parallel merge-all sort, parallel binary-merge sort, parallel redistribution binary-merge sort, parallel redistribution merge-all sort, and parallel partitioned sort. The first two methods are previously proposed approaches to parallel external sorting which have been adopted as status quo of parallel database sorting, whereas the latter three methods which are based on redistribution and repartitioning are new that have not seen discussed in the literature of parallel external sorting.

1 Introduction

Sorting is one of the most common operations in database processing [8]. Sorting may be requested explicitly by users through the use of `Order By` clause in SQL. The `Order By` clause basically requires the query results to be ordered on the designated attributes in ascending or descending order. Another operation which also requires sorting is duplicate removal through the use of `Distinct` keyword in SQL. The `Distinct` operation basically removes all duplicates found in the query result. This can be achieved by first sorting the query results and then followed by removing duplicates through scanning [8].

Sorting may also be required in join operations through the use of sort-merge join algorithm [16]. This is less explicit than the use of `Order By` and `Distinct` clauses in SQL. However, some query optimizers allow users to specify any join algorithms to be used when invoking an SQL query [15].

The topic of sorting in traditional data structure and algorithm subjects is divided into two areas, namely *internal* and *external sorting* [12]. *Internal sorting* is where sorting takes place totally in main memory. The data to be sorted is assumed to be small and fits the main memory. Internal sorting has been a foundation of computer science. A number of internal sorting both serial and parallel has been explored [3]. *External sorting* on the other hand is where the data to be sorted is large and resides in secondary memory. Thus, external sorting is also known as file sorting. In databases, since data is stored in tables (or files) and is normally very large, database sorting is therefore an external sorting. External sorting is not really a new research topic. It has been explained in computer science textbooks [6]. However, external sorting has always been discussed in a uni-processor environment through the use of multiple disks or tapes [7]. Parallel external sorting has not been fully explored. The traditional approaches of parallel external sorting have been to perform local sort in each processor in the first stage and to carry out merging by the host or using a pipeline hierarchy of processors in the second stage [2,3].

It is the aim of this paper to fully explore parallel external sorting for high performance parallel database systems. We assume that the parallel database architecture used is a shared-nothing architecture, where each processor has its own processor and memory (main and secondary) [1]. In the taxonomy, besides the two traditional approaches, we add with three new approaches based on the redistribution/repartitioning methods. In these approaches, parallelism is better achieved because bottleneck problem and inefficient merging are solved through redistribution and repartitioning.

Before we present the taxonomy, it is necessary for the reader to be familiar with the concept of serial

0-7695-0589-2/00 $10.00 © 2000 IEEE

external sorting, since this is the foundation of parallel external sorting as local sort in each processor is actually a serial external sorting. A brief background on serial external sorting will be discussed next.

2 Serial External Sorting: A Background

Serial external sorting is external sorting in a uni-processor environment. The most common serial external sorting algorithm is based on sort-merge. The underlying principle of sort-merge algorithm is to break the file up into unsorted subfiles, sort the subfiles, and then merge the sorted subfiles into larger and larger sorted subfiles until the entire file is sorted. Notice that the first stage is to sort the first lot of subfiles, whereas the second stage is actually the merging phase. In this scenario, it is important to determine the size of the first lot of subfiles which are be sorted. Normally, each of these subfiles must be small enough to fit into main memory, so that sorting these subfiles can be done in main memory using any internal sorting technique.

We divide a serial external sorting algorithm into two phases: *sort* and *merge*.

The sort algorithm, which incorporates a partitioning of the original file, is explained as follows: First, determine R_i the number of records, which we can reasonably sort internally. Second, determine K the total number of disks we can use. Third, sort R_i records at a time internally, writing the results in turn onto each of $K/2$ disks with file markers at their ends. Finally, repeat the third step above, writing additional files onto the $K/2$ disks.

