UNIVERSITÄT LEIPZIG

Institut für Informatik

Disk Scheduling for Intermediate Results of Large Join Queries in Shared-Disk Parallel Database Systems

Holger Märtens

Report Nr. 9 (1998)

ISSN 1430-3701

Disk Scheduling for Intermediate Results of Large Join Queries in Shared-Disk Parallel Database Systems

Holger Märtens

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

July 1998

Abstract:
In shared-disk database systems, disk access has to be scheduled properly to avoid unnecessary contention between processors. The first part of this report studies the allocation of intermediate results of join queries (buckets) on disk and derives heuristics to determine the number of processing nodes and disks to employ. Using an analytical model, we show that declustering should be applied even for single buckets to ensure optimal performance.
In the second part, we consider the order of reading the buckets and demonstrate the necessity of highly dynamic load balancing to prevent excessive disk contention, especially under skew conditions.

Keywords:
parallel databases, shared-disk systems, hash join, load balancing

1 Introduction

Shared-disk (SD) database systems (DBS) offer a great load balancing potential due to the fact that every processing node (PN) can access any disk at uniform cost. In this report, we will present some ideas on how to exploit this potential in the context of large join queries, where *large* means that the amount of data to be processed does not fit into the aggregate memory of the system's processing nodes. Specifically, we will consider different ways of distributing intermediate results across the disks involved and demonstrate that the degree of declustering, in particular, has to be chosen with care. In addition, we study the order of disk access while reading the data back in for the join itself and conclude that highly dynamic scheduling is required to control disk contention caused by execution skew.

1.1 Preliminary considerations

For the purposes of this report, we will presume as given:

- the number and performance of PNs and disks in the system;
- the size and allocation of the base relations involved in the join;
- the selectivity of the scan and join operators.

We will further assume the following simplified conditions:

- two-way equi-joins;
- even, but not necessarily full declustering of the base relations;
- homogeneous PNs and disks;
- no pipelining between the scan and join operators.[1]

For ease of presentation, we will first consider only one of the relations involved in the join; an extension to both relations will follow. Similarly, most of our thoughts will be developed for single-user mode first and extended to multi-user mode later on.

Let us assume the following scenario: Base relation R is declustered across r disks. From these it is read by n scan nodes, which distribute their output to d disks. These intermediate results are read and processed by m join nodes. Following the notation of [Ra93], each scan node produces an output R_i $(i = 1 \ldots n)$; each join node expects an input R_j $(j = 1 \ldots m)$. The data transfer is assumed to take place by hashing so that, in general, each scan node must deliver to each join node a relation fragment R_{ij} $(i = 1 \ldots n,\ j = 1 \ldots m)$. However, since we are considering large queries, the join nodes will not be able to hold their entire input. In order to avoid the multiple reads required for further fragmentation in a *Grace* or *hybrid hash join* [SD89], it is appropriate to partition the data into a greater number of fragments (then called *buckets*) that do fit into a PN's memory.

Since the number of buckets, b, can grow very large (especially in the case of skew, as discussed later), it may well exceed both the number of nodes and the number of disks present in the system. As a consequence, disk contention between processors will occur, and the buckets must be stored in such a way as to minimize this contention. For this purpose, we introduce the additional parameter v, denoting the degree of declustering for every single bucket. Altogether, we have five parameters to determine for a simple two-way join query, as shown in table 1. (The declustering of the base relations is assumed to be fixed.)

1.2 Processing model

We assume a two-way join query to be processed in the following simplified manner:

1. Pipelining is possible only if at least one relation can be kept in main memory.

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

Table 1: Tuning parameters for join processing

n	number of scan processors
m	number of join processors
d	number of disks to store intermediate results
b	number of buckets for intermediate results
v	degree of declustering for buckets

1. In the scan phase, the n scan processors read relation R, producing $b \cdot n$ *scan fragments*. (Each node contributes to each bucket.)
2. Since the buckets are to be declustered with a degree of v, sets of n/v *scan fragments* (for the same bucket but from different nodes) are integrated into $b \cdot v$ *bucket fragments* and stored on disk. If the degree of bucket declustering exceeds the number of scan nodes ($v > n$), this integration is really a further partitioning. This case, however, is inefficient, as shown below.
3. In the join phase, the m join nodes process the buckets one by one. Although different PNs work on different buckets, each node handles just one bucket at a time. A set of buckets processed by the same join node is called a *run*.

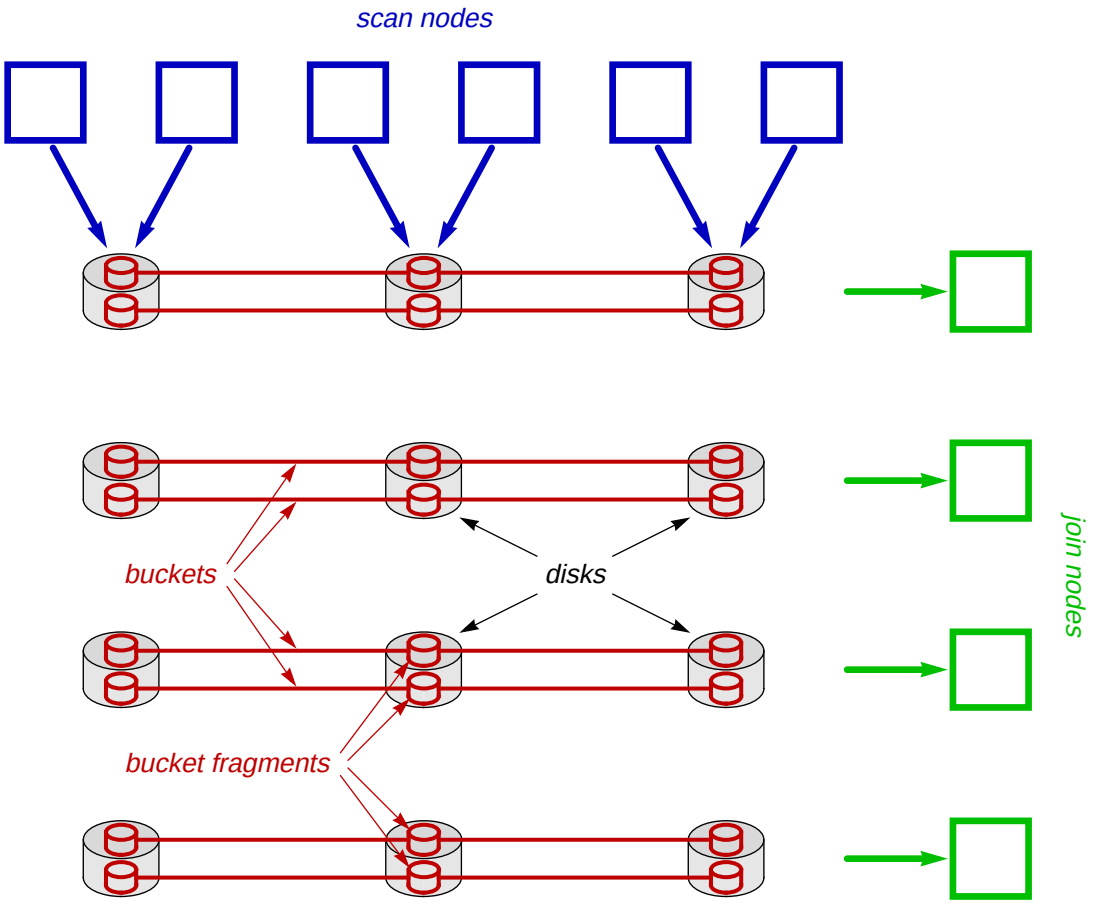
Table 2 lists some more characteristic values derived from the parameters in table 1, which will be important in the remainder of this report. Figure 1 shows an example with some typical settings and demonstrates an assignment of disks to PNs with minimal access conflicts. It also illustrates the calculation of the corresponding coefficients.

