UNIVERSITÄT LEIPZIG

Institut für Informatik

Options in Scan Processing for Shared-Disk Parallel Database Systems

Holger Märtens

Report Nr. 8 (1998)

ISSN 1430-3701

Options in Scan Processing for Shared-Disk Parallel Database Systems

Holger Märtens

University of Leipzig
maertens@informatik.uni-leipzig.de

July 1998

Abstract:

Shared-disk database systems offer a high degree of freedom in the allocation of workload compared to shared-nothing architectures. This creates a great potential for load balancing but also introduces additional complexity into the process of query scheduling.
This report surveys the problems and opportunities faced in scan processing in a shared-disk environment. We list the parameters to tune and the decisions to make, as well as some known solutions and commonsense considerations, in order to identify the most promising areas of future research.

Keywords:

parallel databases, shared-disk systems, load balancing, query optimization

1 Introduction

In the debate on the best architectures for parallel database systems (PDBS), many researchers have prematurely focused on shared-nothing (SN) environments, ignoring the superior potential for load balancing found in shared-disk (SD) systems. As a contribution to the discussion, this report presents a comprehensive survey of the choices to be made and the gains to be achieved in parallel scan processing in SD database systems.

In the following chapters, we will list a number of questions to answer, problems to solve, and parameters to tune in a shared-disk PDBS. This list will focus on, but not be limited to, the differences between SD and SN with respect to scan processing. While we will try to derive some approximate solutions from previous work, rough analytical models or plain common sense, this paper is mainly devoted to exploring the enormous decision space and finding the most interesting spots for closer scrutiny. These will be further examined in later work.[1]

Our paper is structured as follows: The remainder of this chapter reviews some fundamentals of shared-disk PDBS and introduces parts of the terminology. Chapter 2 lists some basic parameters to consider for scan processing in SD environments, while chapter 3 identifies and discusses the three main categories of scan queries. We study the specific approach of affinity-based scan scheduling in chapter 4 before we close with an outlook on future research in chapter 5.

1.1 PDBS architectures

In common terminology [DG92, Va93], there are three main architectures of parallel database systems, which are illustrated in figure 1: *shared everything* (SE), *shared nothing* (SN), and *shared disk* (SD). The three approaches differ in their complexity, their scalability, and their load balancing potential, all of which are vital to performance in terms of both response times and throughput of transactions.

- In SE systems, all processors share access to main memory as well as to disk storage, allowing for great flexibility in processing. But due to heavy contention arising from concurrent memory and disk access, SE is not scalable beyond a few dozen nodes and cannot be used for very large databases demanding high throughput of transactions and queries. Thus, we will not regard it any further. It is, however, a good choice for smaller applications.
- In SN architectures, on the other hand, both disks and main memory are partitioned among processing nodes. As this paradigm eliminates contention between processors, such systems can be scaled to hundreds or even thousands of nodes. They achieve high throughput especially for on-line transaction processing (OLTP), which is characterized by short, independent transactions that access only a few records of data, requiring little communication between nodes [NZT96]. For complex queries, however, SN systems face load balancing problems because the place of processing a data item is largely predetermined by its storage location.
- As a compromise, SD architectures aim at combining the advantages of SE and SN while avoiding their drawbacks by partitioning main memory and sharing disk access. Although the memory bottleneck is eliminated, similar contention can now occur on the disks, and complex concurrency and coherency control schemes are necessary. On the other hand, any node can process any data, enabling good load balancing even for complex queries that access arbitrary data.

Although several successful commercial database systems such as Oracle and DB2/MVS exist for shared-disk architectures [Ra94], most academic researchers have dismissed them due to

1. Our work is funded by the Deutsche Forschungsgemeinschaft (DFG) within the project “Datenallokation und dynamische Lastbalancierung in Parallelen Datenbanksystemen”.

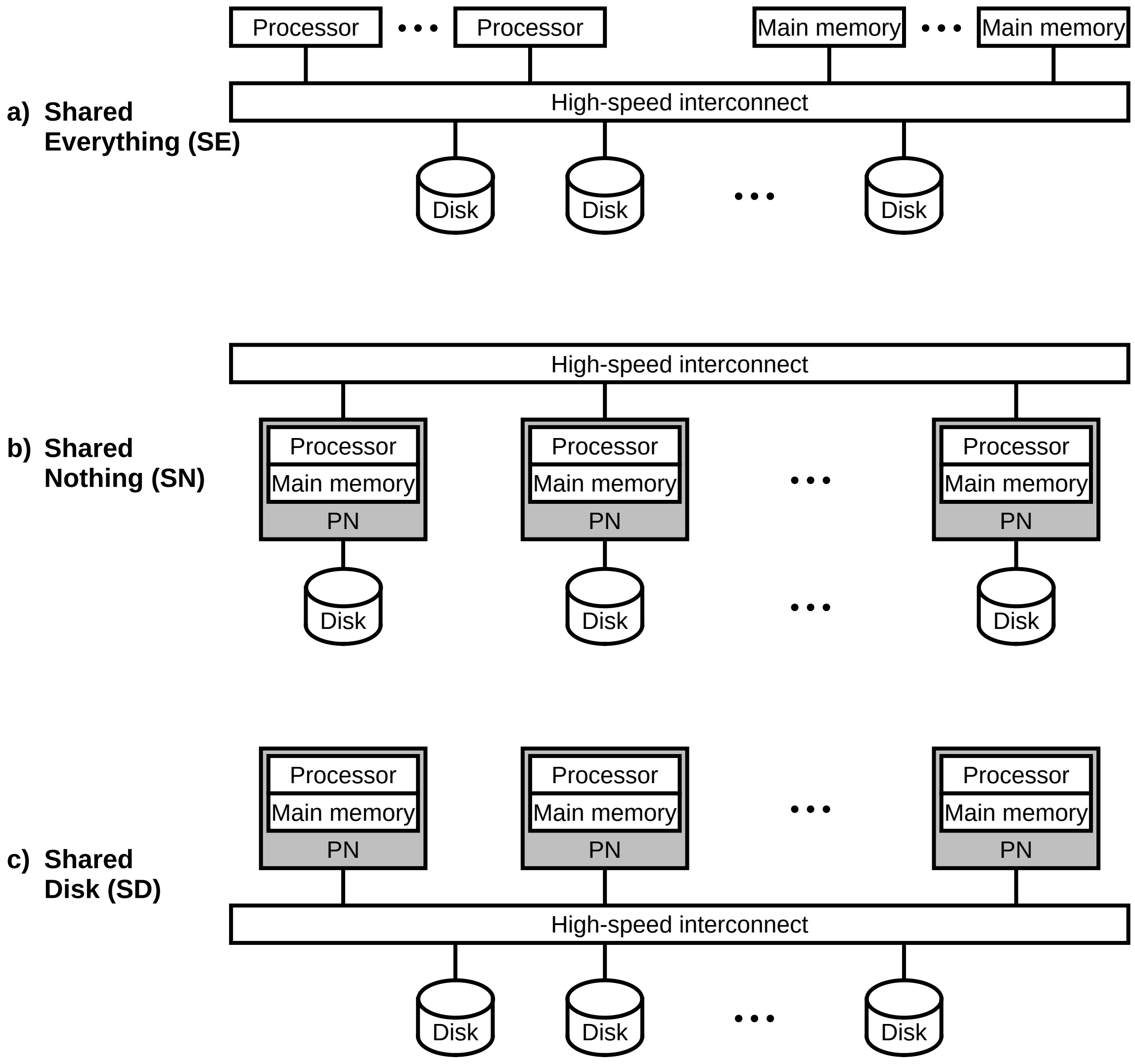


Figure 1: PDBS architectures (adapted from [Ra94])

their complexity with regard to concurrency and coherency control [DG92]. We believe, however, that their superior load balancing potential justifies further research to objectively assess the pros and cons of shared-disk database systems.

1.2 Terminology and assumptions

Before we present our ideas on efficient scan processing in shared-disk PDBS, we shall briefly introduce a few technical terms as well as some of the basic assumptions underlying the remainder of this report.

For the database system under consideration, we presume a *homogeneous* architecture in which all *processing nodes* (PNs) have the same capabilities and processing power. In particular, each node can access any disk and process any query. This does not contradict the fact that some PNs may be reserved for special purposes (cf. section 2.2.2). The system may or may not be *dedicated*, i. e. there might be other software running on it apart from the DBMS, which will add to the problem of load balancing.

We assume the DBS to be based on the *relational data model*. When a *query* (which we will consider synonymous to a *transaction*) enters the system, it is assigned to a *coordinating node* (or *coordinator*) that is responsible for *scheduling* its execution. The coordinator may delegate (parts of) this responsibility. Scheduling can be *static*, using a fixed number of processors and a fixed or random set of PNs, or *dynamic*, depending on the properties of a particular query and/or on the current system state. In the remainder of this paper, we will assume dynamic scheduling. For every query, we are interested in its *response time* (the timespan from its submission to the system to the end of processing), which is usually longer than its *processing time* (the timespan for processing itself = the response time minus an initial queuing delay).