Once sorting of subfiles is completed, merging phase starts. An algorithm for the merging process is described as follows. First, do a $K/2$-way merge using the first subfile from each disk, writing the output onto one of the $K/2$ empty disks. Second, repeat the first step above for each of the rest of the $K/2$ empty disks. Finally, repeat steps 1 and 2 above, merging in rotation onto the $K/2$ subfiles until the original $K/2$ disks are empty.

As stated in the beginning that serial external sort is the basis for parallel external sort, because in a multi-processor system, particularly in a shared-nothing environment, each processor has its own data, and sorting this data locally in each processor is done as per serial external sort explained above. Therefore, the main concern in parallel external sort is not the local sort, but whether local sort is done first or later, and how merging is performed. The next section describes different ways of parallel external sort by basically considering the two factors mentioned above.

3 A Taxonomy of Parallel External Sort

We present five parallel external sort for parallel database systems. They are *parallel merge-all sort, parallel binary-merge sort, parallel redistribution binary-merge sort, parallel redistribution merge-all sort,* and *parallel partitioned sort.* Each of them will be described in more details in the following.

3.1 Parallel Merge-All Sort

Parallel Merge-All Sort method is a traditional approach which has been adopted as the basis for implementing sorting operations in several research prototype (Gamma) [4] and some commercial Parallel DBMS. Parallel Merge-All Sort is composed of two phases: *local sort* and *final merge*. The *local sort phase* is carried out independently in each processor. Local sorting in each processor is performed as per normal serial external sorting mechanism. We emphasize that it is a serial external sorting as it is assumed that the data to be sorted in each processor is very large and they cannot be fitted into the main memory, and hence external sorting (as opposed to internal sorting) is required in each processor.

After local sort phase is completed, the second phase: *final merge phase* starts. In this final merge phase, the results from the local sort phase are transferred to the host for final merging. The final merge phase is carried out by one processor namely the host. This merging operation is influenced by the practice of k-way merging (i.e. $k \geq 2$) in serial external sorting [7].

Figure 1 gives an illustration of parallel merge-all sort process. For simplicity, we use a list of numbers which are to be sorted. In the real world, the list of numbers is actually a list of records from very large tables.

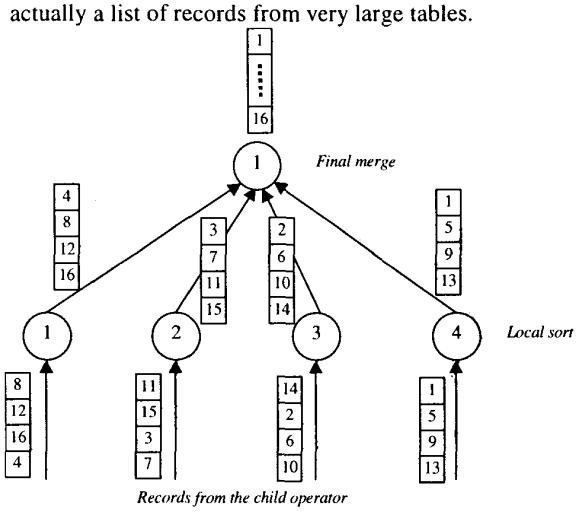

Figure 1. Parallel Merge-All Sort

Looking at the graphical illustration in Figure 1, Parallel Merge-All Sort is simple, as it is a one-level tree. Load balancing in each processor at the local sort phase is relatively easy to achieve, especially if a round-robin data placement technique is used in the initial data partitioning

[5]. It is also easy to predict the outcome of the process, as performance modelling of such process is relatively straightforward.

Despite its simplicity, Parallel Merge-All Sort method incurs an obvious problem, particularly in the final merging phase, as merging in one processor is heavy. This is true especially if number of processors is large, and there is a limit of number of files to be merged (i.e. limitation in number of files to be opened or number of disks available in the host). If the number of processors is greater than the number of open files permitted, final merging in the host must perform multiple level merging. For example, if there are 8 processors and only 5 files may be open at once, final merging in the host can merge the first 5 lists from 5 processors and produces one sorted list. The same method is applied to the other three lists producing another list. And finally, the two temporary lists may be merged to produce the final results. Detailed discussions on this issue are commonly found in literatures on serial external sorting.

