White Paper

Database Layout for SAP Installations with Informix

Data Balancing

Design: SAP Communications Media

Contents

1 Introduction

An important key to the performance of a computer system is the allocation of data to the disk subsystem. Because of the increasing complexity of applications the amount of application data is growing extremely fast. While todays storage systems offer a high capacity for storing application data, the speed of disk accesses can not keep up with the increasing amount of data. So it becomes vital to performance to distribute the data in an efficient way to guarantee optimal access times.

In recent times several customer installations with SAP hit the size of 1 terrabyte and over. With the conventional database layout I/O-performance is often not sufficient. Database growth often leads to hot spots on single disks resulting in degrading I/O-througput. This paper introduces a new concept for the database layout that can help to avoid I/O-problems by distributing all data over all available disks. This layout is applicable for large databases in high end environments as well as for smaller databases.

This paper summarizes our experience we gained in many customer projects with database sizes in the range of 200 GB to 1.2 TB. We want to present a database layout that meets the performance requirements of high volume SAP customers on Informix databases and has proven to be a success in several installations over the past 1½ years. With table fragmentation, detached indexes and Parallel Database Query (PDQ) Informix offers several features that can speed up performance if the database layout is designed correspondingly. These features can increase performance of daily operation as well as of administrative tasks.

The main design strategies are the same for all SAP applications and differ mostly in the database tables that need special treatment. This layout focuses on the following design strategies:

- Equal load distribution on all hardware components (disks, disk controllers, host adapters)
- Solution to I/O-problems
- Ease of administration
- Reduction of costs
- Speed up of administrative tasks like Update Statistics, Index Build, Checks by factor 10 and more (depending on hardware environment)

This paper first provides an overview of the current situation by describing the traditional database layout used by SAP. Chapter 2 also summarizes the optimizations that are currently implemented by many customers to improve I/O-performance. Chapter 3 presents the ideas of the new layout and gives a detailed suggestion for setting up a performant database layout. Chapters 4 and 5 provide information on the implementation of table fragmentation with Informix and the usage of PDQ. Additional information for the database administration during production with regard to database extensions without destroying the overall database layout is given in chapter 6. The appendix provides a detailed example of a database layout and gives examples for using PDQ to speed up administrative tasks and special application processes. This also includes information on the increase in performance that has been measured.

2 Traditional database layout

2.1 Standard layout

In a standard installation all database tables are assigned to a given set of standard dbspaces delivered by SAP. The classification of tables into these dbspaces was done based on their main purpose (master data, application data, pool- and cluster-tables, report loads etc.). Table 2.1 lists all standard dbspaces and gives information on their expected growth and the amount of read/write accesses to them. Initial stands for dbspaces that are only frequented heavily during initialization.

Each dbspace comes with a default size and SAP determines the allocation of dbspaces to disks depending on the rawdevices that are made available to the system. It is up to the customer to decide if this standard installation fullfills his needs or if he needs to change the default sizes and distribution of dbspaces.

Dbspace	Growing in Size	Read intensive	Write intensive	Use
PSAPBTAB	Yes	Yes	Yes	Application data
PSAPCLU	Yes	Yes	Yes	Cluster tables
PSAPDDIC	No	No	No	Data dictionary data
PSAPDOCU	No	No	No	Documentation data
PSAPEL40B	No	Initial	No	Compiled program data
PSAPES40B	No	No	No	Program sources
PSAPLOAD	No	Initial	No	Compiled program data
PSAPPOOL	Yes	Yes	Yes	Pool tables
PSAPPROT	Depends	Yes	Yes	Temporary objects
PSAPSOURCE	No	No	No	Program sources
PSAPSTAB	Depends	Yes	Yes	Master data
PSAPUSER1	Depends	Depends	Depends	User defined tables
ROOTDBS	No	No	No	Informix system
LOGDBS	No	No	Yes	Informix logical log
PHYSDBS	No	No	Yes	Informix physical log
TEMPDBS	No	Yes	Yes	Informix temporary data
PSAPSYSTEM	No	Yes	Yes	Informix system (since 4.5B)

Table 2.1: Standard dbspaces

Without paying attention to the dbspace allocation the data will be located on arbitrary disks, data from different dbspaces will get mixed up on the same disks (especially after dbspaces have been extended), fast growing or heavily accessed tables lie together on the same disks. When adding a new chunk all new data will go to the same disk leading to hot spots that are moving over time. Performance will suffer because of competing accesses to this data on one and the same disk.

2.2 Traditional load distribution

If I/O-performance problems arose, it was tried to eliminate hot spots by applying the following actions:

- Striping
- Relocation of disk partitions
- Individual dbspaces
- Fragmentation

These actions and their intended goal will be analysed in the following sections.