For simplicity, we usually assume a scan's *selection predicate* to refer to a single attribute, unless otherwise noted. The share of tuples satisfying the predicate is termed the *selectivity* ($sel \in [0,1]$). We speak of a *high* selectivity if *sel* is close to 1, and of a *low* one if it approaches 0.[2] When considering complex queries of which a scan is only the first *stage*, we will occasionally mention *blocking operators*. These are query operators that collect their entire input before they can produce output tuples, thus blocking any pipelining and potentially requiring a large amount of memory. Blocking operators include sorting, duplicate elimination (as sometimes included in a projection), and various types of aggregation.

We will study both *inter-* and *intra-query parallelism*, i. e. the concurrent execution of independent queries as well as parallel processing within queries. In addition, we assume *inter-* and *intra-operator parallelism*, meaning that for any transaction, both separate operators and different instances of the same operator (working on different subsets of data) may be processed at the same time, on one or more nodes. In fact, ways of parallelizing a single scan will be the main focus of this report, but *multi-user mode* will be an underlying assumption in all cases.

When data is read from disk in parallel, we must take into account its *declustering*, i. e. the way the data is distributed across the disks in the system. We differentiate *logical declustering*, which is controlled by and visible to the DBMS itself, and *physical declustering*, which is determined by the file system, the operating system, or some lower levels of the DBMS[3]. Since physical declustering, as understood in this paper, is unknown to the query optimizer, it cannot be taken into consideration during processing, and we will not regard it any further in this report. Logical declustering, however, can be exploited for scheduling. It usually depends on the values of a specified *declustering attribute*, such that the same values are stored in the same place; most declustering functions are either *range-based* or *hash-based*. The parts of a relation produced by declustering are called *fragments*, and the sum of fragments residing on the same disk is termed a *partition*. For simplicity, we will usually assume each partition to equal one fragment.

Closely related to the declustering of data is the *granularity* of reading and processing, referring to the amount of data accessed at a time; this amount is called the *granule*. Similar to selectivity, granularity is said to be *high* or *low* if the granules are large or small, respectively. The size of a granule is determined by *prefetching* and *read-ahead* factors. While both terms describe multiple data pages being read before they are actually needed, read-ahead implies control by the query operator whereas prefetching is initiated by the buffer manager. (As a consequence, prefetched pages may be forced from the buffer before they are referenced, so that they have to be read again.) Both strategies can reduce disk access times through *continuous reading*, in

2. This requires an explanation because some authors understand a query of *high selectivity* to be "very selective", i. e. *sel* is close to 0 (conversely for *low*).
3. Physical declustering is also applied in a *disk array* [PGK88], which we will view for our purposes as a single disk of high capacity and bandwidth. It has been stated that control over declustering (as well as prefetching and other parameters) is a desirable feature of parallel file systems [CK93]. Since some operating systems (such as most UNIX variants) do not offer this kind of control, many commercial DBSs operate on so-called "raw devices" [Gr93].

which adjacent pages are read from disk in a single operation, saving multiple adjustments of the disk's read/write head.

2 General considerations

In this section, we will consider some of the more general concepts pertaining to processing in shared-disk PDBS. These include basic parameters to tune for each query (e. g. multiprogramming level, degree of parallelism) as well as some secondary variables that characterize the system state (e. g. load measures, resource consumption). We will also include some heuristics to find appropriate parameter settings and discuss the basic trade-offs involved.

Most of the ideas given in this section apply to different kinds of database operations, not only to scans. However, since they are essential to the performance of processing, we will list them anyway, focussing on the aspects relevant to scan queries.

2.1 Inter-query parallelism

The *multiprogramming level* (MPL) denotes the maximum number of queries that may be processed at any point in time; further incoming queries are stored in a waiting queue. It thus denotes a *degree of inter-query parallelism* that can be defined

- for the system as a whole and/or for each processing node in the system;
- for complete queries and/or for sub-queries generated from them;
- for queries in general and/or for classes of queries sharing common access patterns.

Combinations of these are possible.

The MPL is one of the fundamental choices to make in a PDBS. With too few queries in the system, resources such as CPUs, disks, and main memory may become idle because there is no transaction ready to use it, even though there may be queries queuing for entrance into the system. If there are too many transactions, competition for shared resources may slow down each of them as well as the system as a whole. In fact, the absolute amount of work will be increased with the MPL due to additional process switches, disk track seeks (in the case of concurrent access), and memory paging.

Defining MPLs by query classes enables *priorities* to be implemented. Queries of different classes can then be treated according to their importance in the allotment of resources. Apart from the MPL itself, priorities may affect the degrees of parallelism, the amounts of buffer space, the priorities of operating system processes, and the order of disk access. In this respect, shared-disk environments have a higher flexibility than shared-nothing systems due to the freedom of processor allocation.

Furthermore, the MPL is the most important tool in systemwide load control: When it is exceeded, further queries will be suspended until others have left the system. But although MPLs can be set for different query classes defined according to their resource requirements, it is rather difficult to represent in them the various types of bottlenecks that can occur during processing. When one or more of the resources CPU, disk, memory, and communication network gets overloaded, reacting on this by MPL adjustment means to identify the query classes that most use the scarce resource(s) and to lower their MPLs only. (The simpler solution of reducing all MPLs is very rough and may leave some of the non-bottleneck resources idle.) On the other hand, query priorities are straightforward to integrate into an MPL-based load control scheme: When system load becomes too high, MPLs for high-priority transactions are kept up as long as possible, while less important queries are suspended.

In [MD93], the choice of MPL (for different classes of queries defined by their memory requirements) was effectively combined with a strategy for buffer allocation. While their algorithm was restricted to single-node systems, an extension to shared-disk PDBSs is being considered for [Mü98], although primarily in the context of join queries. The drawback of this approach is that it is restricted to just one type of resource.

2.2 Intra-query parallelism

The choice of processing nodes for a newly arrived query can be decomposed into two (interrelated) tasks: determining the degree of parallelism ("How many nodes?") and finding the actual set of PNs ("Which nodes?"). This freedom of choice is the characteristic feature of the shared-disk architecture, enabling a balance of workload and thus a stable performance.

2.2.1 Parallelization

Generally speaking, it would be desirable for any processing task to be parallelized as broadly as possible in order to reduce its response time. However, there are three main obstacles to this goal:

- As stated in *Amdahl's Law*, any task contains a certain share $s \in [0,1]$ that cannot be parallelized. Thus, the maximum performance improvement to be achieved by parallelization can be no greater than by a factor of $1/s$.
- Similarly, the partitioning of work has to take place on a certain level of granularity. As a consequence, there will always be some largest sub-task of relative size $r \in\ ]0,1]$ that has to be processed on a single node and thus sets a lower bound to response time. A *degree of parallelism* (DP) beyond $\lceil 1/r \rceil$ will – theoretically – provide no further improvement.
- In practice, parallelization causes a certain overhead for partitioning and assignment of tasks, which can decrease or even outweigh the performance gains achieved through a high degree of parallelism.

As can be seen from these statements, the choice of DP strongly depends on the possibilities of task partitioning. In the case of scans, the latter is largely determined by the declustering of data because it is usually disadvantageous to have multiple processors scan the same disk. But while it would be rather straightforward to infer an optimal degree of parallelism from the partitioning of the data concerned, things are more complicated in multi-user mode. For high DPs, the number of sub-tasks (within the system as well as on each PN) will increase, leading to the same competition overhead on the level of sub-queries as described in section 2.1 between whole queries. In addition, the parallelization process itself will put further load on the system.

In short, while a high DP is desirable to decrease response times of single queries, a low DP is best to ensure a high throughput of transactions per time unit. A related trade-off exists between CPU load (which is reduced by a low DP) and memory or disk load (which is reduced – for each single node[4] – by a high DP) [Ra96].

For scans in particular, there are three main reasons to reduce the degree of parallelism in situations of high system load:

- the parallelization overhead on CPUs and network;
- the (possibly) increased disk contention through concurrent reads by multiple sub-queries;
- the buffer requirements for each sub-query.

The latter can also be kept in check by restricting buffer usage to one page at a time instead of read-ahead optimization. Obviously, this will slow down the query (as reducing the DP would). In addition, it will increase disk load because continuous reads (cf. section 2.3) are no longer supported.

4. Disk load might also decrease systemwide with a higher degree of parallelism if the query concerned needs a large amount of memory. In such situations, I/O to temporary files may be reduced or even avoided. However, this applies mainly to join queries (as stated e. g. in [Ra96] referring to [RM95]) rather than to scan queries.

Note that a reduction of the DP under high load should not be handled simply by introducing a sub-query MPL; this could lead to unfairness among queries. For example, if the MPL would allow 23 more sub-queries in the system, a transaction might be assigned a DP of 20. The next incoming query, possibly with similar properties, would have to make do with a DP of 3 or be suspended until the system load has decreased. A fair solution must reduce DPs in situations of high load *before* the MPL is exhausted.

2.2.2 Processor allocation

Choosing a set of PNs according to the degree of parallelism can be further divided into two steps. In the first step, one has to determine the nodes *eligible* to processing the query in the first place. There are several restrictions to this:

Apart from heterogeneities in the system architecture that prevent some nodes from executing certain tasks (which we will not discuss in this paper), it is possible to logically partition the processors into groups that are reserved for particular purposes. For example, it may make sense to route OLTP transactions and complex queries to different sets of processors according to their disparate resource and data requirements. Furthermore, a logical partitioning of data among PNs may require a query to be processed in a particular place as in the case of *data affinity*. The idea here is to assign (sub-)queries to processing nodes based on the subset of data they access, so that queries using the same data are processed on the same nodes. There are several advantages to this method:

- Disk contention is reduced because each disk is accessed by fewer PNs and there is a larger share of continuous read operations (decreasing disk seek times).
- With each PN accessing only a subset (*region*) of the data, buffer hit rates are improved, saving some read operations and reducing disk load and response times accordingly.
- Locking overhead can be significantly reduced if the partitioning of regions among the PNs corresponds to the distribution of locking authority and page ownership (cf. section 2.5).