Another problem with Parallel Merge-All Sort is network contention, as all temporary results from each processor in the local sort phase are passed to the host. The problem of merging by one host is to be tackled by the next sorting scheme whereby merging is not done by one processor but shared by multiple processors in a form of hierarchical merging.

3.2 Parallel Binary-Merge Sort

Parallel Binary-Merge Sort is first proposed by Bitton et al [2,3]. The first phase of *Parallel Binary-Merge Sort* is a *local sort* as like in parallel merge-all sort. The second phase: the *merging phase* is pipelined, instead of concentrating on one processor. The way the merging phase works is by taking the results from two processors, and merging the two in one processors. As this merging technique uses only two processors, this merging is called "Binary Merging". The result of the merging between two processors is passed on to the next level until one processor left; that is the host. Subsequently, the merging process forms a hierarchy. Figure 2 gives an illustration of the process.

The main motivation to use parallel binary-merge sort is that the merging workload is spread to a pipeline of processors, instead of one processor. It is true however that final merging has still to be done by one processor.

Some of the benefits of parallel binary-merge sort are similar to those of parallel merge-all sort, such as balancing in local sort can be done if a round-robin data placement is initially used to the raw data to be sorted. Another benefit as stated before that merging workload is now shared among processors.

However, problems relating to the heavy merging workload in the host still exists, even though it is now the final merging only merges a pair of list of sorted data, not a k-way merging like that in parallel merge-all sort. Binary merging can still be time consuming, particularly if the two lists to be merged are very large.

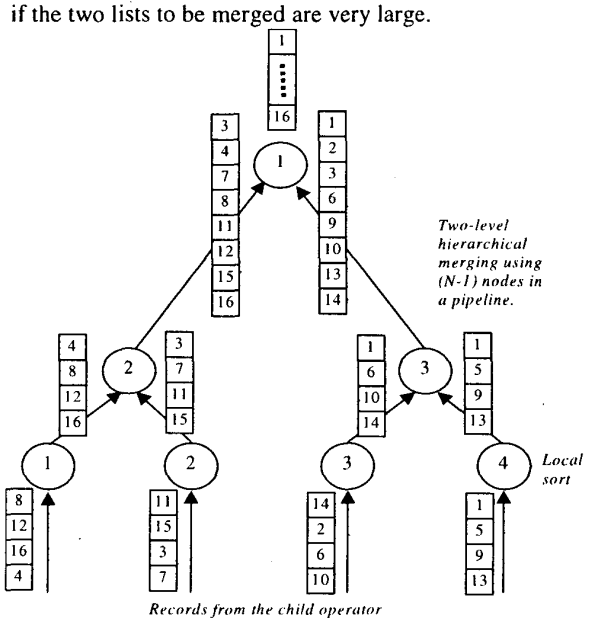

Figure 2. Parallel Binary-Merge Sort

The main difference between k-way merging and binary merging is that in k-way merging there is a searching process in the merging; that is to search the smallest value among all values compared at the same time. In binary merging, this searching is purely a comparison between two values compared at one time.

Regarding the system requirement, k-way merging requires sufficient number of files to be opened at the same time. This requirement is trivial in binary merging, as it only requires a maximum of two files to be opened, and this is easily satisfied by any operating systems.

Pipeline system as in the binary merging will certainly produce extra work through the pipe itself. The pipeline mechanism also produces a higher tree, not a one-level tree as in the previous method. However, if the limitation of the number of opened files permitted in the k-way merging, parallel merge-all sort will incur merging overheads.

In parallel binary-merge sort, it is still no true parallelism in the merging as only a subset, not all, of available processors is used.