Table 2: Characteristics of join processing

scan fragments (sf)	$b \cdot n$	fragments produced by the scan nodes
bucket fragments (bf)	$b \cdot v$	fragments stored on disk
degree of unification (du)	n/v	scan fragments united into a bucket fragment
write contention (wc)	$max(n/v, 1)$	scan nodes writing to each disk
write parallelism (wp)	$max(d/v, d/n)$	disks written to by each scan node
run length (rl)	b/m	buckets to be processed by each join node
bucket parallelism (bp)	$m \cdot v$	bucket fragments processed at the same time
read contention (rc)	$max(m \cdot v/d, 1)$	join nodes reading from each disk at a time
read parallelism (rp)	v	disks read from by each join node at a time

2 Allocation of buckets

In this section, we will provide rough estimates of the five parameters listed in table 1. As in most other studies, we will assume integer proportions between some of those values to simplify our considerations. This concerns the ratios of n to d, m to d, v to d, v to n, and b to m. We will further presume d to be greater than both n and m, but we make no such statement about v. In reality, the parameters can either be chosen so as to fulfill these conditions, or they can be ap-



In this example, eight buckets are processed using six scan nodes and four join nodes. The buckets are declustered across twelve disks with a degree of three. To minimize access conflicts, each disk is used by just two scan nodes and one join node. The parameters from tables 1 and 2 are set as follows:

$n = 6$	$m = 4$	$d = 12$	$b = 8$	$v = 3$
$sf = 48$	$bf = 24$	$du = 2$	$wc = 2$	$wp = 4$
$rl = 2$	$bp = 12$	$rc = 1$	$rp = 3$	

This example corresponds to a read-optimal setting (cf. section 2.5.4).

Figure 1: Example for processing model

proximated in a straightforward way. When skew effects are involved, the assumption of integer proportions may become obsolete. The following calculations are based on bandwidth considerations similar to those presented by Mehta and DeWitt [MD95], but designed for the storage of intermediate results on disk.

2.1 Number of scan nodes (n)

Given the degree of declustering r of base relation R and the performance of the system's disks, we can easily determine the available read bandwidth. In single-user mode, n can then be se-

lected based on the nodes' processing power such that the disk bandwidth is optimally exploited. In multi-user mode, the same calculation applies; however, the coefficients of performance applied for both PNs and disks should be adapted to reflect the current system load. In general, this will lead to lower values of *n* because disks tend to have a higher utilization index than processors. (See [Mä98a] for details on scan processing.)

2.2 Number of disks (*d*)

In the reverse process, *d* can be determined from the disk bandwidth required to take up all the output produced by the scan, which can in turn be computed based on the performance indices mentioned. Since the output rate depends on the selectivity *sel* of the scan, and since *d* should be determined before the query is started[2], *sel* must be estimated in advance. If there are no statistics to derive such estimates from, a sampling phase prior to the scan itself may be useful. In general, *d* will then be determined to be about $sel \cdot r$ (assuming that the access times for read and write operations are approximately equal).

This estimate, however, should be viewed as a lower bound of the true value of *d*. This is because a greater number of disks will enable more join nodes to read the buckets from them, so that *m* can be selected higher in the next step. This may be advisable if *r* is very low (meaning that *R* is declustered inappropriately for large queries, which should be rectified for the intermediate results).

In single-user mode, *d* may simply be set to the total number of disks in the system because there are no conflicting queries in the system. There may, however, be contention between the scan on *R* and the writing of the buckets. This can be amended either by using disjunct sets of disks or by allocating large buffers for both operations. In multi-user mode, on the other hand, conflicts with other queries will occur, the magnitude of which depends on the system load. This aspect certainly requires closer examination. For the time being, we consider the above estimate of *d* a good common-sense heuristic.

2.3 Number of join nodes (*m*)

From the number of disks, *d*, we can again calculate the available read bandwidth as above; then, *m* can be determined so as to make use of this bandwidth. As we apply the same performance indices, we can expect the ratio of *m* and *d* to match that of *n* and *r* (except for possible discretization errors). If *m* is computed after the scan has finished, the load situation may have changed and a different ratio may be achieved.

2.4 Number of buckets (*b*)

The number of buckets should be at least large enough to make all buckets fit into a single node's memory. Leaving skew effects aside, this leads to a rule-of-thumb value of

$$b = \frac{sel \cdot |R|}{\text{memory per node}}.$$

However, such large buckets may still flood the buffers of the PNs and come into conflict with other queries in multi-user mode; *b* should be selected higher to prevent this. On the other hand, it is generally useful to keep the number of buckets as small as possible so as to avoid a large amount of disk seek times for both writing (during the scan) and reading (during the join) the

2. One might also develop an adaptive algorithm that determines *d* at runtime, possibly starting with a low value and increasing it when a high selectivity is detected. It would then have to schedule further operation such that the sizes of "old" and "new" fragments are equalized. However, the overhead involved in this is unclear, and we will not pursue this idea any further.

buckets. As a consequence, we suggest to first determine the amount of memory that may be set aside for a local join and to apply the above formula to that value.
When skew is present, buckets cannot be divided so evenly. In this case, *b* should be set high enough to guarantee that the memory limit is met by the largest bucket. Consider the following pessimistic approximation that we used in [Mä98]: If relation *R* has *a* different values on the join attribute and the hash function employed can be assumed to be fair, each bucket will contain about a/b different attribute values (for *b* yet to be chosen). In the worst case, one bucket will acquire the $\lfloor a/b \rfloor$ most frequent values and contain

$$\sum_{i=1}^{\lfloor a/b \rfloor} f(\alpha_i)$$

tuples. (Here, α_i is the *i*-th join attribute value in the order of frequency, and $f(\alpha)$ is the frequency of α.) Then, *b* should be selected such that this number is just below the memory limit.

Excursus. Since for heavy skew, some buckets may be several times smaller than the largest one, it has been suggested to employ "bucket tuning" techniques [HLH95]. These try to make better use of the available memory by uniting small buckets into larger ones. However, bucket tuning has not been proven to significantly reduce response times because it still has to process the same amount of data and make the same disk accesses. In fact, the number of compare operations in a hash join may increase because the hash classes are now larger. More importantly, even if the join on a "tuned" bucket takes no longer than the corresponding series of small joins, it will require a larger amount of memory because all the data from the small buckets is processed at the same time. This will probably be harmful in multi-user mode. Thus, we do not consider bucket tuning advisable. Instead, we suggest processing all local joins on a node sequentially. Only when memory, processors, and disks are all underutilized, buckets might be either "tuned" or simply processed in parallel. But this situation should not occur in the first place if parameters *d* and *v* are properly adjusted to provide high read bandwidth for the join nodes. ❑

2.5 Degree of bucket declustering (*v*)

This is the most interesting of the five parameters we are discussing, and we will elaborate on it more extensively than on the others. The choice of *v* has significant consequences on the degree of parallelism as well as on the amount of disk contention for both the scan phase and the join phase. Before discussing different possible settings of *v*, we observe that there is a certain asymmetry between the scan and join operations: While each scan node must write into all buckets simultaneously, a single join node will access only one bucket (and *v* disks) at a time. One corollary of this is that while disk bandwidth depends on *v* in the join phase, this is not an issue in the scan phase; all disks will be equally loaded independent of *v*, as long as there are at least as many buckets as there are disks (i. e. $b \geq d$). Another conclusion concerns the interpretation of write and read contention:

Excursus. In table 2, we defined write contention by the number of scan nodes simultaneously writing to the same disk. It is traditional wisdom that this figure is an indication of the amount of access conflicts. Note, however, that the true nature of such conflicts is in terms of disk read-write head movement: Assuming that each processor accessing a disk tries to read or write data contiguously stored in a particular location, the number of nodes working on a disk corresponds to the number of different positions between which the disk head has to move to fulfill the requests received. If there are *k* such positions, any request has a probability of $1 - 1/k$ to cause an expensive track seek operation; this probability increases with greater values of *k*.