In a way, affinity scheduling puts the system into a kind of “shared-nothing mode” implying both the advantages (as just described) and the main disadvantage of SN systems, namely the imbalance of load between processors. The important point, however, is that data affinity can be abandoned when an imbalance is detected, and reinstalled later on. As a compromise, affinity can be moderated by assigning overlapping regions to processors, such that several of them are allowed to process the same query. This makes it easier to balance the load without completely losing the benefits of data affinity. This illustrates the obvious trade-off between data affinity and load balancing.

Among the nodes that are generally capable of processing a task, some may not be available due to their current load state. In particular, this applies to limited resources such as main memory. When a set of eligible nodes has been found, it may turn out that the degree of parallelism determined earlier cannot be supported. In this case, the query may be suspended until more suitable PNs become available; otherwise, a reevaluation of the DP is necessary. After that, the set of eligible nodes may have changed (either with time or due to the – now different – requirements of the single [sub-]tasks), and the process may repeat several times until a DP and a set of possible nodes have been found. Alternatively, the two functions may be more closely integrated (see below).

In the second step, the actual set of processors to use must be selected. In general, one will choose the least loaded nodes in order to ensure fast processing. While the most common load measure applied here is the current CPU usage, other aspects, such as the quantity of free memory, may be taken into account. For scan queries, memory consumption seems to be of lesser concern because in general, only a small amount of data pages has to be buffered. Thus, any node with enough space for a few prefetched or read-ahead data pages can process the scan (unless it is to include blocking operators from the next query stage). When memory is plentiful, the read granule can even be increased to reduce disk load.

2.2.3 Integrated policies

It may often be useful to determine the degree of parallelism and the set of nodes in a single step rather than sequentially; this would enable choices that are impossible or at least hard to achieve in separate policies. For instance, the backcoupling of DP setting and node selection mentioned in the previous section could be integrated into a single scheme “collecting” eligible PNs until either arriving at a sufficient aggregate capacity or running out of nodes, and then processing the query on these. In addition, one may want to change the DP and/or the set of PNs for a query after it has been started in response to a change in the overall load situation, according to the concept of dynamic scheduling (cf. section 2.6). Some simple examples of such policies are:

- “Greedy” scheduling: The query uses all nodes eligible at scheduling time and acquires more as they become available during processing, possibly allocating all PNs in the system.
- “Incremental” scheduling: If the number of eligible nodes is less than the DP calculated in the first step, the query uses all of these and later acquires more until the DP is reached.
- “Nice incremental” scheduling: An “incremental” algorithm is enhanced to release nodes at the request of another query with a higher priority or in a situation of high system load.

The latter example uses the concept of query priorities as introduced in section 2.1. In all three examples, tasks can be reallocated off nodes that are no longer eligible due to increased load.

At closer scrutiny, the idea of data affinity discussed in the previous section may also be considered an integrated policy because the DP is bounded by the set of PNs responsible for the data concerned.

2.3 Disk access

It is evident that the response time of a query heavily depends on the sequence of read operations it initiates and on the amount of collisions that ensue. The most influential factor in this context is certainly the allocation of the data on disk, both for base relations and for intermediate results. However, both of these are out of the scope of this paper. For our purposes, we will assume the alloction of base data to be fixed and try to find appropriate processing schemes for it.[5]

What remains to be discussed is the order of read accesses on the disks as far as it is not predetermined by the static allocation of data. Consider the following rules of thumb:

- Disk accesses should be distributed evenly over time. It is disadvantageous to have high contention for some periods of time and to leave the disks idle during others.
- Contiguous data sets should be read in a single operation. This is significantly faster than pagewise processing.
- Different regions of the same disk should be accessed sequentially. It is a waste of time to make the disk head jump to and fro between tracks, trying to read them in parallel.

Naturally, there are trade-offs between these and other goals of query optimization. For example, having to read a large set of data in a single run will cause queries waiting for different data to be suspended. This would be unfair especially towards short transactions that could otherwise be finished in a fraction of the time. It might also require a large amount of memory to store read-ahead data. But the faster a query is processed, the sooner it can release its buffers as well as its locks. Finally, there may be a substantial CPU overhead for scheduling disk access within or even between queries; when queries reside on different processing nodes, coordination between them may well be prohibitive. Note that the latter problem is introduced by the shared-

5. An overview of allocation related issues is found in [SWZ96]. For the special case of intermediate results in join queries, we are currently working on a related report [Mä98a].

disk architecture itself; in shared-nothing, all requests to a single disk are issued by the same PN, which makes for cheaper scheduling.

The approach we suggest to solve the problem of disk scheduling is based on a variant of data affinity; see chapter 4 for details.

2.4 Network load

The load put on the interconnection network is determined by data transfer among PNs on the one hand and between PNs and disks on the other hand. While the latter need not be considered in detail because all arguments can be directly derived from the discussion on disk access, the former can be further divided into exchange of actual data and of control information.

In shared-disk systems, data transfer can be significantly reduced in comparison to shared-nothing because data that is to be processed together (e. g. for a join or an aggregation) can be accessed directly by the same PN. Since scans can be partitioned quite well, data transfer is not a direct problem for them, but additional operations such as selection, projection, or aggregation are often included into the scan for efficiency. Some of these may be blocking operators that require a view of the complete data set, so that such operators cannot be fully parallelized. To avoid excessive data transfer, small scans may be executed sequentially on a single node with the blocking operator included in them. Large scans, however, must still be parallelized for satisfactory response times, and the blocking operation will be performed separately after a transfer of intermediate results.

The amount of control information transferred between processing nodes depends on the degree of parallelism and on the scheduling protocol. With a large DP and/or close cooperation between nodes, coordination messages will increase; when PNs can work largely independently, they can be reduced. The degree of dependency between PNs is influenced by the partitioning of work. In a centralized scheduling scheme, the network load will additionally be biased towards the coordinator node.

2.5 Concurrency and coherency control

Criticism of shared-disk systems is often directed towards their locking overhead. It is argued that, since data can be accessed by any node, the amount of communication required for concurrency control might be prohibitive [DG92]. The same argument applies to buffer coherency control. To alleviate this problem, some very effective approaches have been found [Ra93], e. g. reducing the number of messages passed by integrating concurrency and coherency control, by implementing hierarchical locking schemes, and by distributing the locking authority. Still, locking overhead does exist, and its scale depends on the way the query is processed.

While this paper is not actually concerned with locking schemes, we will keep in mind that the exploitation of data affinity (as understood in either section 2.2.2 or chapter 4) will have the extra benefit of reducing the locking overhead if the PN processing a data item also has the lock authority for it; this node should also be the owner of the primary copy for coherency control.

2.6 Skew effects and scheduling strategies

The term *skew* denotes an uneven distribution of data and/or workload across disks or processing nodes. There are different types of skew that have been categorized for join queries in a classical paper by Walton et al. [WDJ91]. Leaving out the join-specific aspects, we arrive at the following types of *data skew* (DS) relevant to scan processing:

- *Attribute value skew* (AVS) describes variation in the frequency of attribute values. It is a property of the data independent of storage and processing strategies.
- *Partition skew* (PS) means an uneven distribution of tuples across disks and/or processors. It is a frequent consequence of AVS and can occur in two forms during scans:

- *tuple placement skew* (TPS), which refers to the initial partitioning of the data on disk;
- *selectivity skew* (SS), which denotes the difference in a query's selectivity on different subsets of the data.

Note that partition skew is characteristic of a logical declustering of base data. In the case of physical declustering only, we can expect both TPS and SS to be largely equalized due to the random allocation of data. The effect of partition skew on scan processing is evident: When tasks are shared out to PNs on the basis of disk partitions, TPS will immediately lead to a load imbalance. Even if partitions have the same size, SS can cause variations in workload. Either way, the consequence is *execution skew* (ES), which is characterized by disparate execution times among sub-tasks and thus leads to suboptimal response times. It is important to note that one can have

- TPS without AVS (depending on the declustering function used);
- SS without AVS or TPS (induced by the selection predicate and the declustering function);
- ES without DS (because of concurrent queries or DBMS-external workloads).

