On Flexible Allocation of Index and Temporary Data in Parallel Database Systems

Erhard Rahm
Holger Märtens, Thomas Stöhr

University of Leipzig, Germany

1 Introduction

Data placement is a key factor for high performance database systems. This is particularly true for parallel database systems where data allocation must support both I/O parallelism and processing parallelism within complex queries and between independent queries and transactions. Determining an effective data placement is a complex administration problem depending on many parameters including system architecture, database and workload characteristics, hardware configuration, etc. Research and tool support has so far concentrated on data placement for base tables, especially for Shared Nothing (SN), e.g. [MD97]. On the other hand, to our knowledge, data placement issues for architectures where multiple DBMS instances share access to the same disks (Shared Disk, Shared Everything, specific hybrid architectures) have not yet been investigated in a systematic way. Furthermore, little work has been published on effective disk allocation of index structures and temporary data (e.g., intermediate query results). However, these allocation problems gain increasing importance, e.g. in order to effectively utilize parallel database systems for decision support / data warehousing environments.

In the next section we discuss the index allocation problem in more detail and introduce a classification of various approaches that are already supported to some degree in commercial DBMS. While SN offers only few options, the other architectures provide a higher flexibility because index allocation can be independent from the base table allocation. For certain index-supported queries, this can allow for order-of-magnitude savings in I/O and communication cost. We then turn to the disk allocation of intermediate query results for which the allocation parameters can be chosen dynamically at query run time. For the case of parallel hash joins, we outline how to determine an optimal approach supporting a high degree of parallelism. The work discussed is performed within a project aiming at developing strategies to automatically determine optimal data allocation strategies in order to simplify system administration in high performance environments.

2 Index Allocation

Determining an index allocation comprises tasks similar to determining the data placement of tables, namely specifying the distribution granules (fragments), calculating the degree of declustering, and allocating the fragments to disks or (in SN systems) to processing nodes. The latter subproblem can be solved by standard techniques, e.g. to achieve a balanced distribution of access frequencies.

SN systems typically use a horizontal partitioning of tables based on a hash or range partitioning on an attribute TPA (table partitioning attribute). Index allocation follows the table allocation in order to allow each processing node to locally access its data. As a result, index support on a particular attribute IA results in a local subindex per node on attribute IA. For an index query on IA (different from TPA), each node would perform a local index scan on its subindex. For selective queries this is clearly much more efficient than a complete table scan. Still, the communication overhead to start and terminate *p* subqueries and the I/O cost of *p* local index scans can be substantial, e.g. for exact match queries on the primary key (example: *account* table with *branchID* as TPA and *acctNo* as IA).

For Shared Disk (SD) - as well as in other architectures where multiple DBMS instances can directly access the same data - there are more options for index allocation. This is because there is no need to partition an index structure among processing nodes and because index partitioning may be different from table partitioning. The main index allocation alternatives are classified along two dimensions in Fig. 1. At the first level, we consider *logical* index partitioning affecting processing (query) parallelism and the size of the index search space. At level two, we consider *physical* aspects such as the degree of declustering affecting I/O parallelism.

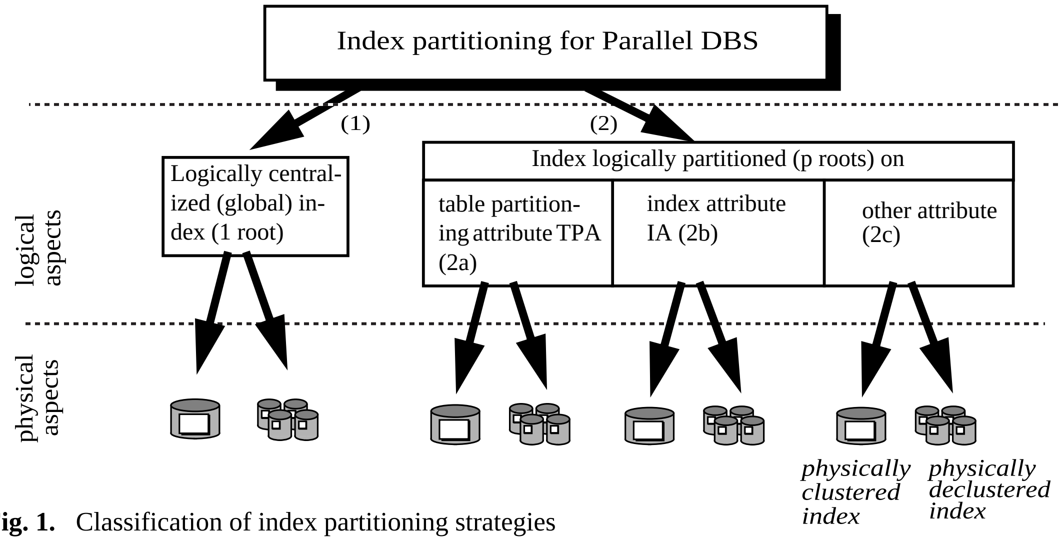
Index partitioning for Parallel DBS
(1)
(2)
Logically centralized (global) index (1 root)
Index logically partitioned (p roots) on
table partitioning attribute TPA (2a)
index attribute IA (2b)
other attribute (2c)
logical aspects
physical aspects
physically clustered index
physically declustered index