2.2.1 Striping

Up to now striping was often considered the ideal means to distribute data over several distinct physical disks and to achieve a good load distribution. Here we must distinguish between hardware and software striping. From our experience, software striping usually has a negative impact on system performance because of the additional overhead.

In general there are two different implementations of hardware striping. The first method combines all disks connected to a controller into one stripeset (vertically, see fig. 3.2) while the second method allocates the stripesets across several controllers (horizontally, see fig. 3.3).

Striping can help to achieve a similar fill level and an even I/O-load on all disks of a stripeset. But usually striping can only move hot spots from a single disk to a stripeset. If all disks of a stripeset are connected to the same controller, the controller will soon become a bottleneck. In addition striping has the following drawbacks:

- Striping causes an administrative overhead that increases with the size of a stripeset
- Software striping causes additional CPU overhead.
- An inadequate stripesize can cause a severe performance degradation if the stripesize does not correspond to the database blocksize.
- With striping there is no I/O transparency regarding read-ahead or prefetching
- Striping limits the read block size to the stripesize of usually 32k, 64k or 128k. This might have a negative impact on the read-ahead functionality because it reads the wrong data during read-ahead (split physical I/Os).
- Striping on its own cannot provide parallel database operations
- Recovery requires all disks of a stripeset to be recovered even if only one disk fails. This especially can present a problem if several dbspaces lie on the same stripeset.

A database layout that uses information on the size, access frequency and access patterns of the tables can achieve a better load distribution than a layout only relying on striping to distribute the data over several disks.

The database layout introduced here mainly relies on a good data distribution done by the database. This also works in combination with (hardware) striping but striping is not a prerequisite for a good layout. If hardware striping is not supported, software striping should not be used for the tables' data because together with table fragmentation this usually has a contrary effect on I/O-performance. For the index dbspaces and non-performance-critical dbspaces software striping can be of additional benefit (see chapter 3).

If software striping shall be used, we recommend to allocate the stripesets carefully so they fit into the overall database layout and are distributed over several disks served by different controllers (so that the number of disks equals a multiple of the number of controllers).

2.2.2 Relocation of disk partitions

The same holds for some tools offered by storage system vendors that allow to move some parts of a heavily used disk to another disk. This can be imagined as exchanging for example one of two hot partitions of a disk with a hardly used partiton of an idle disk. This method can also succeed only temporarily and causes a lot of “moving around” while hot spots continue to show up on new disks.

2.2.3 Individual dbspaces

Another means to resolve hot spots was the introduction of separate dbspaces for special tables. This means that individual tables were taken out of their default dbspace and put into a dbspace of their own. That dbspace was usually called psap<tablename>. The goal was to allocate a table, that was identified to cause a hot spot on one disk (in combination with the other tables of that dbspace lying on that disk) to a different dedicated disk with a lesser access frequency.

But this method could not really solve the problem, because hot spots usually move over time and a new hotspot can come up soon. So one was always chasing hot spots and trying to find out an idle disk where to move data to.

Another drawback imposed by this solution is the increased complexity and administrative overhead.

2.2.4 Fragmentation

Especially if a table threatened to reach the 32 GB dbspace limit, table fragmentation was used to resolve this situation. Fragmentation means that a table is split into several fragments, each located in its own dbspace. Up to now it was often recommended to create several individual fragment dbspaces for each table to be fragmented instead of having one set of fragment dbspaces for all tables (as proposed below).

If these fragments were allocated to distinct physical disks each and the fragments of different tables were also allocated carefully as not to overlap on a few disks, this was also a good way of I/O-distribution. But often fragments of different or even of the same table were allocated to one and the same disk, this again leading to new hot spots.

Even if the allocation to disks was done in an optimal way, this layout was very hard to administrate. So it could easily happen that a well-planned layout was destroyed after some time of operation. This is because different tables often had a different number of fragments and it was hard to keep track of where which tables’ fragments were really located. But also the fast growing number of dbspaces (for a 16-way fragmented table we had 16 new dbspaces) presented a problem for monitoring and database extension.

3 Data balancing

When designing a database layout the main focus should be on a good load distribution on all available hardware components. The goal of this layout is to distribute all data over all available disks and all available controllers (disk adapters (DA)). The following formula is used to determine the degree of distribution D.

$$D = C * n$$

Here, C is the number of controllers and factor n has to be determined individually, depending on the operating system and the storage system in use. For the following let us assume that n equals the number of disks connected to one controller. So with the above formula we can achieve data balancing over all available disks.