In general, skew can be remedied only when its magnitude is known, and not all information on skew is available before the query is started. This leads to the distinction of two fundamental ways of handling skew to achieve a balance of load:

- *Predictive scheduling* attempts to gather as much information as possible before starting query execution and produces a complete schedule in advance based on the findings. This schedule is then carried out without changes.
  Predictive scheduling of scans is based on the measurable parameters such as partition sizes, declustering functions, and current system load, as well as on estimates of attribute value distributions and selectivities. Such estimates are usually gained either from routine statistics such as *histograms* or by *sampling*. Histograms are usually more efficient to maintain, but they are necessarily coarse in order to be generally applicable. Sampling, on the other hand, requires a larger overhead but is naturally optimized for the query at hand. Both approaches have been extensively treated in the literature, and near-optimal techniques have been found [SN92, PI96, PI97, MVW98]. By common sense, we consider histograms sufficient (if at all needed) for relation and clustered index scans, while sampling may be worthwhile for non-clustered index scans where SS can do more harm.
- *Runtime scheduling* tries to balance the load dynamically according to the actual progression of work. It avoids the inaccuracies of advance estimations and uses the observed actual values instead. It can often do with far less parameters than predictive scheduling, using execution times as the main guideline. In fact, a "pure" type of runtime scheduling would do without any advance scheduling at all and assign just one task at a time to each processing node involved. This approach was successfully applied to join processing in [Mä98], where it was termed "on-demand scheduling".
- A specialization of runtime scheduling is *reactive scheduling*. It is based on a – possibly partial – predictive schedule that is usually generated in a simplified, ad-hoc way. When this schedule is executed (and extended as necessary), the system keeps track of possible deviations from it. When a substantial difference is detected, rescheduling measures are taken for compensation. This often implies the *migration* of tasks already begun to another PN, which is relatively cheap for scans that do not have a large context to move (as opposed to e. g. the hash table of a hash join operator).[6]

Clearly, all methods entail a certain overhead. While predictive scheduling requires more complex computations at start-up time, runtime scheduling can incur delays from calculations re-

6. Another variant of runtime scheduling is to compute predictively a *dynamic execution plan* with *choose-plan operators*. These contain multiple alternative schedules for their respective sub-queries, one of which is selected at runtime. This approach has been successfully tested in *Volcano* [CG93].

quired during query execution; there is no general rule as to which approach is more expensive. In addition, the success of predictive scheduling depends on the amount of information available at the time of optimization as well as on the accuracy of its estimates. This is often a problem for complex queries in which errors can increase exponentially with the number of operators [IC91], but it may be less significant for scans alone.

3 Classes of scan queries and appropriate strategies

When discussing the various strategies of scan processing, it is useful to classify queries according to their possibilities of index usage. When there is no index to support processing, the entire relation must be scanned to find matching tuples; this case is aptly called a *relation scan*. When an index can be used, we further differentiate whether it is *clustered* (i. e. the data pages are sorted according to the indexed attribute) or *non-clustered* (otherwise). This gives us three different *scan modes* that are dealt with in quite different ways.

3.1 Relation scan

A relation scan is used not only when there is no index fitting the query; it can also be chosen when the optimizer decides that, even with an index, most of the data pages have to be processed anyway. In such a case, a relation scan can be more efficient because

- it avoids the overhead of index access itself;
- it allows for continuous reads (whereas index usage usually induces pagewise, scattered reading);
- it is easier to parallelize.

Either way, all data pages of the relation concerned must be read and processed. Thus, the sequence of page accesses is (trivially) known in advance, and scheduling becomes relatively straightforward:

Since we are concerned with very large databases, we assume that there is a large amount of pages to be read. Thus, there is a lot of elbowroom to apply the scheduling rules listed in section 2.3, the most important one being the use of continuous reads. As a consequence, each disk that has a partition of the data in question should be read by exactly one node because concurrent access by two or more nodes would invariably lead to scattered reads. Note that the possibility of several PNs *continuously* reading from the same disk is only a theoretical one for shared-disk because it requires too much communication between the nodes. In addition, processors can usually scan more pages per second than a standard disk can deliver, so that sharing a disk between nodes is unnecessary. (If, however, PNs should be so overloaded that they are slower than disks, it would seem appropriate to either suspend the new query or to assign it a low degree of parallelism, again arriving at a single node per disk. The case of disk arrays is discussed later in this section.)

Though each disk should be accessed by only one processor, each node can still read from several disks. In fact, if l is the estimated relative speed of processors vs. disks, each PN should ideally read from $\lfloor l \rfloor$ disks at a time. Note that l can be measured either statically based on the system's average performance, or dynamically according to the current load.[7] In the latter case, the lack of predictability of future loads must be compensated by appropriate heuristics. Either way, the parameter l should take into account the selected conditions of disk access, such as read-ahead and prefetching factors. Still, l must be chosen with great care. When it is set too low (e. g. 1, which is a common preset value), response times will remain suboptimal; when it is set too high, each disk will remain underloaded but be scanned for a longer period of time, risking more contention between queries. In addition, the load on PNs and disks will usually vary over

7. A detailed (if straightforward) way of estimating disk and processor speeds is presented in [MD95].

time when other queries enter and leave the system. When in doubt, a more conservative (i. e. lower) value of *l* seems reasonable because disk contention harms not only response times but also throughput of transactions. For instance, the following parameters may be used as a basis of estimates for *l*:

- the current disk and processor load at the time of query optimization;
- the average load over a recent interval of time;
- the average load typical of the current weekday and time of day;
- either of the above minus some tolerance value.

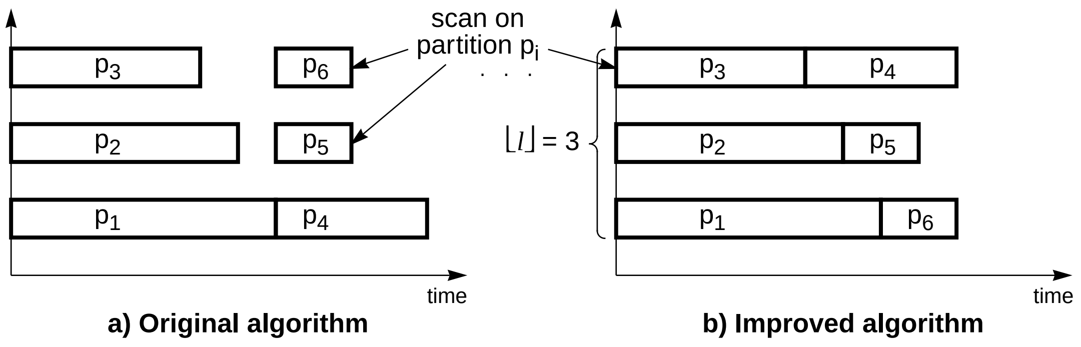
Having found an appropriate value of *l*, the desired degree of parallelism can be derived as $dp_d = \lceil d / \lfloor l \rfloor \rceil$, where *d* denotes the number of disks over which the data to be scanned is declustered. If dp_d exceeds the number of processors eligible for processing the query, the true degree of parallelism *dp* must be selected lower than this (unless one prefers to suspend the query until more PNs are available). The most reasonable choice in this case would be any factor of dp_d, i. e. $dp = \lceil dp_d / k \rceil$ for some integer *k*. The same rule applies if *dp* is to be reduced due to a general overload of the system.

Example. A relation declustered over 24 disks is submitted to a relation scan. The PNs are assumed to process a page 3.1 times as fast as the disks. It follows that $dp_d = \lceil 24 / \lfloor 3.1 \rfloor \rceil = 8$, i. e. the scan is best processed by eight nodes each scanning three disks simultaneously.
Assume now that only seven nodes are available to process the query. The degree of parallelism should then be halved: $dp = \lceil dp_d / 2 \rceil = 4$. Now, each PN will process six partitions, in two runs of three at a time. Note that, while it might seem reasonable to set the DP somewhere in between (e. g. $dp = 6$, with each node reading from four disks), this would still require the processors to work in two runs per node (of two partitions each, or of three and one partitions, respectively). While processing is only slightly speeded up[8], there is an unnecessary overhead stemming from the higher degree of parallelism. ❑

Having selected a degree of parallelism for the query, the next task is to choose the processing nodes themselves. The best choice appears to be the *dp* least loaded nodes (with respect to CPU utilization) among those eligible for the query, where eligibility can be defined so as to exclude certain nodes that are reserved for specific tasks (cf. section 2.2.2).
Finally, the partitions (and thus the disks) to be scanned can be split up among the selected nodes. In general, the LPT (longest processing time first) scheduling rule will deliver good results. It orders the partitions by size (defined in the obvious way, i. e. by the number of pages) and, starting with the largest one, shares them out among the PNs. Each partition is assigned to the node that so far has the least accumulated number of pages to scan. Since the partitions can vary in size, it may be necessary to set a limit to the number of partitions per node to ensure that none of them requires an extra run to process a large number of small partitions. This limit is calculated as $max_{\text{partitions per node}} = \left\lceil \frac{d/dp}{\lfloor l \rfloor} \right\rceil \cdot \lfloor l \rfloor$, i. e. the average number of partitions per node rounded up to the next multiple of the run size, $\lfloor l \rfloor$. This heuristic is based on the assumption that an additional run, even for a less loaded node, would increase response time more than a slight load imbalance will. This assumption may not hold for severe partition skew (see section 3.1.1 for details on skew handling).

Disk arrays. The discussion of DP and processor selection so far is based on the assumption that a single node can handle the data simultaneously delivered from one or more disks. While this is true for traditional disk farms, it no longer applies to disk arrays that consist of several disks presenting themselves to the outside as a single device of high bandwidth. With such hard-

8. Note that we presume here a conservative choice of *l*. If *l* is selected unreasonably high at the outset, there may be a significant performance gain when it is lowered. Otherwise, we assume the difference to be negligible.



The original scheme causes idle times between runs. The improved method reduces these by starting the scan on new partitions earlier, leading to a shorter response time.