These assumptions do not hold in the scan phase in our model. Here, k is not determined by the number of scan nodes; when multiple nodes write to the same disk, they will append their data to the *same* buckets. The truly relevant parameter is the number of bucket fragments stored on each disk, which corresponds to the number of seek positions: $k = b \cdot v / d$. Assuming that the fragments will receive approximately the same number of pages (i. e. no skew) in arbitrary order, the above probability calculation applies. Thus, higher values of v will increase write contention even if the value *wc* used before indicates otherwise.
Strictly speaking, however, the number of nodes accessing a disk is not completely irrelevant; there are a few restrictions to the arbitrary access order we just assumed:

- A single PN will never write two pages at exactly the same time; it will have to produce at least one more tuple to fill the next buffer page until it can generate another write request.
- Two pages produced by the same processor within a short time interval will rarely concern the same bucket (whose buffer page has just been emptied); the node is more likely to write to other buckets in between.
- In the case of synchronous writing, two requests from the same PN will never compete for disk access; the processor will wait for the first one to finish before issuing the second.

The first effect will tend to decrease disk contention, while the second one will slightly increase it; both of them appear to be negligible. Synchronous writing seems unnecessary for intermediate results that require no security measures such as commit protocols. Thus, we consider the parameter k an adequate indicator of disk contention in the scan phase.
Note that these observations apply to the scan phase only: In the join phase, the common assumption that each node accesses contiguous data is true, and *rc* is a realisitic indicator of read contention (within the current query). ❑

The important conclusion drawn from this excursus is that – contrary to intuition – *both* write *and* read contention are best avoided for low values of v. A higher degree of bucket declustering is useful only to support read parallelism in the join phase. The following sections will show the consequences of some common-sense settings of v.

2.5.1 No declustering: v = 1

Without declustering, the number of bucket fragments to be written equals b (or b/d per disk), and write contention is minimal. Note that this setting is impossible if $b < d$, because there would be fewer fragments than disks. It may also be problematic if the file system does not support concurrent writing to the same file; this case would demand $v \geq n$.
In the join phase, read contention is also minimized: Each join node reads from just one disk at a time, and with $d \geq m$, the join nodes can access disjunct disks, assuming good scheduling. In fact, if d is truly greater than m, some of the disks will remain idle while the processors suffer from low read parallelism ($rp = 1$).
⇒ Without read parallelism, this setting must be considered suboptimal in spite of its minimum write contention.

2.5.2 Full declustering: v = d

With every single bucket declustered across all d disks, pages for a particular bucket can be stored on any of them. With this setting, scan nodes can even write to separate files if demanded by the file system (provided that $d \geq n$, which is a reasonable assumption). Write contention, however, is maximized because we have the highest possible number of bucket fragments.
For the join, read contention is also maximized. It equals m because each node has to access all d disks to assemble its current bucket. This will probably outweigh the benefits of read parallelism achieved with the broad declustering.
⇒ Despite good read parallelism, this setting is also inadequate due to its high disk contention.

2.5.3 Write-optimal declustering: $v = n$

Like with full declustering, processors can write to disjunct files if needed; however, the scan fragments are no longer split up unnecessarily ($bf = sf$). Scan nodes can even be assigned separate disks if this should simplify access management.
During the join, read contention is lower than with full declustering ($n \cdot m/d$ instead of m) under the assumption of good scheduling.

⇒ This setting offers the best possible write properties in the scan phase for simple file systems; in addition, it reduces the number of bucket fragments ($b \cdot n$ rather than $b \cdot d$) as well as the amount of read contention in the join phase.

2.5.4 Read-optimal declustering: $v = d/m$

In comparison to the case of no declustering, the number of bucket fragments is significantly increased, leading to write contention in the scan phase. This setting can be supported by simple file systems only if $d/m \geq n$.
For the join, read contention is still avoided ($rc = 1$), but read parallelism is increased from 1 to d/m. This will improve join performance because the available disk bandwidth is fully exploited.

⇒ This setting provides maximum read parallelism for the join phase like full declustering does, while avoiding read contention and reducing write contention.

2.5.5 Conclusions

From the above examples, we can see that write contention – as indicated by the number of bucket fragments instead of parameter *wc* – can never be completely avoided if there are more buckets than disks ($b > d$). However, it can be kept relatively low by reducing the degree of declustering, *v*. Read contention is also decreased for low values of *v*, but it can be avoided completely for $v = d/m$ or lower. Since d/m also enables maximum read parallelism, this setting is ideal for the join phase. Write parallelism, as noted before, is not a problem in any reasonable case (i. e. for $b \geq d$). The development of the major performance indicators is depicted in figure 2. (Note that, due to the different dimensions of values included, the y-axis scale is distorted, and the shapes of the curves are simplified.)
There are only two possible reasons to divert from a setting of $v = d/m$. The first is to reduce the number of bucket fragments in order to alleviate write contention caused by frequent disk head movements. To analyze this situation, we have devised an explicit cost function that captures the disk access times of both query phases, thus weighing the pros and cons of different settings of *v*. Our calculations (which are given in appendix A1 because they are rather lengthy) show that in most cases, d/m is actually the best setting. The only noteworthy exception is single-user mode with a limited number of buckets; in this case, declustering should be avoided completely ($v = 1$).
The second possible reason to choose a different degree of parallelism is a simple file system that does not allow concurrent writing. As mentioned before, this case demands a setting of $v \geq n$, so that $v = max(d/m, n)$ is required.

2.6 Considering the second relation

So far, we have considered only relation *R* in our calculations in order to simplify the presentation. We shall now try to include the second relation, *S*. The computations of *n*, *d*, and *m* will work just as laid out in sections 2.1 to 2.3. However, the degree of declustering they were based on, *r*, must be adapted to reflect the declustering of *S*, the degree of which we denote *s*. If *R* and *S* reside on disjunct disks, *r* should be replaced with $r + s$; if they are stored on the same disks, *r* should be changed to $max(r, s)$ etc.