With the method of table fragmentation Informix provides a means to distribute tables over several dbspaces. Informix offers two different fragmentation schemes: Fragmentation by round robin and fragmentation by expression. We are currently using the round robin scheme because it is easier to implement than fragmentation by expression and it results in equal fragment sizes for all tables. With fragmentation by expression you would have to work out a suitable expression for each single table that should be fragmented. Depending on how this expression is chosen this might soon re-introduce hot spots caused by inequal load on the fragments (higher load on most recent data as described above).

The round robin fragmentation scheme distributes all rows of a table in a round robin manner equally over all available fragments. As it is not recommended from a performance point of view to have attached indexes with a round robin fragmentation scheme, the tables' indexes will be "detached" and allocated to a separate dbspace.

The idea now is to take all tables that are growing in size and all tables that are heavily accessed out of their default dbspaces (psapbtab, psapstab, psappool, psapclu, psapprot), see table 2.1 for a list of all dbspaces that contain such tables. About 98% of all tables in a R/3 system are static regarding their growth and show only little I/O load. Only 1% of all tables grow heavily and so determine the size of the whole database. As well only 1% of all tables show a high I/O. These tables will be collected in a new set of fragment dbspaces. This set will consist of D single dbspaces called e.g. psapcust01 to psapcust<D> for the tables' fragments. The detached indexes will allocated to one or more special index dbspaces (e.g. psapindex).

The most important prerequisite for a good I/O-load distribution is to allocate each fragment dbspace to a different physical disk. To distribute accesses to psapindex or the other remaining standard dbspaces over as many disks as possible, disk striping (hardware or software striping) can be used.

Figure 3.1 gives an example of how this database layout could look like on a storage system that does not generally use hardware striping. Section 3.6 explains how the layout can look like for storage devices that use hardware striping. Chapter 6 deals with possibilities of future database extensions in the context of this layout. Appendix 7.1 gives a more detailed example for the database layout.

On some hardware or operating systems special restrictions might apply that could involve in some points modifications from this layout. But in general the ideas presented here should serve as the basis for each newly implemented database layout. Filesystems and operating system swap space are not regarded in the following, these can usually be allocated to separate disks.

3.1 Fragmented tables

Possible objects for fragmentation are all large tables, all growing tables and all tables that are heavily accessed. Note that even for relatively small tables fragmentation can be useful if they are heavily accessed. On the other hand, fragmentation imposes an overhead on a single table access (e.g. for starting several scan threads). Because of this overhead small tables should generally not be fragmented.

A prerequisite for an equal I/O-load distribution is that each fragment dbspace must be located on a different physical disk. It does not make sense to have different fragments competing for I/O on the same disk. So if hardware striping shall be used, each stripeset should hold exactly one fragment (see section 3.6).

This section only deals with the fragmented tables' data, the allocation of the tables' indexes will be described in section 3.2.