Figure 2: Improvement in scan processing

ware, the speed ratio l will usually be lower than 1, and the above calculations are no longer appropriate. Instead, we suggest a more or less reverse approach:

With each processor delivering a fraction l of a disk array's processing speed, the scan on a single partition should be performed by $\lceil 1/l \rceil$ PNs simultaneously. To this end, the partition has to be logically split into $\lceil 1/l \rceil$ parts. This is straightforward if the partition consists of several fragments, especially if there is a multiple of $\lceil 1/l \rceil$ fragments in a partition. Otherwise, partitions have to be divided along page boundaries such that each node is assigned approximately the same amount of data. For a runtime scheduling scheme, (cf. section 3.1.1), it may be useful to define more sub-partitions than there are processors involved, in order to achieve a fine granularity of load balancing. Again, this may be easy with multiple fragments per partition. ❏

When an assignment of partitions to processing nodes has been found, the actual scan can begin. In the original scheme, each node would process its partitions in runs of $\lfloor l \rfloor$ at a time. However, this can be improved on by starting the scan of the next partition as soon as the first one is finished, so that $\lfloor l \rfloor$ partitions are being processed at *any* point of time (except towards the end). This will reduce idle times as depicted in figure 2. Again, a node should process its partitions in order of size.

3.1.1 Dealing with skew effects

The processing strategy presented so far can equalize slight imbalances of data and load distributions by means of its LPT scheduling rule. For greater skew, however, additional measures have to be taken:

If tuple placement skew (TPS) is heavy, i. e. if the sizes of partitions vary strongly, we can abolish the rule of assigning no more than $max_{\text{partitions per node}}$ tasks to each PN. Instead, we can apply the LPT policy without restriction, approximately equalizing the number of pages to be read by each node, rather than the number of partitions. On each processor, the previously mentioned optimization from figure 2 will ensure a near-optimal utilization of disk bandwidth even for an odd number of tasks because the run size $\lfloor l \rfloor$ is no longer a relevant parameter in the case of heavy TPS. With disk arrays, however, the best we can do is to have each partition read by as many nodes as to best exploit the bandwidth provided; a higher number of processors will provide no further speed-up but will merely increase contention.

With selectivity skew, similar considerations apply. The effort of copying and storing selected tuples adds to the processing load for each partition and should in fact be included in the cost function used for LPT scheduling. This way, a partition with a higher selectivity will appear "larger" due to the additional work, and scheduling can be performed as described before. The selectivity of each partition can be predicted on the basis of histograms or sampling (cf. section 2.6). With runtime scheduling, the corresponding values can be gained and/or updated during processing according to the selectivities actually detected.

Execution skew caused by query-external workloads can manifest itself either on disks or on processors. In the latter case, tasks may be dynamically migrated between PNs; this can take place either for a single partition or for an entire run or set of runs with $\lfloor l \rfloor$ scans executing in parallel. The first option seems fairly complicated because it breaks up runs and possibly involves additional processing nodes. The second way, however, may fail under fluctuating processor load that makes it impossible to keep up complete runs on any node. The best strategy might be to allow each node to suspend some scans in situations of heavy load and to resume them later on (in fact, run sizes may be chosen differently for the various processors from the start), thus dynamically varying the run size; this approach would also reduce the message overhead between a node and its coordinator. When a scan on a certain partition lags because of (inter-query) disk contention, the best idea is also to suspend this scan until disk load decreases. The corresponding PN can work on a different partition in the meantime. This can also be handled locally to reduce the scheduling overhead.

Finally, all these partial solution can be incorporated into an overall runtime scheduling scheme such as the "on-demand" strategy from [Mä98]. It will assign only the minimum number of tasks to a node at a time – in this case, usually $\lfloor l \rfloor$ concurrent scans. When a node has finished one of its jobs, it requests another one from its coordinator until all the work is done. This method can be enhanced by giving each node a short queue of additional tasks (possibly two or three) to avoid idle times during scheduling and to enable some of the local measures previously described to be taken.

We will evaluate some of these policies experimentally in the near future. Note that all the approaches we have suggested (with or without skew management) are *isolated* strategies in the sense that there is no inter-query coordination taking place apart from MPL management as discussed in section 2.1. We will investigate *integrated* schemes in future studies, e. g. [Mü98].

3.2 Clustered index scan

Index scans differ substantially from relation scans because the index makes it possible to restrict the scan to a subset of the pages, potentially saving a lot of work. On the other hand, there is the additional cost of index access itself. Before discussing the most general case of index scans in the next section, we will consider the special class of clustered indices.

In contrast to what the term suggests, it is not the index that is supposed to be clustered, but the relation itself (although for any kind of index, the leaf pages may be stored sequentially on disk). When a clustered index is present, the tuples are assumed to be sorted on the corresponding attribute, and the data pages are stored consecutively on disk[9]. The query optimizer can take advantage of this order on the data by

- limiting the scan to a subset of pages (and possibly of partitions), and
- reading consecutive pages from a disk is a single operation (continuous read).

The only way of (logically) partitioning the data across multiple disks in the presence of a clustered index is by range partitioning on the indexed attribute[10]. Any other partitioning function (e. g. hash-based, random, round robin) would destroy the order of pages and thus the cluster-

9. Note that the order of pages may be destroyed by the underlying file system or even by some lower level functions of the DBMS itself in a physical declustering (cf. section 1.2). Since this is transparent to the query optimizer, however, we have to assume that order is preserved and that it can be exploited e. g. for continuous reads. Otherwise, the query must be processed as a non-clustered index scan.

ing. Note that this assumes a partition to be a contiguous range of tuples. If there is more than one fragment in a partition, the same argument applies to fragments instead, and range declustering is still the only viable option.
The special properties of a clustered index can be exploited best for *range queries*, i. e. queries that select tuples whose value on a certain attribute is within a specified range. Thus, in the remainder of this section, we will make the additional assumption that the system has to process a range query on an attribute that has a clustered index; other selection predicates should be handled as if the index were non-clustered (cf. section 3.3).
Having restricted the problem so strongly, we can now largely reduce it to the relation scan problem, only pertaining to a limited range of pages. From the declustering scheme (which must be known to the optimizer) we can immediately derive the set of partitions that contain relevant tuples, where all but the first and last fragments (in the order induced by the indexed attribute) will have to be scanned in their entirety. For those *bounding fragments*, the first and last pages to access can be found with a single index lookup each, which can be done either immediately by the query coordinator or later by the node that is assigned the respective partition. (The first option seems advantageous because it gives additional information on the amount of pages to be scanned which can be used in scheduling.)
The result is a list of (parts of) partitions to be scanned. Like with a relation scan, each partition is a set of consecutive pages, so that continuous reads can be applied. Based on this similarity, the entire process of scheduling and actual processing can be handled in the same way as for a relation scan (see section 3.1). Note, however, that even if the original declustering of the relation is balanced across the disks, the areas to be scanned can vary in size because the bounding fragments are limited according to the selection predicate. This should be reflected in the scheduling algorithm (cf. section 3.1.1). Note also that a clustered index scan with a low selectivity will often access just one partition of the relation concerned, which will lead to sequential processing even if there is a significant amount of data to be read from this partition. This is in tune with the results of [RM95], which show that index scans of low selectivity should be handled with a small degree of parallelism.

3.3 Non-clustered index scan

A non-clustered index scan occurs either when only a non-clustered index is available or when a clustered one cannot be used effectively because the query's selection predicate is not range-based. In the latter case, the index is used similarly to a non-clustered one. This is the reason for us to include this case here instead of in the previous section. The differences that do exist will be discussed in the relevant passages.
The main problem with non-clustered indices is that no particular order can be assumed on the data, neither between nor within pages. Furthermore, it is not known in advance (i. e. before the index has been evaluated) which partitions contain relevant tuples, and how many. Bearing this in mind, we discern two basic ways of parallelizing a non-clustered index scan:

- Analyze the index first, then distribute a list of data pages to be scanned to the participating nodes; or
- Partition the work based on data or index fragments, then let each node evaluate its own part of the index and data.

The first approach has the obvious drawback of the index being evaluated separately from the relation itself, usually by a single node, which is bound to increase response time. The second scheme, on the other hand, entails the risk of disk contention because, in general, several PNs will have to access the same data partitions. In addition, it cannot be ensured that the various

10. Note that the opposite is not true: Range partitioning can very well be applied with a non-clustered index or without any index at all; the order of tuples within each partition is then arbitrary.

nodes will process the same amount of data, so that load imbalances may occur. For both methods, the degree of parallelism has to be determined before the parallel part of the work can begin. This is easier with the first approach, where the number of pages to be processed in known after the index analysis.
In the following, we will discuss the two methods of non-clustered index scan processing.

3.3.1 Separate evaluation of index and data

In our first approach, scan processing is divided into two stages: In the first stage, the index is analyzed, and pointers to the relevant data pages are derived from the tuple IDs stored in the index leaves. Though this stage will usually be executed on a single node, it may be worthwhile to parallelize it, especially if the selection predicate is

- range-based (which implies that the index is non-clustered, otherwise section 3.2 applies);
- non-invertible (meaning that it is not possible to derive analytically the set of attribute values satisfying it and to look them up directly in the index); or
- invertible but with a large set of solutions.