Fig. 1. Classification of index partitioning strategies

Approach 1 is to have no logical index partitioning but to simply use a single index tree[1] (1 root) as in centralized DBMS. Such an approach is also referred to as a *global index*. Global indices allow minimal storage and access cost because there is only 1 index tree. Depending on index size and access frequency, a global index may be completely stored (clustered) on a single disk or to the minimal number of disks if the index size exceeds the disk capacity, or it may physiscally be declustered across multiple disks, e.g., at the page level. *Physical clustering* is the simplest approach that avoids the need to specify a degree of declustering, size of the declustering granules etc. Furthermore, an index access is confined to a single disk so that occupying multiple disk arms is avoided which may restrict I/O rates. On the other hand, it supports the least degree of parallelism so that it is primarily of interest for small indices or highly selective queries. Even in these cases, mutli-user operation may lead to disk bottlenecks unless most of the index data can be cached in main memory. *Physical declustering* can overcome these restrictions by supporting improved I/O parallelism and load balancing in multi-user mode (inter-query parallelism) [RS95, SWZ98]. It may be transparent to the DBMS, e.g., when supported by a disk array controller.

Global indices allow optimal processing of selective queries. For instance, an exact match query on a primary index can be processed with a single index traversal. SN systems, on the other hand, may require *p* subqueries and *p* local index traversals for such queries causing order-of-magnitude higher processing overhead (communication, I/O). Intra-query parallelism for larger index queries is more difficult to achieve[2] and one of the reasons for logically partitioned indices.

[1] Due to space restrictions we concentrate on B*-tree index structures.

[2] One possibility is to logically partition only the leaf level of a global index, similar to [KFK96].

Logical index partitioning (approach 2) divides an index structure into multiple subindices (p roots) with respect to an attribute IPA (index partitioning attribute). This approach can limit the search space for index queries referring to the IPA by only processing the relevant set of subindices. Furthermore, all selected subindices can be processed in parallel thus supporting intra-query parallelism without introducing (disk) contention between subqueries. Logical index partitioning can be combined with physical partitioning by declustering a single logical partition across multiple disks to improve I/O parallelism.

The IPA may be different from the index attribute IA. In particular, the IPA may correspond to the TPA which is typically the case for SN. That is SN index allocation is usually restricted to case 2a with the additional limitation that the fragmentation and processor allocation (degree of declustering) coincides for tables and indices irrespective of index size and access frequencies. As already pointed out, this often leads to a maximal processing parallelism and high overhead which is particularly harmful for smaller index queries.

SD-like architectures may also use a logical index partitioning on the table partitioning attribute, if the table is logically partitioned. In this case, table and index scans on the TPA can be restricted to a subset of the table data / subindices in order to reduce the amount of work to be performed. Furthermore, range queries on the TPA covering multiple partitions can be processed in parallel without disk contention between subqueries. In contrast to SN, the number of index partitions can be different from the number of table partitions and the index data may be assigned to different disks than the tables. Furthermore, the degree of intra-query parallelism can be chosen smaller than the number of relevant subindices (to limit the communication overhead) and can be determined at query run time (e.g., depending on the query size and the current load situation).

