On Disk Allocation of Intermediate Query Results in Parallel Database Systems

Holger Märtens

Universität Leipzig, Institut für Informatik, Postfach 920, D–04009 Leipzig, Germany
maertens@informatik.uni-leipzig.de

Abstract. For complex queries in parallel database systems, substantial amounts of data must be redistributed between operators executed on different processing nodes. Frequently, such intermediate results cannot be held in main memory and must be stored on disk. To limit the ensuing performance penalty, a data allocation must be found that supports parallel I/O to the greatest possible extent.

In this paper, we propose declustering even self-contained units of temporary data processed in a single operation (such as individual buckets of parallel hash joins) across multiple disks. Using a suitable analytical model, we find that the improvement of parallel I/O outweighs the penalty of increased fragmentation.

1 Introduction

In parallel database systems used for advanced applications like data warehousing, complex queries are performed on very large data sets, often in terabyte ranges. Parallel operators executed on different processing nodes exchange substantial amounts of intermediate results, and when the processors' memory capacity is exceeded, temporary data must be stored on disk. The response time problems caused by slow disk access are alleviated by parallel I/O, often using more disks than processors to avoid bottlenecks.

In a shared-disk architecture, intermediate results can be written out by the sender nodes and read directly by the receivers. Such *disk-based data transfer* is convenient and reduces the overhead of communication between processors. But depending on the operators' access patterns, a smart disk allocation is required to limit disk contention.

In most algorithms, data fragments are stored on many disks, but each fragment is kept on a single device. Thus, when a receiver processes its fragments sequentially, it can read from just one disk at a time and parallel I/O is not fully exploited. In this article, we propose declustering individual data fragments across multiple disks to increase the performance of parallel database systems for complex queries on large amounts of data. We develop an appropriate analytical model to show that the benefits of parallel I/O for the receiving operator usually outweigh the additional disk load due to increased fragmentation. Our approach works for several operators and most system architectures.

Our paper is structured as follows: Sect. 2 describes the processing model of a parallel hash join in a shared-disk system, which serves as a case study throughout the text. Sect. 3 is devoted to finding the optimal degree of declustering and includes our analytical model. In Sect. 4, we outline possible extensions of our method to different operators and architectures. Related work is discussed in Sect. 5, and we conclude in Sect. 6.

P. Amestoy et al. (Eds.): Euro-Par '99, LNCS 1685, pp. 469-476, 1999.

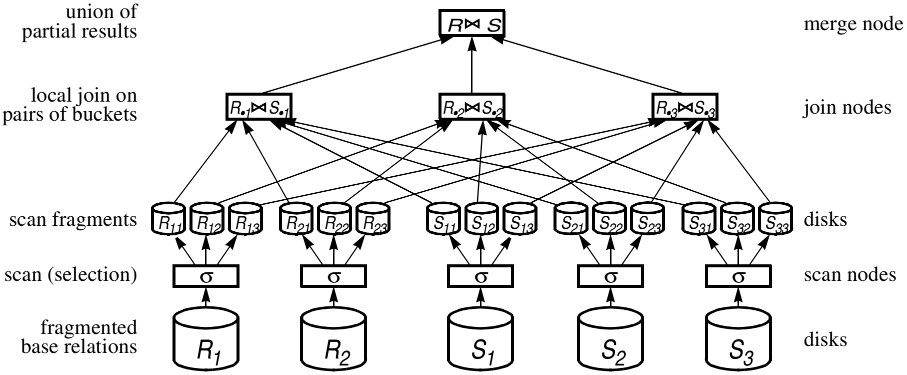

Fig. 1. Processing model of a parallel hash join in a shared-disk architecture

2 Parallel Hash Joins in Shared-Disk Architectures

For a clear presentation, we restrict most of this paper to a concrete case: a parallel two-way hash equi-join in a shared-disk environment. We now describe the basic processing model before we give some heuristics for parameter selection and disk allocation.