In these cases, a large portion or even all of the index leaf pages have to be scanned to find all relevant references, so that parallel processing is useful to reduce response time.
The second stage of the scan will evaluate the data pages of the relation itself based on the list of pointers generated previously. Unless the list is very short, this stage will also be parallelized, and it can be processed analogously to a clustered index or relation scan: The page references are grouped according to the partitions they belong to, and partitions are assigned to PNs in the same way as discussed in the previous sections, including the computation of the degree of parallelism. In fact, the same scheme can be applied to the first stage with respect to the declustering of the index. Note, however, that the number of relevant pages per partition can now vary randomly, so that explicit skew handling is advisable.
This approach of separately evaluating index and data pages has several advantages:

- With the set of relevant data pages known after the first stage, load can be balanced very effectively in the second stage.
- Each of the stages can be processed without (intra-query) disk contention.
- Data pages containing multiple “hits” must be accessed only once because duplicates will be eliminated from the page list in advance.
- When consecutive data pages must be processed, they can be read continuously.
- Scheduling routines for relation or clustered index scans can be reused.

The disadvantages stem from the overhead of separate index analysis:

- The maximum disk bandwidth may not be exploited all the time, especially when
  a) the index is processed sequentially;
  b) the index has a low degree of declustering; or
  c) index and data are stored on different disks.
  (This is less of a problem in multi-user mode, where we can expect other transactions to use the disks up to their bandwidth limits.)
- Storing, updating, sorting, and evaluating the list of page references can be costly and is difficult to parallelize.
- Result tuples cannot be delivered (to the user or to a higher query operator) until the second stage of the scan has been started.

All of these effects will increase response time.

3.3.2 Interleaved evaluation of index and data

The possible overhead from index analysis found in the previous approach inspires an alternative idea, namely to interleave index and data access, both exploiting the maximum disk read bandwidth and saving the processing cost for the page list. The basic concept is simple, and it works well for sequential processing; in fact, it is the standard method for single-node systems: The outer loop of the algorithm analyzes the index, either scanning the leaf pages or looking up individual values according to the selection predicate. Whenever a relevant reference (or set of references) is found, the corresponding pages are read and processed immediately in an inner loop.

It is apparent that this scheme avoids the shortcomings of the two-staged algorithm: All disks can be used simultaneously, results can be delivered immediately, and no reference list must be kept. However, it also has some drawbacks:

- Unless the index is clustered, consecutive attribute values will not be found on adjacent data pages. Indeed, identical values can be distributed across arbitrary pages or even partitions, depending on the declustering scheme. Thus, read-ahead cannot be applied (but prefetching can, although with questionable success). Even with a clustered index, the disk read head may be forced to oscillate between index and data pages if they are stored on the same disks. In either case, continuous reading rarely takes place.
- With the scan jumping to and fro within the data partitions, the same pages may well have to be accessed several times. If they have been forced from the buffer by then, this induces multiple reads from disk.
- In order to make efficient use of the disk bandwidth available, the algorithm must go on scanning the index while waiting for data pages requested from the disks. (Otherwise, only one disk at a time will be accessed, and even this one will be unused while the PN is processing the data received in the previous operation.[11]) This requires a management of "pending" page references that is comparable in complexity to the list kept in the two-stage scheme above.

While these effects may be considered tolerable, further trouble awaits when this method is to be parallelized. The problem here is that the declustering of the index on the one hand and of the actual data on the other hand are often *incompatible*, which leads to disk contention within the current query. Here, we consider their declusterings *compatible* if for each data partition there is exactly one index partition holding the same values of the attribute in question. (This assumes that the data is declustered at least as broadly as the index.) A special case of this is present when corresponding partitions of both index and data reside on the same disk. The concept of clustering compatibility is illustrated in figure 3.

In general (i. e. other than by accident), compatible declustering is possible only in the case of range partitioning. If the partition boundaries correspond, the scan job can be naturally partitioned according to the declustering of the index. With each index partition exclusively pointing to one or more data partitions, disjunct sub-jobs can be generated and intra-query disk contention is avoided.[12]

With incompatible declustering, however, one has to decide whether to partition the scan according to index or data declustering. In the first case, index access is split disjunctly among the PNs, but data access can cause massive disk contention. In the second case, each data page is read by only one processor, while index pages are not only ill-partitioned but may even have to be scanned by every single node so that each can locate its relevant data. In short, the choice is between

- disjunct access to the index and contention on the data, or
- contention (plus multiple reads) on the index and disjunct reads on the data.

11. As above, this does not necessarily leave disks idle in multi-user mode. Still, the query will benefit from an overlapping of I/O and processing.
12. If the boundaries are not exactly the same but reasonably similar, the same scheme can be employed; in this case, intra-query disk contention will still be low.

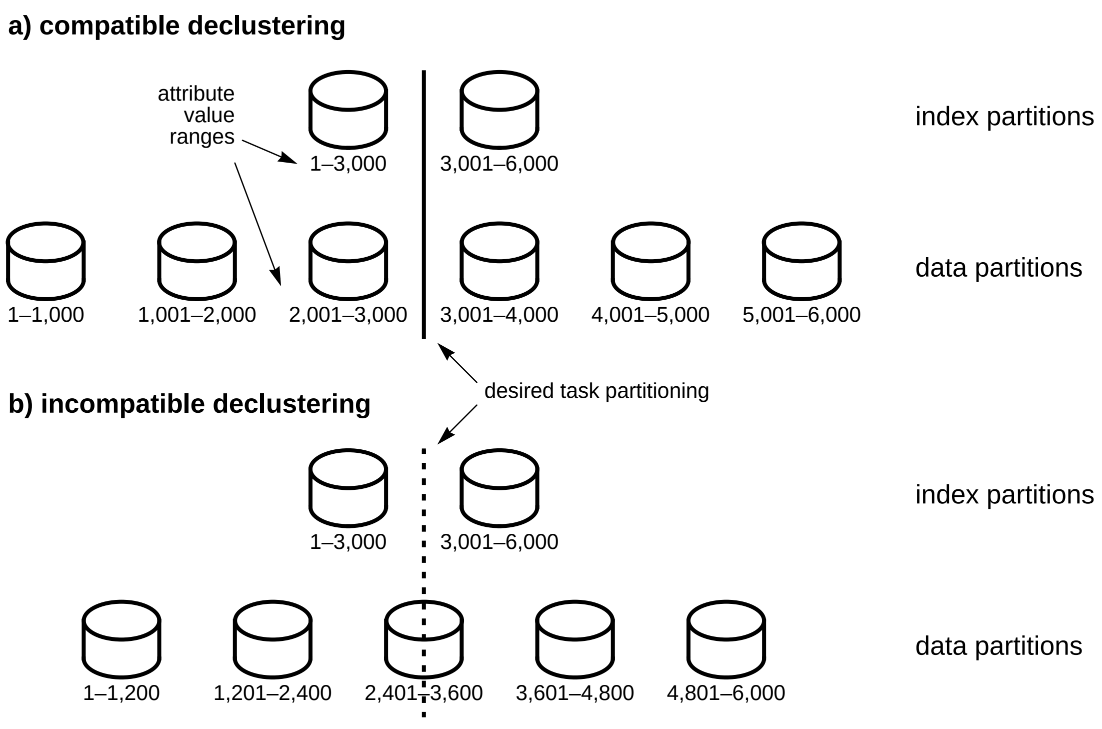


Figure 3: Compatible and incompatible declustering

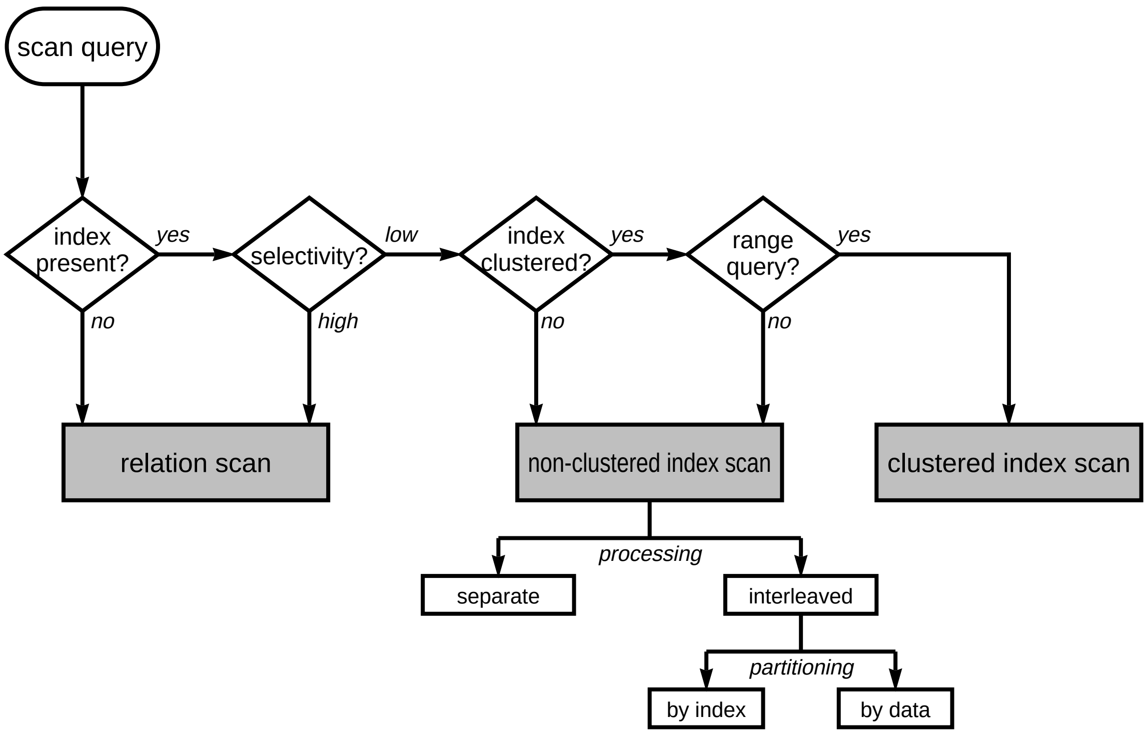
Intuition suggests that index contention is the lesser of the two evils because the index is much smaller. In addition, "partitioning by data" has a higher buffer hit rate[13] and allows for further optimizations, such as first collecting a list of page references as in the scheme from section 3.3.1 (but applied on single nodes).