Moreover, in SD-like architectures logical index partitioning may be based on the index attribute IA itself (case 2b) or other attributes than TPA or IA or attribute combinations (2c). This again allows us to limit the index work to the really relevant subindices as well as to support intra-query parallelism.

All in all, we have a large spectrum of possible index allocation schemes with both logical partitioning and physical declustering possibilities. To better understand the various trade-offs, we plan a comprehensive performance study for various workloads and configurations, including data warehousing scenarios with parallel star joins in multi-user mode. Current SD implementations like Oracle Parallel Server [Or97] already support logically partitioned indices, however without giving sufficient help on how to use them optimally[3].

3 Disk Allocation of Temporary Data

One important aspect in a self-tuning PDBS is the efficient storage and retrieval of temporary data, such as large intermediate query results. Frequently, such temporary data is exchanged between operators running on different processing nodes. When its size exceeds the memory capacity of the participating nodes, the temporary data or parts of it must be stored on disk. Declustering the temporary data across multiple disks is required in order to support I/O parallelism [Wu95]. This type of data allocation can be determined at query run time.

In SD-like architectures, intermediate results can be written out by the sender nodes and directly read in by the receivers. Such a disk-based data transfer is convenient and reduces the overhead

[3] In [Or97, Or99] it is just remarked that an index should be partitioned by TPA for decision support queries and by index attribute for OLTP queries, respectively. It remains unclear how to determine the degree of declustering for tables and indices, where to allocate index partitions, how to incorporate physical declustering, how to deal with mixed workloads, etc.

of communication between processing nodes that can be substantial for SN[4]. But with each receiver getting multiple fragments from every sender - as in many join, sort, and aggregation operations - a smart disk allocation is required to limit disk contention while supporting a high degree of I/O parallelism. For the latter we propose to even decluster individual fragments across multiple disks.

We illustrate our approach [Mä99] for parallel hash join processing in a SD environment with n scan nodes and m join nodes (Fig. 2). The scan output is written to d disks where d should be selected such that the disks can write as fast as the n scan nodes produce their data. We assume the join nodes to perform local hash joins and thus we partition the join input into b buckets. A given bucket contains input from all scan nodes but is processed by exactly one join node. Bucket partitioning, especially selection of b, is performed so that the hash table of any bucket can be kept entirely in the main memory currently available on the join nodes. A major aspect then becomes how to perform the disk allocation of buckets, in particular how to determine the degree of declustering v per bucket, in order to support I/O parallelism and to control disk contention.

With these parameters, it can be shown that the minimal number of processors concurrently accessing a disk is achieved by a matrix-like arrangement of the d disks into v columns and d/v rows as illustrated in Fig. 2. If $b \cdot v/d$ buckets are allocated to each row, each bucket can be declustered across the v disks of that row. Now, n/v scan nodes can be assigned to each column and write their output to the bucket partitions stored there. Similarly, $m \cdot v/d$ join nodes can then read and process the buckets in a row, each reading in parallel from v disks.

Based on this general allocation scheme, we must now select the degree of bucket declustering, v, in such a way as to minimize the overall disk access times. This involves a trade-off between parallel I/O and disk contention as can be seen from the following cases:

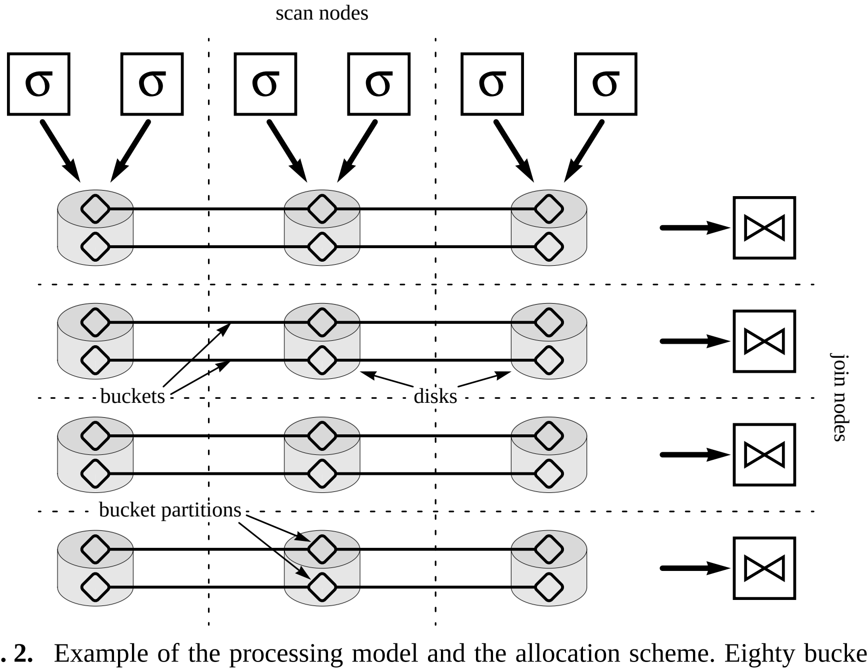

Fig. 2. Example of the processing model and the allocation scheme. Eighty buckets (not all shown) are processed using six scan nodes and four join nodes (n=6, m=4, b=80). The buckets are declustered across twelve disks with a degree of three (d=12, v=3). To minimize access conflicts, each disk is used by just two scan nodes and one join node.

[4] Smaller intermediate results not requiring overflow I/O should always be transferred directly by inter-processor communication. This is much faster than a disk-based data transfer and supports pipeline parallelism.

- A straight-forward approach is to distribute the buckets across all d disks, but to keep individual buckets on a single device ($\boldsymbol{v = 1}$). However, this limits read parallelism because each join node has to sequentially access each bucket from a single disk. In addition, there is a high degree of write contention because each scan node contributes to each bucket resulting in concurrent access of each disk by any of the n scan nodes.
- $\boldsymbol{v = d}$ results in a maximal declustering of any bucket across all disks. This supports a high degree of I/O parallelism but suffers from serious read contention. This is because each join node has to access all disks for each bucket.
- An intermediate case with good read performance is obtained for $\boldsymbol{v = d/m}$. This supports read parallelism without any read contention while at the same time the full bandwidth of all d disks is used for join processing.

In order to quantify the performance of the various allocation alternatives, we have developed a detailed analytical model for the disk access times as a function of v [Mä99]. It turned out that in most practical cases, the setting of $v = d/m$ is the best approach because it provides optimal read performance with acceptable write performance. The example in Fig. 2 is based on this case.

Currently, we are in the process of validating our analytical model by simulation. Furthermore, we are adapting the allocation approach to other types of operations, such as merge joins, non-equi joins, sorting, and various types of aggregation.

4 Conclusions

Effective data allocation for index and temporary data in parallel database systems is an important area that has not yet sufficiently been investigated in the research community. To ease system administration, commercial DBMS have to provide sophisticated tool support for index allocation. This is particularly the case for SD-like architectures in order to exploit their high optimization potential as illustrated by our classification scheme. In addition, flexible data allocation for large intermediate results is to be supported at query run time. Our approach for declustering temporary data may be a good starting point for this.

References

KFK96 Koudas, N., Faloutsos, C., Kamel, I.: *Declustering Spatial Databases on a Multi-Computer Architecture*. Proc. EDBT96, LNCS 1057, 592-614, 1996

MD97 Mehta, M.; DeWitt, D.: *Data Placement in Shared Nothing Parallel Database Systems.* VLDB Journal 6 (1), 1997

Mä99 Märtens, H.: *On Disk Allocation of Intermediate Query Results in Parallel Database Systems*. Proc. Europar99 Conf., LNCS, Springer 1999

Or97 *Taking Advantage of Partitioning in Oracle8.* Oracle Technical White Paper, www.oracle.com, 1997

Or99 Miscellaneous Oracle8 server manuals.
www.irm.vt.edu/oracle_803_docs (June 25, 1999)

RS95 Rahm, E., Stöhr, T.: *Analysis of Parallel Scan Processing in Shared Disk Database Systems*. Proc. Europar95, LNCS 966, 485-500, 1995

SWZ98 Scheuermann, P., Weikum, G., Zabback, P.: *Data Partitioning and Load Balancing in Parallel Disk Systems.* VLDB Journal 7(1), 48-66, 1998

Wu95 Wu, K. et al.: *A Performance Study of Workfile Disk Management for Concurrent Mergesorts in a Multiprocessor Database Systems*. Proc. VLDB95 conf., 1995