2.1 Processing Model

Let R and S be the inner and outer relations of a join query, declustered across r and s disks, respectively. In the *scan phase*, R is read by n scan nodes which apply a selection and partition their output into b buckets. If the scan result is very large, the buckets cannot be held in main memory and are stored on d disks. S is processed similarly, possibly by a different number of scan nodes but with a corresponding partitioning of buckets.

In the *join phase*, m join nodes each process one bucket pair at a time, using hash joins in which a hash table is built from an R-bucket and the matching S-bucket is probed against it. The local results are merged at a specified processor. This model is illustrated in Fig. 1. In the shared-disk environment we assume, the allocation of buckets to processors can be chosen dynamically to balance the workload.

For large data sets, each join node processes several bucket pairs and each disk must hold several buckets. Also, any scan node can contribute to any bucket, creating $b \cdot n$ *scan fragments*. Consequently, disk contention between processors occurs. To limit contention while supporting parallel I/O, buckets must be properly allocated to disks. We introduce the parameter v, denoting the degree of bucket declustering. With each bucket split across multiple disks, parallel reading is enabled. Assuming the same degree of declustering for all buckets, $b \cdot v$ *bucket fragments* are stored on disk.

Thus, the parameters n, m, d, b, and v must be found for a two-way equi-join query.

2.2 Selection of Basic Parameters and Disk Allocation

Our main concern is finding an optimal degree of declustering (v), which we discuss in detail in Sect. 3. Before that, we provide a simple heuristic for the remaining parameters and a disk allocation scheme as a basis for further calculations.

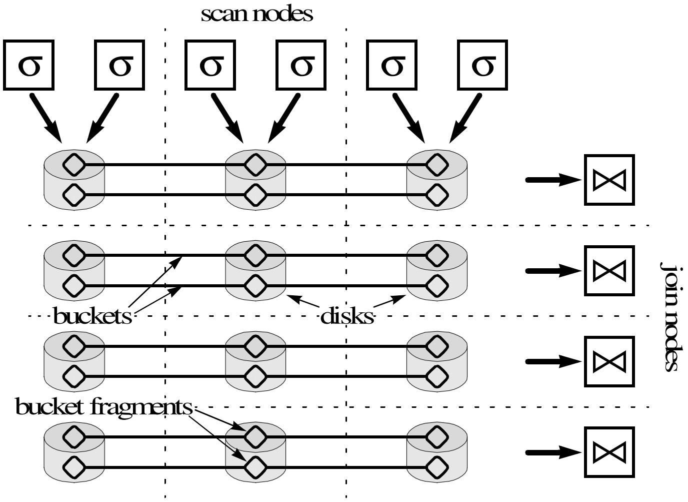

Fig. 2. Example of the processing model and the allocation scheme. Eighty buckets (not all shown) are processed using six scan nodes and four join nodes. The buckets are declustered across twelve disks with a degree of three. To minimize access conflicts, each disk is used by just two scan nodes and one join node. The parameters from Sect. 2.1 are set as follows:

$$n = 6 \qquad m = 4 \qquad d = 12 \qquad b = 80 \qquad v = 3$$

The numbers of processors and disks—*n*, *m*, and *d*—should be set so as to match their processing rates as closely as possible, starting from the degree of declustering *r* of relation *R*. These calculations involve some information on system performance (preferably reflecting the current load state) as well as the selectivity of the scan, which is estimated by histograms or sampling. The number of buckets, *b*, should be large enough to fit each bucket into a single node's available memory (which may be just a fraction of physical memory). When data skew occurs, bucket sizes will vary and *b* must be selected high enough for the largest bucket to meet the memory restriction.