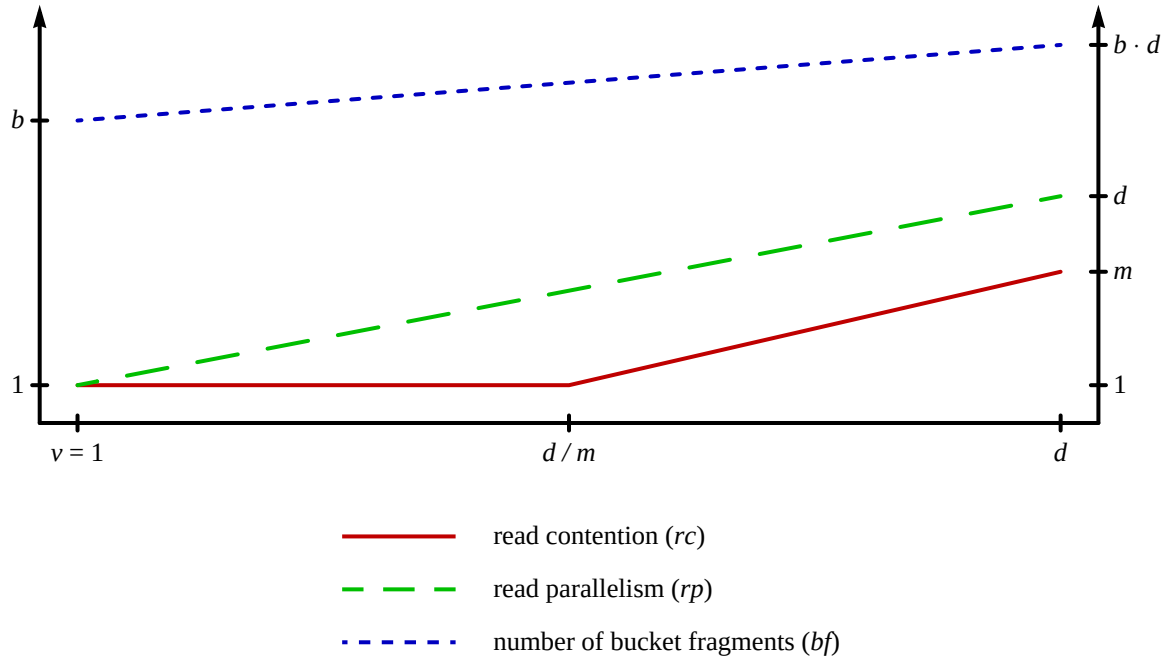


Figure 2: Development of performance indicators

To determine b, which we now interpret as the number of bucket *pairs*, the aforementioned heuristics must be applied to the *inner* relation from which the hash tables for the local joins are built and which thus decides their memory requirements. In general, the smaller of the two relations will be chosen as the inner one; this may be different in certain skew situations.
Finding an appropriate value of v is more involved. While the previous rules for determing v from the parameters n, m, p, and d are still valid, v itself can be interpreted in different ways, for instance:

1. The buckets of both relations are declustered with a degree of v, i. e. $v_R = v_S = v$.
2. The degrees of declustering for buckets from R and S are selected proportional to their sizes such that their sum equals v, i. e. $v_R + v_S = v$ and $v_R / |R| \approx v_S / |S| \approx v / |R \cup S|$. Two buckets of the same pair should then reside on different disks.
3. As in 2., with the additional constraint that the buckets of R and S reside on disjunct disks.

The first alternative seems appropriate for join methods that read the R- and S-buckets of a pair separately, as in standard hash join algorithms. In this case, both buckets may be stored on the same disks because there will be no access conflicts between them. Alternatively, it is possible to calculate v_R and v_S separately to reflect the fact that the build and probe phases may proceed at different rates. Options 2 and 3 are applicable when both buckets in a pair are processed simultaneously, e. g. in nested-loop or sort-merge joins (although the latter will require changes in the scan phase to ensure the sorting). Here, the matching buckets should be located on different disks; although they are never accessed at *exactly* the same time, the disk read-write head would oscillate between them if they were stored on the same devices.

3 Scheduling the join phase

Having determined the storage parameters for the buckets, processing the scan and storing the intermediate results is straightforward. With all scan nodes writing to all buckets, there are no more optimizations to be performed. In the join phase, proper scheduling is necessary to imple-

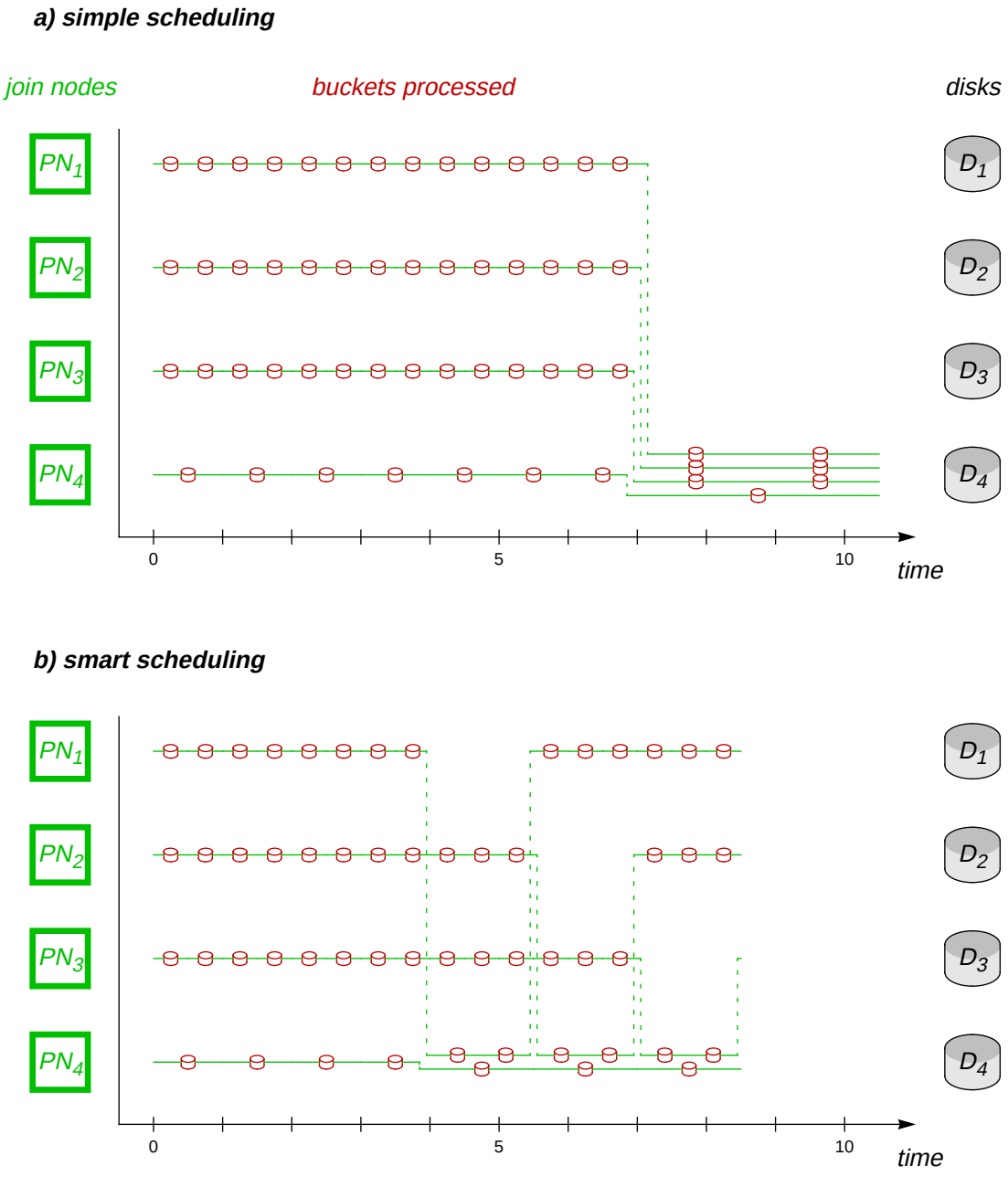


Figure 3: Possibilities of join scheduling

ment the conflict-free execution that is theoretically possible. When all buckets are the same size and all processing nodes work at the same speed (i. e. no skew), such scheduling is trivial: Each of the m nodes is assigned d/m disks and processes all the buckets stored on them. With skew, however, things get more complicated. Consider the case in figure 3:

Example. A set of 56 buckets has to be processed by four join nodes. They are stored on four disks without declustering, so that each node could simply read from one disk[3] if there were no

3. We have omitted declustering for clarity of presentation. It could easily be incorporated into the example by replacing single disks with sets of v disks and having each join node read from one such set. Either way, the illustration implements read-optimal declustering as found favorable in section 2.5.

skew. But for some reason (such as different bucket sizes or varying processor load), one of the nodes (PN_4) delivers only half the performance and has processed just seven buckets when the others have finished their shares of 14 each.
With simple scheduling (figure 3a), this will be noticed only towards the end of the join phase. The three faster nodes can then be assigned parts of the remaining work in order to "help" the overloaded processor. But since the buckets concerned all reside on the same disk, contention will occur and the nodes will hinder rather than help each other.
A smarter scheduling method (figure 3b) will monitor the progression of work and soon detect that PN_4 is lagging. It can then reassign work early and ensure that no more than two nodes access a single disk at the same time. This will reduce contention and increase the probability of "short" disk accesses compared to the previous case. Moreover, the maximum read bandwidth of disk D_4, which was mostly underused with simple scheduling, is now exploited more often.❑

3.1 Cost model

In order to compare the two approaches (which we will term *late scheduling* (LS) and *early scheduling* (ES), respectively) for more general parameters, we have to create another cost model. It is similar to the one used in sections 2.5.5 and A1 in that it is mainly based on the distinction of long and short disk access times. It will be restricted to single-user mode, though, because we can no longer assume asynchronous access in the join phase; synchronous processing, however, leads to a complex queuing model that is beyond the scope of this paper. In addition, we will assume that the parameters d and m have been selected such that the join nodes will optimally exploit the available disk bandwidth. As a consequence, we can only regard those types of skew that are caused by varying processor performance; bucket size skew can never be remedied by multiple nodes reading from the same disks if the latter are already fully loaded. To simplify our presentation, we will assume without loss of generality that $v = 1$, leading to $m = d$.[4] Now, we can formulate cost functions for both scheduling methods:

3.1.1 Late scheduling

Let us assume that each PN is assigned p pages of data for processing. One node is slower that the others by a ratio of $0 < l < 1$ and will have processed only $l \cdot p$ pages when the other $m - 1$ have finished their work. The remaining $p - lp$ pages have to be processed in a second stage, i. e. after the others are done. The overall disk access cost is

$$T_{LS} = p \cdot t_r + (p - lp) \cdot t'_r.$$

Here, t_r and t'_r denote the cost of a single disk access in the respective stage. In the first stage, there is no contention because each disk is accessed by just one processor; thus, $t_r = t_s$, representing a short disk access without any track seek overhead. In the second stage, h nodes cooperating in reading the remaining data will actually interfere with each other, making the disk read-write head jump between tracks:[5]

$$t'_r = \frac{1}{h} \cdot t_s + \left(1 - \frac{1}{h}\right) \cdot t_l.$$

It is obvious that the probability of a long disk access (t_l) increases with the number of nodes reading from the disk, h. As a consequence, the final $p - lp$ pages should be processed by a single (fast) node. This is because one processor can already load the disk to its maximum band-

4. The model can be extended to declustering with $v > 1$ in the same way as the example (cf. previous footnote). The results, however, will be exactly the same.
5. This assumes that there are at least h independent portions of data (i. e. buckets) left to process. Towards the end of the join phase, h will have to be reduced.

width; further parallelization will provide no more performance gains but merely increase contention. (This is also the reason why we did not represent parallelism in the formula, e. g. by dividing the second part by *h*. That would have been unrealistic because the disk has to fulfill all requests sequentially.) Setting $h = 1$ leads to $t'_r = t_s$, so that, finally,

$$T_{LS} = (2 - l) \cdot p \cdot t_s.$$

Note that all disk accesses are short; there is no disk contention at any time because the leftover pages will be processed by just one node. The price for this is a lost potential of parallelism both for PNs and for disks (the bandwidth of the overloaded node's disk is not fully exploited in the first stage).

3.1.2 Early scheduling

With early scheduling, processing is not split into two separate stages like with late scheduling. Still, the cost function will consist of two parts for the times when a node works on its own data only and when it shares a disk with another node. Furthermore, we have to set up two equations from the viewpoint of a fast node and of the slow node, respectively. Let us begin with a fast node:

$$T_{ES} = p \cdot t_r + \frac{p - q}{m - 1} \cdot t'_r$$

The first part of the formula represents the node processing its own data. It is the same as for late scheduling, including the fact that $t_r = t_s$ due to the exclusive disk access. In the second part, we can now assume parallel processing because the various nodes will help at different times. Note that *q* now denotes the *total* number of pages processed by the slow node; it does *not* equal $l \cdot p$ because the node will work on its data even while it is being helped. Furthermore, we now have a more specific definition of t'_r, knowing that when a fast node helps the slow one, there will be exactly two processors working on the same disk:

$$t'_r = \frac{1}{2} \cdot t_s + \left(1 - \frac{1}{2}\right) \cdot t_l = \frac{t_s + t_l}{2}.$$

From the viewpoint of the slow node, the cost formula looks like this:

$$T_{ES} = (p - q) \cdot t'_r + q' \cdot \frac{t_s}{l}.$$

The first part represents the times of cooperation with one of the fast nodes. In the second part, the slow node works alone on q' pages; being slower by a factor of *l*, the node needs an interval of t_s / l per page. The number of pages processed while working alone, q', is the difference of the total, *q*, and those read while being helped. Assuming that when being helped, the node takes no longer than t'_r per page (this is true for $l \geq t_s / t'_r$, which is the likeliest case), we get

$$q' = q - \frac{p - q \cdot t'_r}{t'_r} = 2q - p.$$

Putting both definitions of T_{ES} together and solving for *q*, we find that

$$q = p \cdot \frac{l + 1 - z}{2 - z} \quad \text{with} \quad z = \frac{m - 2}{m - 1} \cdot \frac{t'_r}{t_s} \cdot l.$$

Note that in contrast to LS, full parallelism is used all the time, regarding not only the processing nodes but also the disks. On the other hand, ES requires concurrent disk access that will lead to increased response times.

3.1.3 Comparison

To find out which scheduling approach is preferable under which circumstances, we have to compare the values of T_{LS} and T_{ES}. Starting from

$$T_{LS} > T_{ES} \qquad \Leftrightarrow \qquad (2-l)\cdot p\cdot t_s > p\cdot t_s + \frac{p-q}{m-1}\cdot t'_r,$$

we can find several equivalent conditions (the somewhat lengthy transformation is presented in appendix A2):

$$m > \frac{1-l}{2\cdot t_s/t'_r - l} + 1 \quad \Leftrightarrow \quad l < \frac{2\cdot t_s/t'_r\cdot(m-1)-1}{m-2} \quad \Leftrightarrow \quad \frac{t'_r}{t_s} < \frac{2\cdot(m-1)}{(m-2)\cdot l+1}.$$

We can see that ES outperforms LS for a large number of processors, for a truly slow overloaded node, or for a low ratio of concurrent and exclusive access times. This corresponds to the common-sense observation that LS will fail when there is a significant share of leftover work that must be processed sequentially (low *l*); conversely, ES will profit from a high degree of parallelism (large *m*) and suffer when concurrent access is expensive (high t'_r/t_s). To see that the above condition is realistic, consider the final version of the inequation: Knowing that $l \le 1$, we can state that

$$\frac{2\cdot(m-1)}{(m-2)\cdot l+1} \ge \frac{2\cdot(m-1)}{(m-2)\cdot 1+1} \ge \frac{2\cdot(m-1)}{m-1} \ge 2.$$

This means that, for ES to be superior to LS, it is sufficient to know that

$$t'_r \le 2\cdot t_s \qquad \Leftrightarrow \qquad t_l \le 3\cdot t_s.$$

If this condition is not true for single page requests, it can easily be fulfilled by allocating more buffer space and reading several pages of data at once, reducing the relative overhead for disk head positioning. (The access times for the corresponding number of pages must then be substituted for those of single pages in our formulae.) In fact, such a setting should be applied in general to speed up processing, and in the case of hash joins, the small buffer overhead will be negligible compared to the amount of memory required for the hash table itself.