Finding the best degree of parallelism is more problematic for interleaved than for separate processing of index and data. When declusterings are compatible, the DP can be determined similarly to a relation scan. However, when computing the number of disks to be read simultaneously by each node, one should take into consideration that each index partition goes together with a fixed amount of data partitions and that index and data pages may or may not be accessed at the same time, depending on the algorithm used.

For "partitioning by index", similar considerations apply, but disk contention for data access must be taken into account. This can be accomplished by regarding the aggregate bandwidth of all the disks concerned, reducing the value according to the contention expected. The DP can then be chosen so as to exploit this bandwidth; alternatively, it can be kept lower, making allowances for other queries in the system.

"Partitioning by data" allows for a scheduling scheme based on the size of data partitions only, because index access will cost approximately the same for all PNs. (However, it is a good idea to choose a different order of index analysis for each node so as to avoid hot spots on disk.) Again, the basic method can be copied from the relation scan and modified to include index access.

13. The hit rates for data pages improve when only a subset of them can be accessed by each node and when pages are needed more than once. This rarely applies to the index because for non-invertible or range-based selections (which would be the ones to parallelize) index leaf pages are usually scanned sequentially.



Based on the type of index (if any) and the properties of the selection predicate, the query optimizer decides on the scan mode to apply (full, clustered, non-clustered). A non-clustered index scan can be processed in several different ways for the choice of which there are no clear criteria yet. This diagram assumes that the selection refers to a single attribute, the index of which is used if present.

Figure 4: Decision on the scan mode

In all three cases, however, interleaved processing is susceptible to skew. It is very hard to predict how many pages of each data partition have to be processed, and how many result tuples will be generated from them. While as much knowledge as available should be used in advance, robust algorithms require some sort of dynamic skew management (cf. section 3.1.1).

Two more things should be mentioned about index scans in general, both clustered and non-clustered: First, all efforts of parallelization can be abandoned when the selectivity of the query is known to be very low, e. g. for exact-match queries on a primary key. Such transactions are most naturally processed on a single node and can be viewed as special cases of either the interleaved or the separate approach. Second, as indicated earlier, both clustered and interleaved non-clustered index scans can be combined with separate processing applied on each participating node.

3.4 Summary

Figure 4 presents an overview of the conditions under which the various scan modes should be applied. When there is no index, or when the query has a high selectivity that requires most data pages to be accessed, a full relation scan should be performed. Otherwise, only a range query in the presence of a clustered index qualifies for a clustered index scan. All other cases should be handled as non-clustered index scans.

The latter can be treated in different ways characterized by the order of index and data access as well as by the partitioning of disk accesses among the processing nodes. Criteria for these decisions are yet to be found; so far, the only clear case is that of compatible declustering (cf. section 3.3.2), in which partitioning by index and by data coincide.

4 Affinity-based disk scheduling

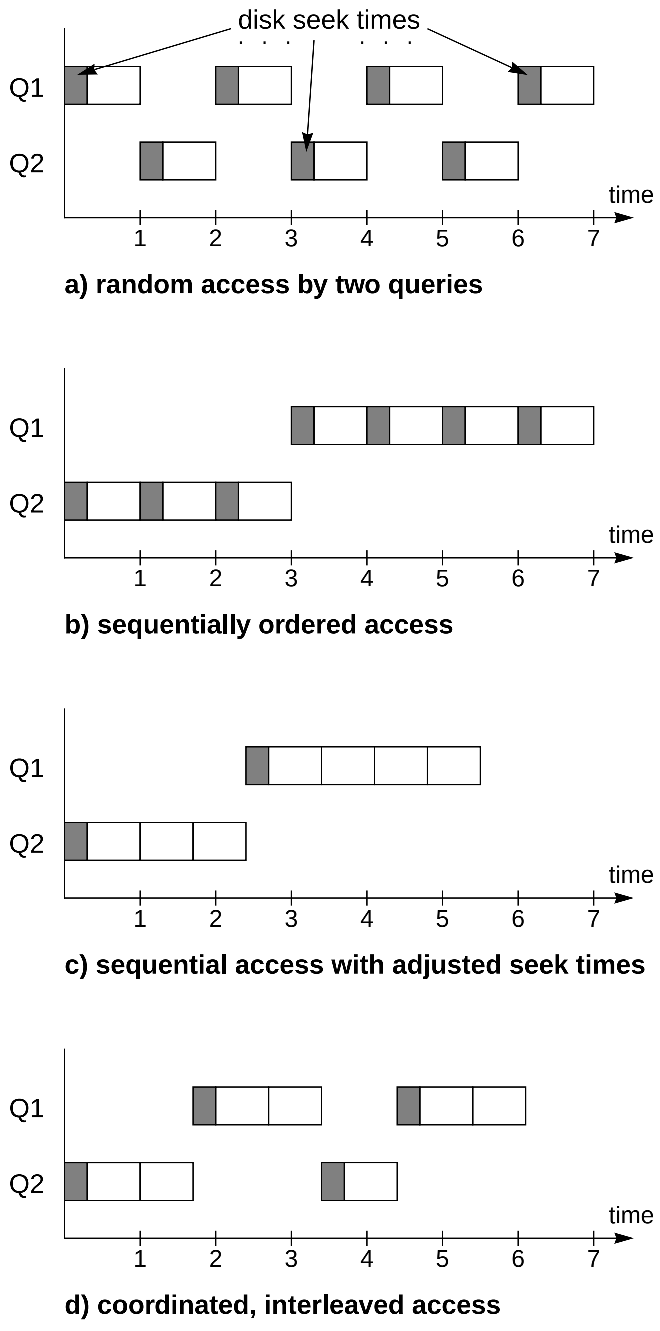

disk seek times
Q1
Q2
time
1
2
3
4
5
6
7
a) random access by two queries
b) sequentially ordered access
c) sequential access with adjusted seek times
d) coordinated, interleaved access

Figure 5: Coordination of disk access

In this chapter, we present a disk scheduling policy designed to increase the share of continuous read operations. It is based on the idea of data affinity in the sense that for each disk, one particular processing node is responsible for coordinating access to it; we thus call this node the *access coordinator* (AC). In contrast to the original data affinity approach (cf. section 2.2.2), however, the AC need not necessarily process its data itself. Instead, other PNs may access the disk after getting permission from the AC, and load balancing can still be applied without restriction. The job of the AC is to ensure an order of access that exploits continuous reads. The benefits of this principle are illustrated in figure 5.

Example. Two queries, Q1 and Q2, are started at time unit zero and compete for access to a certain disk. Without any coordination, they frequently interrupt each other, causing a disk seek overhead every time. Since both queries require about the same amount of data and the chunks read in each step are idealized as equal-sized, they finish roughly at the same time – after seven or six time units, respectively (**a**). A mere (hypothetical) reordering of access will cause Q2 to finish in half the time while Q1 is unaffected performancewise (**b**).^{14} However, assuming that both queries have to read contiguous pages of data, disk seek times will be reduced and both queries will be accelerated, finishing at time units 5.5 and 2.4, respectively (**c**).
Our choice of the shorter query, Q2, to be processed first was not accidental. While in this order, the queries finish at time units seven and three (**b**), the reverse choice would have resulted in finishing times seven and four, respectively (with Q2 lasting longer). More generally, the preferred query will finish in minimum time, whereas the delayed one will require

the sum of both processing times[15]. Thus, the global minimum of response times is achieved by processing the shorter transaction first. The same reasoning applies to the case of adjusted seek times (**c**) and of more than two queries at a time. We realize that this raises questions of fairness because queries are accelerated by different factors. However, since the slowest query (whichever it is) cannot finish any sooner, fairness could be achieved only by artificially inhibiting the faster transaction, which we consider nonsensical.
Note that – as any strategy enforcing continuous reads – this method requires sufficient buffer space to hold the corresponding number of pages and/or enough CPU capacity to process the data as fast as the disk delivers it. If either is scarce, disk access may be scheduled in smaller portions that are still greater than those achieved by concurrent reads, striking a balance between optimized reading and disk seek overhead (**d**). ❑