Example. Let base relation *R* be scanned by six processors. If each node's output rate is sufficient to keep two disks busy, twelve disks are used to store the buckets. If, in the join phase, one node can process the data delivered by three disks, only four join nodes are required. Assuming a total scan result of 800 MB with 10 MB of memory available on each of the selected processing nodes, 80 buckets are created (no skew). This example is depicted in Fig. 2; the allocation and declustering applied are justified below. ❑

For a given selection of all five parameters, the following allocation scheme (also exemplified in Fig. 2) can be shown to yield the smallest number of processors accessing the same disks, thus minimizing disk contention. As in most studies, we assume integer proportions between some parameters for simplicity.

1. Arrange the d disks into a matrix of v columns and d/v rows (one disk per cell).
2. Assign $b \cdot v/d$ buckets to each row, declustering every bucket across v disks.
3. Assign n/v scan nodes to each column; make them write to the d/v disks there.
4. Assign $m \cdot v/d$ join nodes to each row; let them read from the v disks in that row.

3 Determining the Degree of Declustering

In this section, which constitutes the core of our study, we trade off parallel I/O against disk contention to determine the optimal degree of bucket declustering, v. We define and analyze basic indicators of processing performance before introducing our analytical model that leads to the final solution. All of these considerations are based on the above allocation scheme. Details omitted due to space restrictions are discussed in [5].

3.1 Performance Indicators

Since disks are normally shared by several nodes, the available degree of I/O parallelism cannot be measured simply by the number of disks a processor can access at a time. Rather, it must be interpreted as the number of disks available divided by the number of nodes accessing them. With each scan node writing to d/v disks and a disk shared by n/v nodes, the available degree of *write parallelism* is $wp = d/n$, which is independent of v. In the join phase, each node reads from v disks to assemble its current bucket. It may have exclusive access to these if $m \cdot v \leq d$; otherwise, a disk is shared by $m \cdot v/d$ processors. Thus, the degree of *read parallelism* is $rp = min(v, d/m)$.

Disk contention can be defined by the number of concurrent read and write operations per disk. With k such operations running at a given time, the disk read-write head will have to move between k different positions, and the resulting seek times constitute the allocation-dependent share of I/O cost. In the scan phase, k is the number of bucket fragments per disk. Thus, *write contention* is measured as $wc = b \cdot v/d$. During the join, a disk is accessed by only one node if $m \cdot v \leq d$ or shared by $m \cdot v/d$ processors otherwise (as mentioned above). With each node reading from a single position, *read contention* is $rc = max(1, m \cdot v/d)$.

In the example from Sect. 2.2 and Fig. 2, the performance indicators have the following values: $wp = 2$, $rp = 3$, $wc = 20$, $rc = 1$.

Observations. Although these coefficients are not proportional to either performance or response times (their precise effects are analyzed in Sect. 3.2), we can make some general observations: While both write and read contention are best avoided for low values of v, higher degrees of bucket declustering are useful to support read parallelism. Write parallelism, however, is constant; thus, we need not regard *wp* any further.

Let us examine three common-sense settings of v, viz.: no declustering ($v = 1$), full declustering ($v = d$), and *read-optimal* declustering ($v = d/m$). The latter is so named because it just allows full read parallelism without introducing read contention. As can be inferred from Table 1, the optimal degree of declustering must be between 1 and d/m. In this interval, there is a true trade-off between parallelism and contention. For $v \geq d/m$, however, contention is increased without further gains in parallelism. To find the true optimum within the range of $v \in [1, d/m]$, we have devised an elaborate analytical model which is presented in the next section.

3.2 Analytical Model

We construct a cost function to capture the total disk response time for the I/O of the join buckets. $T = T_w + T_r$ comprises writing and reading in the scan and join phase, respectively. Let p be the total number of pages (or other suitable, uniform I/O *gran-*

Table 1. Development of performance indicators for different degrees of declustering

	contention		***parallelism***	
declustering	***write***	***read***	***write***	***read***
none ($v = 1$)	low	none	constant	low
read-optimal ($v = d/m$)	medium	none		high
full ($v = d$)	high	high		high

ules) to be written into the buckets. If all d disks are busy all the time (neglecting skew), then $T_w = p/d \cdot t_w$, where t_w is the average time for a single write operation, estimated as $t_w = 1/k_w \cdot t_s + (1 - 1/k_w) \cdot t_l$.