3.2 Analysis

We have seen that in most sensible cases – in particular, for a realistic level of processor performance and with a reasonable buffer allocation policy – ES will outperform LS because the performance gains from parallel processing outweigh the losses from concurrent disk access. The option of late scheduling with full parallelism was quickly discarded because it causes extreme disk contention.

While our analytical model is restricted to single-user mode, we can still draw come conclusions for the multi-user case. We have seen in the calculation for the optimal degree of declustering (sections 2.5 and A1) that inter-query parallelism will increase the likelihood of track seek delays for any disk access; the resulting performance loss is greater for operations that previously relied on exclusive access than for those that already suffer from intra-query contention. In the current model, this means that t_r, now equivalent to t_s, will be increased by a larger factor than t'_r. As a consequence, early scheduling will appear even more favorable than in single-user mode.

In order to implement ES, a system has to be monitored during query execution, so that the progression of work can be supervised and the appropriate measures can be taken *as soon as possible*. This supports the case for dynamic load balancing we have made earlier [Ra96, Mä98, Mä98a]. The approach of on-demand scheduling (ODS) [Mä98], which allocates tasks to pro-

cessors one at a time according to the current status of processing, appears to be a suitable basis for an implementation of dynamic disk scheduling as considered in this chapter. Though ODS does not currently account for the order of disk access, this could easily be incorporated into the algorithm.

4 Conclusion

In this report, we have investigated various strategies of disk scheduling for intermediate results of large join queries. To the best of our knowledge, ours is the first report that has studied this issue in the context of shared-disk systems while including the particular conditions of multi-user mode. Apart from some general heuristics for the selection of processing nodes, disks, and buckets, we have reached two important conclusions:

- When join buckets are stored on disk, it is useful in most cases to decluster each bucket across several disks to facilitate parallel reading during the join itself. The optimal degree of declustering is such that the join processors can keep all disks busy without introducing intra-query contention.
- When processor performance fluctuates during the join phase, task reassignment between nodes should take place early enough to enable full parallelism without excessive disk contention. This requires a dynamic load balancing authority that monitors the progression of work and takes the appropriate measures.

In the future, we plan to validate these results in simulation studies, using our newly developed simulation system, *SimPaD* [MS98]. In particular, we will extend our on-demand scheduling algorithm that has proven successful in previous studies [Mä98] to incorporate the new findings.

Acknowledgment

The author wishes to thank Dr. Dieter Sosna for his help in clarifying the properties of the cost function used in appendix A1.

References

[HLH95] K. A. Hua, C. Lee, C. M. Hua: *Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning*. IEEE Transactions on Knowledge and Data Engineering 7 (6), 1995.

[Mä98] H. Märtens: *Skew-Insensitive Join Processing in Shared-Disk Database Systems*. IADT Workshop, Berlin, 1998.

[Mä98a] H. Märtens: *Options in Scan Processing for Shared-Disk Parallel Database Systems*. Technical Report, University of Leipzig, 1998.

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

[MS98] H. Märtens, Th. Stöhr: *SimPaD – A Generic Simulation System for Parallel Databases*. Technical Report, University of Leipzig, in preparation.

[Ra93] E. Rahm: *Parallel Query Processing in Shared-Disk Database Systems*. HPTS-5 Workshop, Asilomar, 1993.

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

[SD89] D. A. Schneider, D. J. DeWitt: *A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment*. ACM SIGMOD Conf., Portland, 1989.

A1 Determining the optimal degree of declustering

We construct a cost function for the I/O of the intermediate results, i. e. the buckets. It consists of writing and reading the buckets in the scan and join phase, respectively:

$$T = T_w + T_r .$$

If p denotes the number of pages to be written, and if we assume all the disks to be employed all the time (no skew), then

$$T_w = \frac{p}{d} \cdot t_w ,$$

where t_w is the average time taken for a single write operation. This is estimated as

$$t_w = \frac{1}{k_w} \cdot t_s + \left(1 - \frac{1}{k_w}\right) \cdot t_l .$$

Here, k_w denotes the number of write positions active at the same time. There is a probability of $1/k_w$ that the disk read-write head need *not* be moved, leading to a "short" disk access (t_s); otherwise, a "long" access (t_l), including a track seek operation, will occur. Defining $t_\Delta = t_l - t_s$ as the difference between long and short disk accesses, we can simplify

$$t_w = t_l - \frac{t_\Delta}{k_w} .$$

The number of active write positions, k_w, is determined by the number of bucket fragments per disk plus an adequate number of entry points for concurrent queries in multi-user mode, x. Thus,

$$k_w = \frac{b \cdot v}{d} + x .$$

Note that our formula does not include waiting times caused by write requests not being served immediately. This is because we assume asynchronous access so that processing can continue while data is (queuing to be) written. Since we further assume that the disks are not generally overloaded, our model need only capture the actual disk access times.
For read operations in the join phase, we can only assume $m \cdot v$ disks to be active (each of the m join nodes reads its current bucket from the v disks across which it is declustered). We have to note, though, that $m \cdot v$ must not exceed d; otherwise, there would be contention between the join nodes. This assumption is harmless, however, because we have already limited v to a maximum of d/m in section 2.5.5. Now, we can define

$$T_r = \frac{p}{m \cdot v} \cdot t_r \quad \text{with} \quad t_r = t_l - \frac{t_\Delta}{k_r},$$

similar to the scan phase. The number of read positions, however, is much lower now because we have excluded contention within the current join:

$$k_r = 1 + x .$$

Here, we assume the same value of x as in the scan phase to represent the same degree of inter-query contention. We can now write down the complete cost formula as a function of v:

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

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

We must now find the minimum of this function within the bounds of $v \in [1, d/m]$. To this end, we will distinguish a number of cases.

A1.1 Single-user mode

Single-user mode is easily represented in our cost formula by setting the number of disk access positions for concurrent queries to zero: $x = 0$. T then simplifies to

$$\begin{aligned} T(v) &= \frac{1}{v} \cdot \left(\frac{p \cdot t_l}{m} - \frac{p \cdot t_\Delta}{m} - \frac{p \cdot t_\Delta}{b} \right) + \frac{p \cdot t_l}{d} \\ &= \frac{p}{v} \cdot \left(\frac{t_s}{m} - \frac{t_\Delta}{b} \right) + \frac{p \cdot t_l}{d}. \end{aligned}$$

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

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

It is somewhat counter-intuitive that the result of this analysis depends on the number of buckets, b, but not on the actual amount of data, p. This can be interpreted as follows: For a high number of buckets, there are already a large number of fragments on each disk, leading to a very low probability of "short" write times (k_w is large, so that $1/k_w$ becomes small). Thus, increasing k_w further through declustering will do little harm in the scan phase while considerably speeding up the join phase. With few buckets (and bucket fragments), however, there is still a significant share of short write operations that will be destroyed by declustering, outweighing the performance gain during the join (which is the same as in the previous case because it does not depend on b). In contrast to b, p does not influence the number of write positions. It is merely a constant that can be factored out of the equation.