We propose three possible alternatives using the concept of redistribution or repartitioning. The first approach we would like to introduce is a modification of parallel binary-merge sort by incorporating redistribution in the pipeline hierarchy of merging. The second approach is an alteration to parallel merge-all sort, also through the use of redistribution. The third approach we would like to include in this paper differs from the others, as local sorting is delayed after partitioning is done. The details of these three methods are described next.

3.3 Parallel Redistribution Binary-Merge Sort

Parallel Redistribution Binary-Merge Sort is motivated by parallelism at all levels in the pipeline hierarchy. Therefore, it is similar to parallel binary-merge sort where both methods use hierarchy pipeline for merging local sort results, but differ in the context of number of processors involved in the pipe. Using parallel redistribution binary-merge sort, all processors are used in each level in the hierarchy of merging.

The steps for parallel redistribution binary-merge sort can be described as follows. First, a local sort is carried out in each processor as like in the previous sorting methods. Second, redistribute the results of local sort to the same pool of processors. Third, do a merging using the same pool of processors. Finally, repeat the above two steps until final merging. The final result is the union of all temporary results obtained in each processor. Figure 3 gives an illustration of parallel redistribution binary-merge sort method.

Notice from the illustration that in the final merge phase, some of the boxes are empty (e.g. gray boxes). They indicate that they do not receive any values from the designated processors. For example, the first gray box on the left is because there is no values ranging from 1-5 from processor 2. Practically, in this example, processor 1 performs final merging of two lists, because the other two lists are empty.

Also notice that the results produced by the intermediate merging in the above example are sorted within and among processors. It means that for example, processors 1 and 2 produce a sorted list each, and the union of these results is also sorted where the results from processor 2 are preceded by those from processor 1. This is applied to other pairs of processors. Each pair of processors in this case forms a pool of processor. In the next level of merging, two pools of processors use the same strategy as in the previous level. Finally, in the final merging, all processors will form one pool, and therefore, results produced in each processor are sorted, and these results union altogether are sorted based on the processor order. In some systems, this is already a final result. If there is a need to place the results in one processor, results transfers is then carried out.

The apparent benefit of this method is that merging becomes lighter compared to those without redistribution, because merging is now shared by multiple processors, not monopolized by just one processor. Parallelism is therefore accomplished at all levels of merging, even though performance beneficial of this mechanism is restricted (In the performance evaluation section, this matter will be further clarified and quantified).

The same problem as that in without redistribution method which relates to the height of the tree remains outstanding. This is due to the fact that merging is done in a pipeline format. Another problem raised by the redistribution is skew [13]. Although initial placement in each disk is balanced through the use of round-robin data partitioning, redistribution in the merging process may likely produce skew, as shown in Figure 3. Modelling skew is also known to be difficult, and hence simplified assumption is often used, such as using a *Zipf* distribution to model skew. Like the merge-all sort method, final merging in the redistribution method is also dependent upon the maximum number of files opened.