Here, k_w denotes the number of disk access positions used at the time of writing. There is a probability of $1/k_w$ that the disk head need *not* be moved because it is already in the right position from the previous access. This case causes a "short" disk access (t_s); otherwise, a "long" access (t_l), including a track seek operation, occurs. This is the most important distinction to make when modeling disk activity because seek times are known to dominate disk response times [11]. Note that average values of t_s and t_l are quite sufficient for our purposes since we are interested in the overall sum of access times only. Defining $t_\Delta = t_l - t_s$, we can now simplify $t_w = t_l - t_\Delta/k_w$.

The number of access positions on each disk, k_w, corresponds to the number of bucket fragments per disk plus an adequate number of entry points for concurrent queries in multi-user mode, x. The term x can be composed of arbitrary sub-terms; we are only interested in its average total magnitude. While this model of multi-user mode may seem simplistic, we will see later that it is quite sufficient. For now, $k_w = b \cdot v/d + x$.

Note that our formula does not include waiting times caused by write requests not being served immediately. Rather, we assume asynchronous access so that processing can continue while data is (queuing to be) written. We further presume that the disks are kept busy but are not overloaded; this assumption is justified because we specifically selected the ratio of disks and processing nodes so as to match their processing rates (cf. Sect. 2.2). Thus, our model need only capture the actual disk access times.

For read operations in the join phase, we can assume only $m \cdot v$ disks to be used at the same time (each of the m join nodes assembles its current bucket from v disks). Note that $m \cdot v$ cannot exceed d because we have limited v to a maximum of d/m in the previous section. Now, we can define $T_r = p/(m \cdot v) \cdot t_r$ with $t_r = t_l - t_\Delta/k_r$, similar to the scan phase. The number of access positions, however, is lower now because we can exclude contention within the current join: $k_r = 1 + x$.

We assume x to have the same value as in the scan phase to represent the same degree of inter-query contention. After some more transformations, we can write the complete cost formula as a function of v:

$$T(v) = \frac{1}{v} \cdot \left(\frac{p \cdot t_l}{m} - \frac{p \cdot t_\Delta}{m + mx}\right) - \frac{1}{bv + d \cdot x} \cdot p \cdot t_\Delta + \frac{p \cdot t_l}{d} \ . \tag{1}$$

To find its minimum within the bounds of $v \in [1, d/m]$, we discern several cases.

Single-User Mode. With $x = 0$ to represent single-user mode, T simplifies to

$$T(v) = \frac{p}{v} \cdot \left(\frac{t_s}{m} - \frac{t_\Delta}{b}\right) + \frac{p \cdot t_l}{d} . \tag{2}$$

The properties of this function depend on the relationship of t_s/m to t_Δ/b or, rearranging the terms, of b/m to t_Δ/t_s. If both should happen to be equal—in other words: if the number of buckets per join node corresponds to the ratio of disk seek time and short access time—the function is constant and all values of v are equivalent.

If b/m is greater (many buckets), total I/O cost strictly decreases with v because the performance gains from parallel reading outweigh the losses due to write contention. In this case, v should be selected as large as possible, i. e. $v = d/m$. If b/m is less than t_Δ/t_s (few buckets), the opposite applies and disk contention dominates. Now, a small value of v is appropriate, i. e. $v = 1$.

Multi-User Mode. In multi-user mode, the cost function cannot be simplified, and lengthy calculations ensue. However, it can be shown that $T(v)$ is strictly decreasing if

$$\frac{b}{m} \geq \frac{t_\Delta}{t_l - t_\Delta/(1+x)} . \tag{3}$$

This condition is true for most sensible parameters (i. e. $b/m \geq 2$ and $x \geq 1$). In other words: Unless we are "almost" in single-user mode ($x < 1$, meaning that there is just one competing operation per disk at any given time), or we process just one bucket per join node, we should decluster the buckets with a degree of $v = d/m$.