A1.2 Multi-user mode

In multi-user mode, the cost function cannot be simplified. Still, we will transform it into a more convenient notation by introducing four new "shorthand" coefficients:

$$\alpha = \frac{p \cdot t_l}{m} - \frac{p \cdot t_\Delta}{m + mx} \qquad \beta = \frac{p \cdot t_\Delta}{b} \qquad \gamma = \frac{dx}{b} \qquad \delta = \frac{p \cdot t_l}{d}.$$

Now, we can write T as

$$T(v) = \frac{\alpha}{v} - \frac{\beta}{v + \gamma} + \delta$$

and try do derive some results without having to consider the meaning of α, β, γ, and δ. We note, however, that all four parameters are greater than zero in the multi-user case.

Our first observation is that T is continuous for positive values of v (which is the range we are interested in). For $v \to 0|_{v>0}$, $T(v)$ converges to positive infinity, while for $v \to \infty$, it approaches δ. In between, it may or may not reach a minimum.

A1.2.1 The case of $\alpha \geq \beta$

Using the fact that

$$\frac{\beta}{v} - \frac{\beta}{v+\gamma} = \frac{v\beta + \gamma\beta - v\beta}{v(v+\gamma)} = \frac{\gamma\beta}{v(v+\gamma)},$$

we can rewrite T as

$$T(v) = \frac{\alpha - \beta}{v} + \frac{\beta\gamma}{v(v+\gamma)} + \delta$$

and find that for $\alpha \geq \beta$, T is strictly decreasing, meaning that a large value of v will deliver the best performance; thus, we should set $v = d/m$. For interpretation, the condition $\alpha \geq \beta$ can be expanded using the original definitions of the Greek shorthand coefficients:

$$\alpha \geq \beta \Leftrightarrow \frac{p \cdot t_l}{m} - \frac{p \cdot t_\Delta}{m + mx} \geq \frac{p \cdot t_\Delta}{b}$$

$$\Leftrightarrow \frac{1}{m} \cdot \left(t_l - \frac{t_\Delta}{1+x} \right) \geq \frac{t_\Delta}{b}$$

$$\Leftrightarrow \frac{b}{m} \geq \frac{t_\Delta}{\left(t_l - \frac{t_\Delta}{1+x} \right)}$$

In the final version we have isolated the number of buckets per join node, b/m, as well as the amount of inter-query contention, x. It is easily verified that for most sensible parameters (i. e. $b/m \geq 2$ and $x \geq 1$), the condition is true. In other words: Unless we are "almost" in single-user mode ($x < 1$), or we process just one bucket per join node, we should decluster the buckets with a degree of $v = d/m$.

The interpretation is similar to the single-user case: With a reasonable number of buckets, disk contention in the scan phase is already so severe that declustering will do no further harm at the time of writing the buckets, while increasing read performance during the join. This is true even in multi-user mode where inter-query contention affects both phases.

A1.2.2 The case of $\alpha < \beta$

Even though it will rarely apply in practive, let us now consider the case of $\alpha < \beta$. We now have no direct information on the shape of T, so we have to form its derivative to find a minimum:

$$T(v) = \frac{\alpha}{v} - \frac{\beta}{v+\gamma} + \delta \qquad \Rightarrow \qquad T'(v) = \frac{\beta}{(v+\gamma)^2} - \frac{\alpha}{v^2}.$$

Setting $T'(v) = 0$ and solving for v, we get (after some lengthy calculations we omit here)

$$v_1 = \frac{\alpha\gamma}{\beta - \alpha} \cdot \left(1 + \sqrt{\frac{\beta}{\alpha}} \right) \qquad \text{and} \qquad v_2 = \frac{\alpha\gamma}{\beta - \alpha} \cdot \left(1 - \sqrt{\frac{\beta}{\alpha}} \right).$$

Knowing that $\beta > \alpha$, we find that v_2 is negative and does not qualify as a solution. Thus, v_1 is the only sensible result. We know that v_1 is a true minimum because $T'(v)$ is negative for $v \to 0|_{v>0}$, positive for $v \to \infty$, and continuous in between (saving an analysis of the second derivative). The final question we have to study is how v_1 relates to the interval $[1, d/m]$, from which we have to choose v. If v_1 is within this range, it is our solution for the case of $\alpha < \beta$. If it is less than 1, we must set $v = 1$ because $T(v)$ is strictly increasing for $v > v_1$. If v_1 is greater than d/m, we select $v = d/m$ because $T(v)$ is strictly decreasing for $v < v_1$.

Unfortunately, no general statements can be made about the relationship in question. In fact, examples can be found for all three possibilities (v_1 being below, within, or above $[1, d/m]$), albeit for some rather unrealistic parameter settings. We now expand the solution of v_1 to

$$v_1 = \frac{\left(\frac{p \cdot t_l}{m} - \frac{p \cdot t_\Delta}{m + mx}\right)\frac{dx}{b}}{\frac{p \cdot t_\Delta}{b} - \frac{p \cdot t_l}{m} - \frac{p \cdot t_\Delta}{m + mx}} \cdot \left(1 + \sqrt{\frac{\frac{p \cdot t_\Delta}{b}}{\frac{p \cdot t_l}{m} - \frac{p \cdot t_\Delta}{m + mx}}}\right)$$

$$= \frac{\left(t_l - \frac{t_\Delta}{1 + x}\right) \cdot dx}{m \cdot t_\Delta - b \cdot \left(t_l - \frac{t_\Delta}{1 + x}\right)} \cdot \left(1 + \sqrt{\frac{t_\Delta \cdot m}{b \cdot \left(t_l - \frac{t_\Delta}{1 + x}\right)}}\right)$$

and replace some of the absolute parameters with their respective ratios:

$$b' = \frac{b}{m} \qquad d' = \frac{d}{m} \qquad t' = \frac{t_l}{t_\Delta}.$$

Here, b' is the number of buckets per disk, d' is the number of disks per join node (and the upper bound for v), and t' is the ratio of long disk access times to track seek times. This substitution leads to the following, slightly simplified formula:

$$v_1 = \frac{\left(t' - \frac{1}{1 + x}\right) \cdot d'x}{1 - b' \cdot \left(t' - \frac{1}{1 + x}\right)} \cdot \left(1 + \sqrt{\frac{1}{b' \cdot \left(t' - \frac{1}{1 + x}\right)}}\right).$$

This version is easier to analyze because we can set some strict bounds on the remaining coefficients:

$b', d', t' \geq 1$	according to their definitions;
$b' \geq d'$	because otherwise $d > b$, which would prejudicate declustering;
$b' \geq 2 \Rightarrow x < 1$	because otherwise, section A1.2.1 would apply;
$t' < 2$	because otherwise, section A1.2.1 would apply;
$x > 0$	because we are studying multi-user-mode.

We have evaluated the formula for various parameter settings; the results are shown in figure 4. We can see that in all cases, there is a very narrow margin of values of x for which the presumed optimum, v_1, is within the interval $[1, d/m]$. With any sensible degree of inter-query contention – in particular, for all cases of $x \geq 1 - v_1$ exceeds the upper bound so that the actual degree of declustering will be set to d/m. For some very small values of x, v_1 is below 1, and declustering is not advisable. This corresponds to “near”-single-user mode with few buckets per join node (cf. section A1.1).

The interpretation once again follows the same pattern as before: Even with a small number of buckets, inter-query contention of any practical degree can be expected to interfere with writing in the scan phase to such an extent that declustering will do no further harm while perceptibly speeding up the join phase through parallel reading.