Simple analytical model. Let *s* be the seek time of the disk drive under consideration; let *p* be the transfer time for a single page. Let *n* and *n'* be the number of consecutive page requests serviced at a time (*read granule*) without or with ABDS, respectively. If *m* is the message overhead associated with ABDS and *t* denotes the total number of pages to be read in a query, we can estimate the sum of disk access times for the entire query as

$$a = \frac{t}{n} \cdot (s + np) = \frac{t}{n} \cdot s + tp$$

without ABDS and

$$a' = \frac{t}{n'} \cdot (s + m + n'p) = \frac{t}{n'} \cdot (s + m) + tp$$

with ABDS. Apparently, *tp* is a constant that reflects a lower bound of access time (all pages must be transferred). The remaining component of the second formula approaches $s + m$ (a single seek operation + one scheduling action) for growing *n'*. It would approach *s* in the first equation, but read granules will tend to be small in multi-user mode.
It is easily shown that *a* corresponds to *a'* if the granule ratio $r := n'/n$ equals $1 + m/s$. For example, if disk seek time and message overhead are similar in magnitude, ABDS begins to pay off as soon as the granule is doubled, which is a modest assumption. For more realistic values – e. g. $s = 8$ ms, $p = 1$ ms, $m = 1$ ms, $t = 100$, $n = 2$, and $n' = 10$ – we arrive at accumulate access times of 500 ms and 190 ms, respectively. This corresponds to a reduction by 62 %. ❑

In general, the AC must either execute the respective transactions itself or be asked for permission by any other PN that should do so. In practice, the latter possibility will be preferred for the benefit of load balancing. Unfortunately, this will introduce a large amount of additional communication if all transactions have to take part in the scheduling scheme. But as the example has shown, the shorter of any two conflicting queries should be prioritized in disk access. As a consequence, we can immediately exclude “short” transactions from scheduling and allow them to interrupt any longer operation that may be running on the disk. The definition of “short”, however, requires closer reflection: While it would originally include only those transactions that can be guaranteed to be shorter than all others (i. e. accessing just one page), we should extend it to encompass all those queries that are “not worthwhile” scheduling. This requires an experimental measurement of profits and overheads effected by our scheme, and we will dedicate a future report to it.
Note that the problem of disk access scheduling cannot be satisfactorily solved by simply using large caches and sophisticated buffer management strategies. These are unable to use the knowl-

14. Strictly speaking, Q1 suffers from late delivery of result tuples that may affect subsequent query stages. However, we consider this a minor problem compared to the disk contention in the original scheme (**a**). This is particularly true if the complete query involves blocking operators.
15. Here, we understand *processing time* to mean the “net” time of processing without the initial queuing delay, whereas *response time* includes both (cf. section 1.2).

edge of future data requirements that can only be derived from the queries themselves, so they have to fall back on heuristics designed for some “average” case. Besides, they will still create concurrent disk requests because their policies are necessarily restricted to single PNs. Even smart disk controllers with priority-based scheduling can only work with requests already received; they cannot consider “intensions” of future processing that a transaction may have.

Finally, it should be mentioned that affinity-based disk scheduling combines very well with the original concept of data affinity. When queries on the same data are actually processed by the same node, network load due to disk scheduling is reduced; at the same time, more small queries can be included in the scheme without message overhead. Note, though, that this requires the regions (cf. section 2.2.2) of data to coincide with whole disks for the integration to work best.

5 Conclusions

In this report, we have explored the problem space of scan processing in shared-disk database systems. We have identified the most important parameters and the trade-offs between them, as well as some rules of thumb for the essential decisions. Naturally, we have raised more questions than we provided answers, and there is still a lot of work to be done. Thus, in conclusion, we will briefly review the particular areas that we consider fruitful for further research:

- *Degree of parallelism and multiprogramming level:*
  Although these are very basic parameters, there are not yet any clear criteria of choosing DPs and MPLs, especially when different classes of (possibly prioritized) queries have to be distinguished. Specifically, their sensitivity to high system load is yet to be fathomed. Some of our very first efforts will be devoted to this field.
- *Scan modes:*
  Although we have given some common-sense rules for deciding on the basic mode of processing (cf. figure 4), such parameters as the selectivity threshold for the choice between index and relation scans have not yet been determined. More importantly, the non-clustered index scan offers several alternative ways of processing that must be evaluated. Apparently, this point is strongly related to the previous one.
- *Query scheduling:*
  We are especially interested in a performance comparison between predictive, reactive, and on-demand scheduling strategies as previously performed for join queries [Mä98]. This will be done for all three scan modes.
- *Disk scheduling:*
  Next, we will investigate different strategies of disk access scheduling, including the affinity-based approach introduced in chapter 4. In addition to the definition of “small” transactions to be excluded from the algorithm, ways of responding to system load will be of particular interest.
- *Skew handling:*
  As laid out in section 2.6, skew effects are a ubiquitous problem faced on every level of query processing. As a consequence, we will treat skew not as a separate field of research but rather as a boundary condition that is orthogonal to all the above considerations.

Since shared-disk architectures are poorly represented in current database research, most of these problems are yet unsolved. Thus, there is room for significant advances from future studies, and we are confident to be able to demonstrate the virtues of SD systems for very large database applications.

References

[CG94] R. G. Cole, G. Graefe: *Optimization of Dynamic Query Evaluation Plans*. Proc. ACM SIGMOD Conf., Minneapolis, 1994.

[CK93] T. H. Cormen, D. Kotz: *Integrating Theory and Practice in Parallel File Systems*. Proc. DAGS Symposium on Parallel I/O and Databases, Hanover, 1993.

[DG92] D. J. DeWitt, J. Gray: *Parallel Database Systems: The Future of High Performance Database Processing*. Comm. ACM 35 (6), 1992.

[Gr93] G. Graefe: *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys 25 (2), 1993.

[IC91] Y. E. Ioannidis, S. Christodoulakis: *On the Propagation of Errors in the Size of Join Results*. Proc. ACM SIGMOD Conf., Denver, 1991.

[Mä98] H. Märtens: *Skew-Insensitive Join Processing in Shared-Disk Database Systems*. Proc. Issues and Applications of Database Technology, Berlin, 1998.

[Mä98a] H. Märtens: *Disk Scheduling for Intermediate Results of Large Join Queries in Shared-Disk Parallel Database Systems*. Technical report, University of Leipzig, to appear.

[MD93] M. Mehta, D. J. DeWitt: *Dynamic Memory Allocation for Multiple-Query Workloads*. Proc. 19th VLDB Conf., Dublin, 1993.

[MD95] M. Mehta, D. J. DeWitt: *Managing Intra-operator Parallelism in Parallel Database Systems*. Proc. 21st VLDB Conf., Zürich, 1995.

[Mü98] M. Müller: *Simulative Bewertung von Verarbeitungsstrategien für Mehr-Wege-Joins in parallelen Shared-Disk-Datenbanksystemen*. Master's thesis, University of Leipzig, in preparation.

[MVW98] Y. Matias, J. S. Vitter, M. Wang: *Wavelet-Based Histograms for Selectivity Estimation*. Proc. ACM SIGMOD Conf., Seattle, 1998.

[NZT96] M. G. Norman, T. Zurek, P. Thanisch: *Much Ado About Shared-Nothing*. ACM SIGMOD Records 25 (3), 1996.

[PGK88] D. A. Patterson, G. Gibson, R. H. Katz: *A Case for Redundant Arrays of Inexpensive Disks (RAID)*. Proc. ACM SIGMOD Conf., Chicago, 1988.

[PI96] V. Poosala, Y. E. Ioannidis: *Estimation of Query-Result Distribution and its Application in Parallel-Join Load Balancing*. Proc. 22nd VLDB Conf., Mumbai, 1996.

[PI97] V. Poosala, Y. E. Ioannidis: *Selectivity Estimation Without the Attribute Value Independence Assumption*. Proc. 23rd VLDB Conf., Athens, 1997.

[Ra93] E. Rahm: *Empirical performance evaluation of concurrency and coherency control protocols for database sharing systems*. ACM Transactions on Database Systems 18 (2), 1993.

[Ra94] E. Rahm: *Mehrrechner-Datenbanksysteme*. Addison-Wesley, Bonn, 1994.

[Ra96] E. Rahm: *Dynamic Load Balancing in Parallel Database Systems*. Proc. Euro-Par Conf., Lyon, 1996.

[RM95] E. Rahm, R. Marek: *Dynamic Multi-Resource Load Balancing in Parallel Database Systems*. Proc. 21st VLDB Conf., Zürich, 1995.

[RS95] E. Rahm, T. Stöhr: *Analysis of Parallel Scan Processing in Shared Disk Database Systems*. Proc. Euro-Par '95 Conf., Stockholm, 1995.

[SN92] S. Seshadri, J. F. Naughton: *Sampling Issues in Parallel Database Systems*. Proc. 3rd EDBT Conf., Vienna, 1992.

[SWZ96] P. Scheuermann, G. Weikum, P. Zabback: *Data Partitioning and Load Balancing in Parallel Disk Systems*. Technical Report A/02/96, University of the Saarland, Saarbrücken, 1996.

[Va93] P. Valduriez: *Parallel Database Systems: the case for shared-something*. Proc. 9th ICDE Conf., Vienna, 1993.

[WDJ91] C. B. Walton, A. G. Dale, R. M. Jenevein: *A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins*. Proc. 17th VLDB Conf., Barcelona, 1991.