If no such property can be ensured, more case distinctions are required. We found that for all cases of $x \geq 1$, the degree of declustering should be set to d/m. For some very small values of x, declustering should be avoided. This corresponds to "near"-single-user mode with few buckets per join node as above. There is only a very narrow margin of values of x for which the optimum of v is within the interval $[1, d/m]$.

Analysis. The results for both single- and multi-user mode can be interpreted as follows: For a high number of buckets, disk contention in the scan phase is already severe because there are many fragments on each disk, causing a very low probability of "short" write times. Thus, further increasing k_w through declustering has little effect on the scan phase while the join phase is sped up considerably through parallel reading. This is true even when inter-query contention affects both phases. This result also justifies our choosing a simple coefficient like x: It is unnecessary to use a more complex term that will still exceed the boundary of 1 in any true multi-user system.

With few buckets, there is still a significant share of short write operations that are destroyed by declustering, outweighing the performance gain during the join. Note that the number of data pages, p, does not directly influence the number of seek positions although b usually increases with p. Also, the ratio t_Δ/t_s varies with the size of the read/write granule; the larger the granule, the more useful bucket declustering will be.

Summary. Looking for an optimal degree of bucket declustering, we found that in all practical cases, the read-optimal setting from Sect. 3.1 is favorable. The only notable exception is for small numbers of buckets in single-user mode; in this case, declustering should be avoided. Medium degrees of declustering are not useful to consider.

4 Extensions

To account for the second relation, S, the computations of n, d, and m work as in Sect. 2.2. However, the declustering of base relations they start from, r, must now be replaced with $r + s$, $max(r, s)$, or the like, depending on whether R and S are stored on disjunct disks. If R and S result from sub-queries, their processing rates must be used. For the number of bucket pairs, b, the previous heuristics must be applied to the *inner* relation (usually the smaller one). To find an appropriate value of v, we can also use the previous rules but have to interpret v differently, e. g., $v_R = v_S = v$ (for separate scanning as in standard hash joins) or $v_R + v_S = v$ (for simultaneous scanning as in sort-merge joins).

Our approach is not restricted to two-way equi-joins. In principle, it is applicable to all blocking operators that exchange large amounts of data (exceeding main memory) in a many-to-many relationship. Possible applications include non-equi-joins, distribution sort (cf. Sect. 5), and several types of aggregation, especially when combined with group-by clauses. Some adaptations of the allocation scheme may be required for operators with different access patterns.

Our approach assumes that every processor can access any disk. Thus, our method can be used in shared-disk and shared-everything systems, some hybrid architectures, or certain variants of NUMA. It can even be adapted to shared-nothing architectures by transferring the data through the network and having the receivers write the buckets back to their local disks as above, provided that each node owns at least d/m disks. However, shared-nothing architectures are less flexible in dynamic task allocation, complicating load balancing and/or causing a higher communication overhead [4, 9].

5 Related Work

While parallel I/O in general is naturally applied in parallel database systems, declustering of single data units such as join buckets has received little attention. Of the operators we mentioned, aggregation and grouping have not been associated with this idea.

In the context of joins, most load balancing studies have focused on CPUs and main memory [1, 4, 8, 10, 13]. While the significance of I/O has been asserted, only its overall reduction has actually been addressed [10]; declustering is either not performed or not discussed. For shared-nothing architectures, *bucket spreading* (full declustering) was introduced to equalize skew effects [3], but optimizing I/O was not a primary goal.