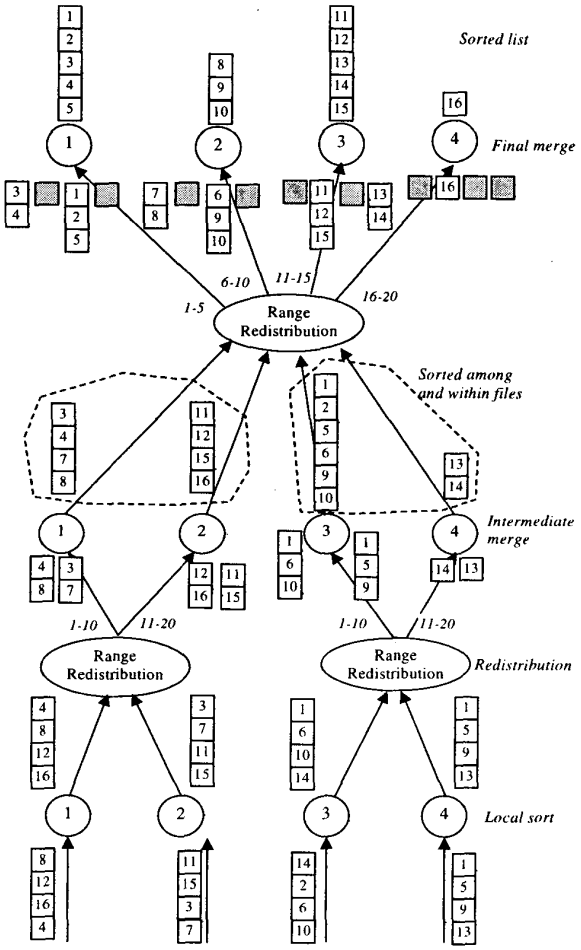

Figure 3. Parallel Redistribution Binary-Merge Sort

3.4 Parallel Redistribution Merge-All Sort

Parallel Redistribution Merge-All Sort is motivated by two factors, particularly reducing the height of the tree while maintaining parallelism at the merging stage. This can be achieved by exploiting the feature of parallel merge-all and parallel redistribution binary-merge methods. In other words, parallel redistribution is a two

phase method (local sort and final merging) like parallel merge-all sort, but does a redistribution based on a range partitioning. Figure 4 gives an illustration of parallel redistribution merge-all sort.

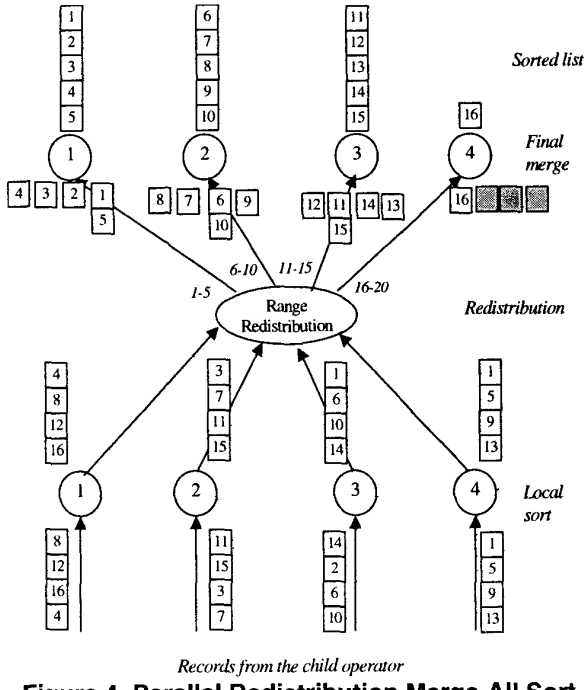

Figure 4. Parallel Redistribution Merge-All Sort

As shown in Figure 4, parallel redistribution merge-all sort is a two-phase method, where in phase one, local sort is carried out like in other methods, and in phase two, results from local sort are redistributed to all processors based on a range partitioning, and merging is then performed by each processor.

Like in parallel redistribution binary-merge sort, empty boxes drawn by gray boxes are actually empty list as a result of data redistribution. In the above example, processor 4 has three empty lists coming from processors 1, 3, and 4, as they do not have values ranging from 16-20 specified by the range partitioning function.

Also notice that the final results produced in the final merging phase is each processor are sorted, and these are also sorted among all processors based on the order of the processors specified by the range partitioning function.

The advantage of this method is the same as that in parallel redistribution binary-merge sort, including true parallelism in the merging process. However, the tree of parallel redistribution merge-all sort is not a tall tree as in the parallel redistribution binary-merge sort. It is in fact a one-level tree like in parallel merge-all sort.

Not only the advantages of parallel redistribution merge-all sort are mirroring those in parallel merge-all sort and parallel redistribution binary-merge sort, but also the problems. Skew problems found in parallel redistribution binary-merge sort also exist in this method. Consequently, skew modelling needs some simplified assumptions as well. Additionally, bottleneck problem in merging which is similar to that is parallel merge-all sort is also common here, especially if the number of processor is large and exceeds the limit of number of files can be opened at once.