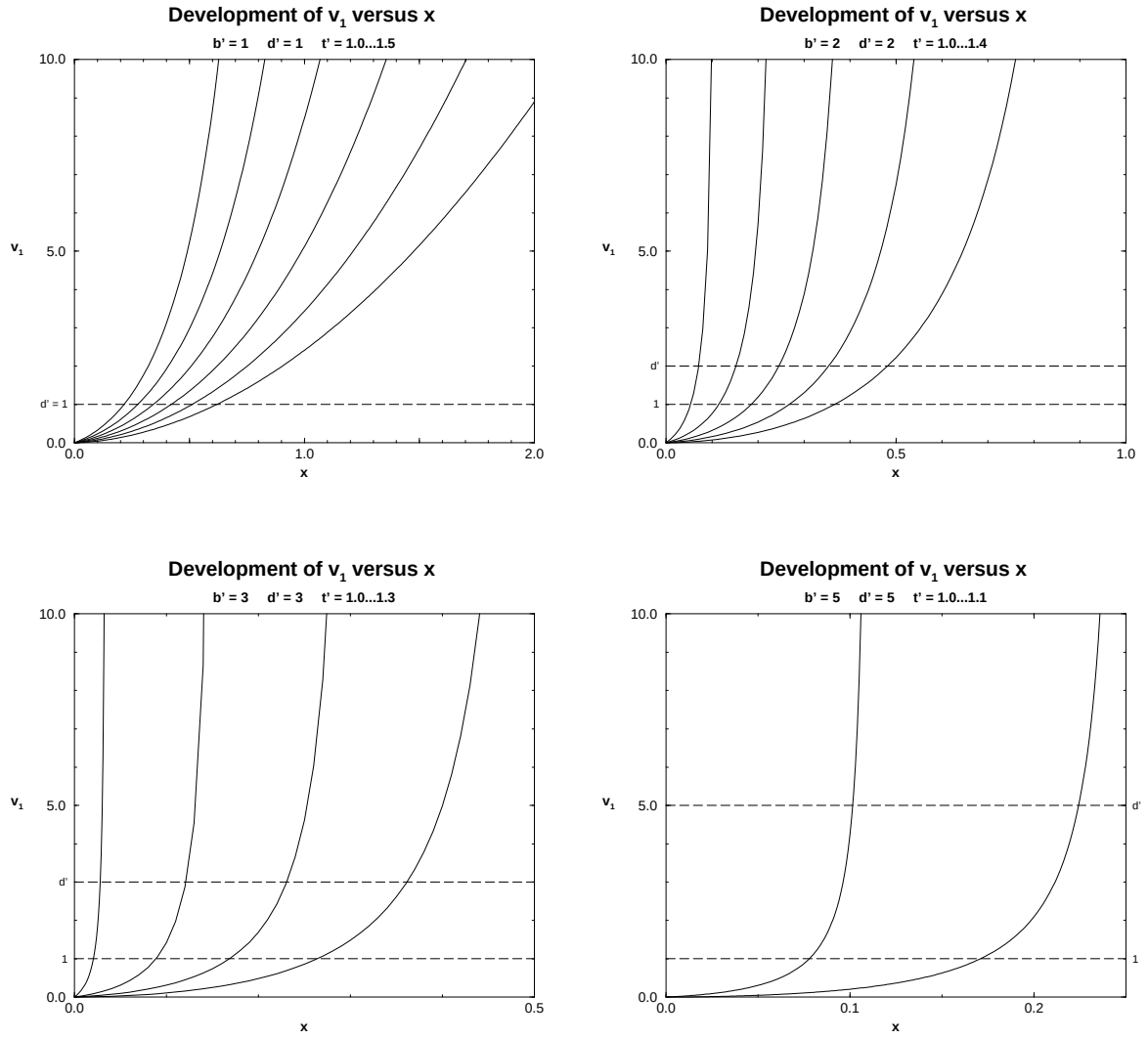


Figure 4: Development of the optimal degree of declustering

A1.3 Summary

Looking for an optimal degree of declustering, we found that in all practical cases, the "read-optimal" setting is favorable. The only noteworthy exceptions are for relatively small numbers of buckets in single-user mode; in this case, declustering should not be applied at all. Cases of medium declustering are not useful to consider.

The fact that v_1 is extremely sensitive to changes in *x* suggests that it is not a pronounced minimum of $T(v)$. Rather, the curve appears to be "almost strictly" decreasing in the sense that the actual value of the minimum, $T(v_1)$, is only slightly lower than the limit of $T(v)$ for $v \to \infty$, δ. As a consequence, even in the improbable case of a broader-than-optimal declustering, the resulting loss of performance can be considered negligible.

A2 Comparing the response times of late and early scheduling

To find out which scheduling approach is preferable under which circumstances, we have to compare the values of T_{LS} and T_{ES}. We find that

$$
\begin{aligned}
T_{LS} > T_{ES} &\Leftrightarrow (2-l)\cdot p\cdot t_s > p\cdot t_s + \frac{p-q}{m-1}\cdot t'_r \\
&\Leftrightarrow (1-l)\cdot p\cdot t_s > \frac{p-q}{m-1}\cdot t'_r \\
&\Leftrightarrow 1-l > \frac{1-q/p}{m-1}\cdot\frac{t'_r}{t_s} \\
&\Leftrightarrow 1-l > \frac{1-\dfrac{l+1-\dfrac{m-2}{m-1}\cdot\dfrac{t'_r}{t_s}\cdot l}{2-\dfrac{m-2}{m-1}\cdot\dfrac{t'_r}{t_s}\cdot l}}{m-1}\cdot\frac{t'_r}{t_s} \\
&\Leftrightarrow 1-l > \frac{1-\dfrac{(l+1)(m-1)-(m-2)\cdot t'_r/t_s\cdot l}{2(m-1)-(m-2)\cdot t'_r/t_s\cdot l}}{m-1}\cdot\frac{t'_r}{t_s} \\
&\Leftrightarrow 1-l > \frac{\dfrac{2(m-1)-(m-2)\cdot t'_r/t_s\cdot l}{2(m-1)-(m-2)\cdot t'_r/t_s\cdot l}-\dfrac{(l+1)(m-1)-(m-2)\cdot t'_r/t_s\cdot l}{2(m-1)-(m-2)\cdot t'_r/t_s\cdot l}}{m-1}\cdot\frac{t'_r}{t_s} \\
&\Leftrightarrow 1-l > \frac{2-(l+1)}{2(m-1)-(m-2)\cdot t'_r/t_s\cdot l}\cdot\frac{t'_r}{t_s} \\
&\Leftrightarrow 1-l > \frac{1-l}{2\cdot t_s/t'_r\cdot(m-1)-(m-2)\cdot l} \\
&\Leftrightarrow 2\cdot t_s/t'_r\cdot(m-1) > (m-2)\cdot l+1 \;=\; (m-1)\cdot l+(1-l) \\
&\Leftrightarrow 2\cdot t_s/t'_r > l+\frac{1-l}{m-1}.
\end{aligned}
$$

Using the final two variants, we can solve the inequation for the respective coefficients:

$$
m > \frac{1-l}{2\cdot t_s/t'_r-l}+1 \quad\Leftrightarrow\quad l < \frac{2\cdot t_s/t'_r\cdot(m-1)-1}{m-2} \quad\Leftrightarrow\quad \frac{t'_r}{t_s} < \frac{2\cdot(m-1)}{(m-2)\cdot l+1}.
$$