Mergesort algorithms naturally provide for parallel reading; in addition, workfiles may be striped across disks. Full striping is indeed found useful [14] especially in multiuser mode if the striping unit corresponds to the read granule; workload is balanced across disks by randomization. These results, however, cannot be easily generalized due to the particular access patterns of the mergesort operator. For distribution sort, which is more similar to joins and aggregation, full striping of single files is used to achieve parallel I/O [7]. Again, optimal declustering is not an aim.

Disk arrays automatically provide parallel I/O. But even though allocation strategies have been developed for various applications [6, 12], disk arrays cannot address the particular allocation requirements of different algorithms. Specifically, independent join buckets are best stored on disjunct devices to allow reading them without contention; automatic (possibly full) striping in disk arrays usually defeats this goal [2].

6 Conclusion

In this paper, we have investigated ways of allocating intermediate results of large database queries across the disks of a parallel system. Based on a well-founded analytical model for the sample case of join queries, we concluded that in most cases, it is useful to decluster even individual join buckets across several disks to enable parallel reading in the subsequent query stage. The benefits of parallelism usually outweigh the penalty of disk contention. The optimal degree of declustering is such that the receiving processors can keep all disks busy without introducing intra-query contention.

Our results are applicable to several different operators and largely independent of the underlying system architecture. To the best of our knowledge, this is the first study that has considered bucket declustering in such a general context. In the future, we plan to validate our results by simulation studies for various architectures and workloads.

Acknowledgment. The author would like to thank Dr. Dieter Sosna for his help in handling the cost function used in Sect. 3.2.

References

1. DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: Practical Skew Handling in Parallel Joins. Proc. 18th VLDB Conference, Vancouver (1992) 27–40
2. Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Computing Surveys, Vol. 25, No. 2 (1993) 73–170
3. Kitsuregawa, M., Ogawa, Y.: Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC). Proc. 16th VLDB Conference, Brisbane (1990) 210–221
4. Märtens, H.: Skew-Insensitive Join Processing in Shared-Disk Database Systems. Proc. IDPT Conference, Vol. 2, Berlin (1998) 17–24
5. Märtens, H.: Disk Scheduling for Intermediate Results of Large Join Queries in Shared-Disk Parallel Database Systems. IfI-Report Nr. 9/98, Universität Leipzig (1998)
6. Merchant, A., Yu, P.S.: Analytic Modeling and Comparisons of Striping Strategies for Replicated Disk Arrays. IEEE Trans. Computers, Vol. 44, No. 3 (1995) 419–433
7. Nodine, M.H., Vitter, J.S.: Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors. Proc. 5th SPAA, Velen (1993) 120–129
8. Omiecinski, E.: Performance Analysis of a Load Balancing Hash-Join Algorithm for a Shared-Memory Multiprocessor. Proc. 17th VLDB Conference, Barcelona (1991) 375–385
9. Rahm, E.: Dynamic Load Balancing in Parallel Database Systems. Proc. Euro-Par '96 Conference, Lyon (1996) 37–52
10. Rahm, E., Marek, R.: Dynamic Multi-Resource Load Balancing in Parallel Database Systems. Proc. 21st VLDB Conference, Zürich (1995) 395–406
11. Ruemmler, C., Wilkes, J.: An introduction to disk drive modeling. IEEE Computer, Vol. 27, No. 3 (1994) 17–28
12. Scheuermann, P., Weikum, G., Zabback, P.: Data Partitioning and Load Balancing in Parallel Disk Systems. VLDB Journal, Vol. 7, No. 1 (1998) 48–66
13. Wolf, J.L., Dias, D.M., Yu, P.S., Turek, J.: New Algorithms for Parallelizing Relational Database Joins in the Presence of Data Skew. IEEE Trans. Knowl. Data Eng., Vol. 6, No. 6 (1994) 990–997
14. Wu, K.-L., Yu, P.S., Chung, J.-Y., Teng, J.Z.: A Performance Study of Workfile Disk Management for Concurrent Mergesorts in a Multiprocessor Database System. Proc. 21st VLDB Conference, Zürich (1995) 100–109