3.5 Parallel Partitioned Sort

Parallel Partitioned Sort is influenced by the techniques used in parallel partitioned join, where the process is split into two stages: partitioning and independent local work [14,16]. In parallel partitioned sort, first we partition local data according to range partitioning used in the operation. Notice the difference between this method and others. In this method, the first phase is not a local sort. Local sort is not carried out here. Each local processor scans its records and redistribute or repartition according to some range partitioning.

After partitioning is done, each processor will have an unsorted list in which the values come from various processors (places). It is then local sort is carried out. Thus, local sort is carried out after the partitioning, not before. It is also noticed that merging is not needed. The results produced by the local sort are already the final results. Each processor will have produced a sorted list, and all processors in the order of the range partitioning method used in this process are also sorted. Figure 5 shows an illustration of this method.

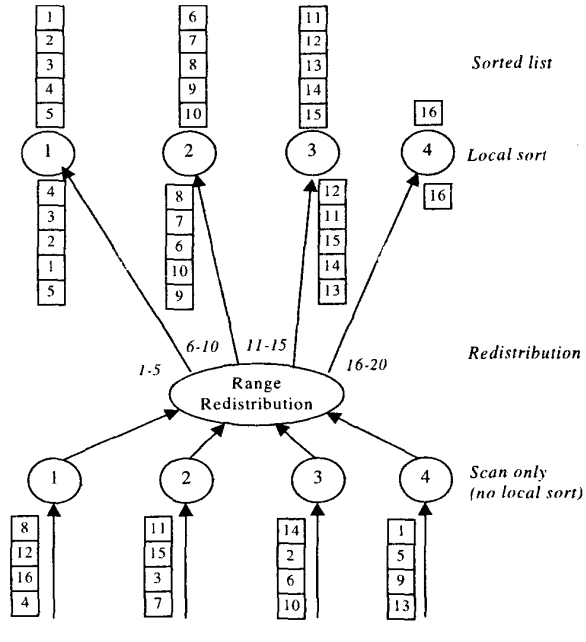

Figure 5. Parallel Partitioned Sort

The main benefit of parallel partitioned sort is that no merging is necessary, and hence the bottleneck in merging is avoided. It is also a true parallelism, as all processors are being used in the two phases. And most importantly, it is a one-level tree reducing unnecessary overhead in the pipeline hierarchy.

Despite these advantages, the problem remain outstanding is skew which is produced by the partitioning. This is a common problem even in the partitioned join. Load balancing in this situation is often carried out by producing more buckets than the available number processors, and workload arrangement of these buckets can then be carried out to evenly distribute buckets into processors [9,10,11]. For example, in Figure 6, seven buckets is created for 3 processors. The size of each bucket is likely to be different, and after the buckets are created, bucket placement and arrangement are performed to make the workload of the three processors balanced. For example, buckets A, B, and G go to processor 1, buckets C and F to processor 2, and the rests to processor 3. In this way, the workload of these three processors will be balanced.

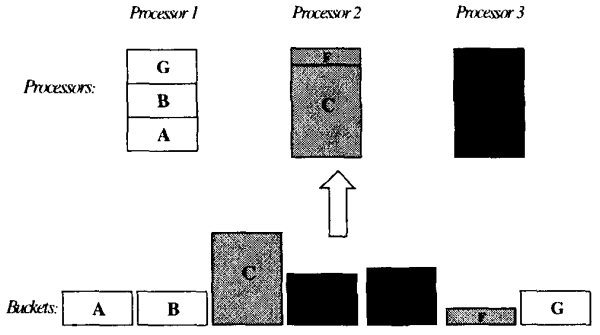

Figure 6. Bucket Tuning Load Balancing

However, bucket tuning in the original form as shown in Figure 6 is not relevant to parallel sort. This is because in parallel sort, the order of the processors is important. In the above example, buckets A will have values which are smaller than those in bucket B, and values in bucket B are smaller than those in bucket C, etc. Then buckets A to G are in order. The values in each bucket are to be sorted, and once they are sorted, the union of values from each bucket together with the bucket order produces a sorted list. Imagine that bucket tuning as shown in Figure 6 is applied to parallel partitioned sort. Processor 1 will have three sorted lists; from buckets A, B, and G. Processors 2 and 3 will have 2 sorted lists each. However, since the buckets in the three processors are not in the original order (i.e. A to G), the union of sorted lists from processors 1, 2 and 3 will not produce a sorted list, unless further operation is carried out. We reserve the discussion on load balancing for future work, and thus it is out of scope of this paper.

4 Conclusions and Future Work

In this paper, we have presented a taxonomy for parallel external sorting in high performance database systems. The taxonomy consists of five sorting methods, namely *parallel merge-all sort*, *parallel binary-merge sort*, *parallel redistribution binary-merge sort*, *parallel redistribution merge-all sort*, and *parallel partitioned sort*. The first two sorts are widely known through previous publications. We added with the other three methods, through the use of redistribution and repartitioning concepts.

Our future work is to investigate the behaviour of each of the parallel external sorting methods. This will include analytical analysis and performance measurements. We also plan to examine the skew problems and techniques to solve them.

References

[1] Bergsten, B., Couprie, M., and Valduriez, P., "Overview of Parallel Architecture for Databases", *The Computer Journal*, vol. 36, no. 8, pp. 734-740, 1993.

[2] Bitton, D., et al, "Parallel Algorithms for the Execution of Relational Database Operations", *ACM TODS*, 8(3), 1983.

[3] Bitton, D., et al, "A Taxonomy of Parallel Sorting", *ACM Comp. Surv.*, vol. 16, no. 3, pp. 287-318, September 1984.

[4] DeWitt, D.J., et al, "The Gamma Database Machine Project", *IEEE TKDE*, vol. 2, no. 1, March 1990.

[5] DeWitt, D.J. and Gray, J., "Parallel Database Systems: The Future of High Performance Database Systems", *Comm. of the ACM*, vol. 35, no. 6, pp. 85-98, 1992.

[6] Elmasri, R. and Navathe, S.B., *Fundamental of Database Systems*, 2nd ed., Benjamin/Cummings Publishing, 1994.

[7] Feldman, M.B., "Chapter 10: Sorting External Files", *Data Structures with Ada*, Prentice Hall, pp.290-300, 1985.

[8] Graefe, G., "Query Evaluation Techniques for Large Databases", *ACM Comp. Surv.*, 25(2), pp. 73-170, 1993.

[9] Hua, K.A. and Lee, C., "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", *Proc. of the 17th VLDB Conf.*, pp. 525-535, 1991.

[10] Hua, K.A., Lee, C. and Hua, C.M., "Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning", *IEEE TKDE*, 7(6), pp. 968-983, 1995.

[11] Kitsuregawa, M. and Ogawa, Y., "Bucket Spreading Parallel Hash: a New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)", *Proc. of the 16th VLDB Conf.*, Brisbane, pp. 210-221, 1990.

[12] Knuth, D.E., *The Art of Computer Programming: Sorting and Searching*, vol. 3, Addison-Wesley, 1973.

[13] Liu, K.H., Leung, C.H.C., and Jiang, Y., "Analysis and Taxonomy of Skew in Parallel Databases", *Proc. of High Perf. Comp. Symp. HPDC'95*, Canada, pp. 304-315, 1995.

[14] Mishra, P. and Eich, M.H., "Join Processing in Relational Databases", *ACM Comp. Surv.*, 24(1), pp. 63-113, 1992.

[15] Oracle, *Oracle 8 Server Concepts*, Release 8.0, 1998.

[16] Wolf, J.L., Dias, D.M and Yu, P.S., "A Parallel Sort Merge Join Algorithm for Managing Data Skew", *IEEE TPDS*, vol. 4, no. 1, pp. 70-86, January 1993.