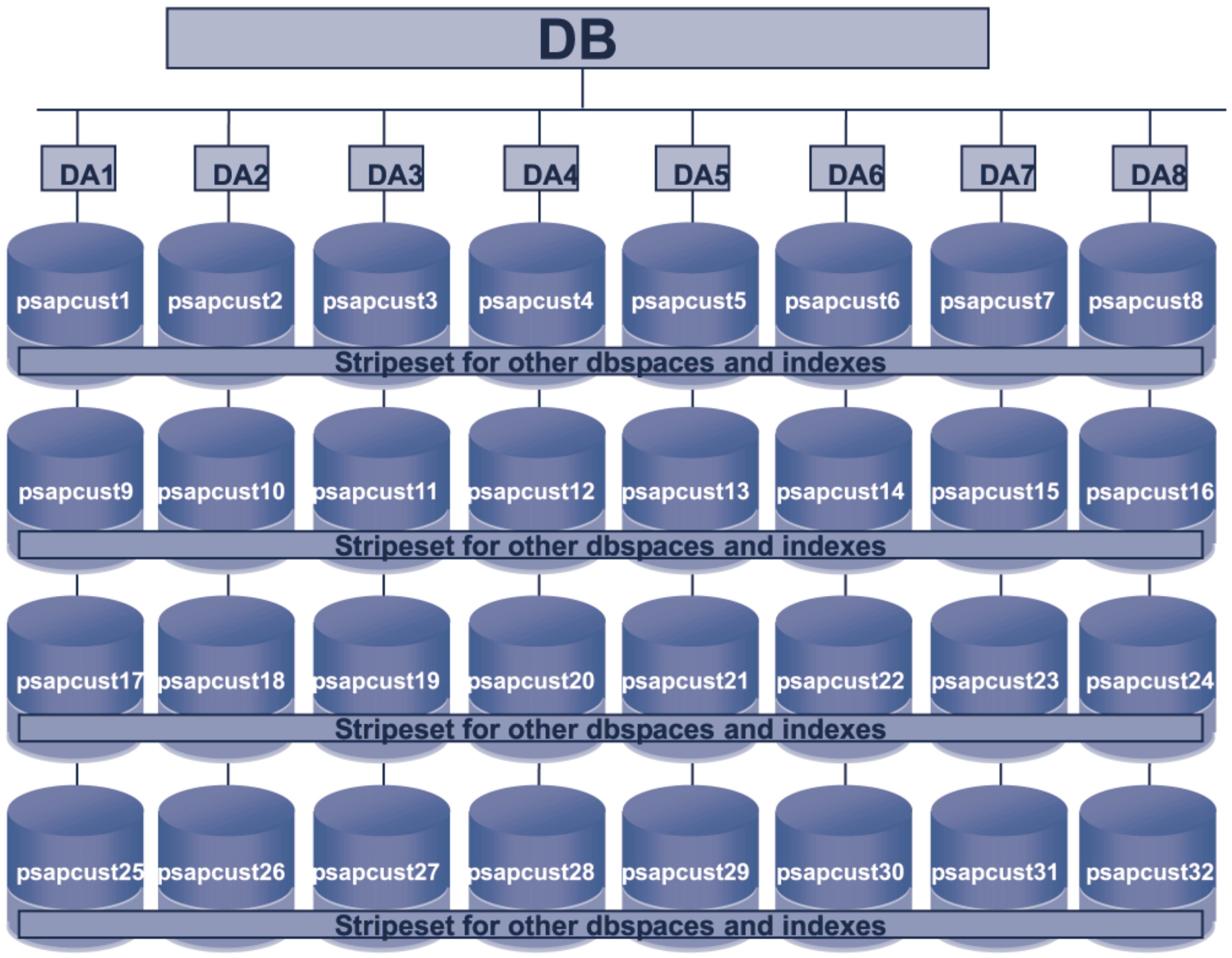


Figure 3.1: Optimal data balancing without hardware striping

Distributing data over that many different disks by using fragmentation offers the following advantages:

- The system's I/O load is evenly distributed across all hardware components.
- There will be no hot spots on single disks because all tables are distributed on all disks and all fragments have about the same size (sizes may vary a bit even with the round robin distribution scheme because of deletions).
- System performance exploits the whole performance capacities provided by the disk subsystem. Performance bottlenecks can be identified early, insufficient I/O-performance can be clearly tracked back to an insufficient hardware power.
- Full table scans and administrative tasks like index build, update statistics and oncheck can be sped up drastically by using parallelism (PDQ).

Further advantages compared to traditional methods of I/O-distribution (as described in chapter 2.2):

- Planning and maintaining of database layout is easier and less expensive
- Monitoring and administration of the database is easier and less expensive
- No hot spots to resolve (Black Box in a positive meaning)
- Under ideal circumstances only the fragment dbspaces have to be monitored for growth because the remaining dbspaces do not grow much over time.
- Only the fragment dbspaces or index dbspaces have to be extended when the database grows.
- Easier to monitor for freespace than with traditional fragmentation: We only have to check if a fragment dbspace threatens to fill up. If each fragmented table had its own dbspaces, all these would have to be monitored.
- Fewer need to extend dbspaces compared to traditional fragmentation. Only one chunk needs to be added to each of the fragment dbspaces and all fragmented tables can continue to grow. If each fragmented table had its own dbspaces, the need to extend some of them arises more often.
- No waste of disk space. If each table had its own fragment dbspaces, each of them must have freespace to allow table growth. If that table does not grow, this space is wasted. With "collective" fragment dbspaces the available freespace can be used by all tables in there.

On the other hand it is important to know that:

- Having collective dbspaces for all fragmented tables has the disadvantage of extent overlapping between different tables in these dbspaces. But from our experience this does not have much negative impact on performance if the first and next extent sizes for these tables are chosen appropriately. Extent overlapping can mostly be avoided by appropriate first and next extent sizes.
- Collective dbspaces have the danger of excessive extent growth if the next extent size is not chosen correctly and if there is not enough freespace available to guarantee table growth by this size (e.g. if only small extents can be allocated). Because a reorganization would be very difficult and time consuming it is vital to monitor the number of extents of these tables on a regular basis and to adjust the next extent size if necessary. There must always be enough freespace available in that dbspaces to enable a correct table growth (also see 3.5.1).
- Software striping should not be used at all for the fragment dbspaces because this has a contrary effect to the goals of fragmentation.
- When an extension of the fragment dbspace becomes necessary, one chunk must be added to each fragment. This can be quite a time consuming action if the number of fragments is high. There must be free disk space on each disk to avoid an overlapping of fragments (and coming with that the reintroduction of possible hot spots).
- A further increase in I/O-performance is only possible by exchanging or extending the current hardware. But this is also an advantage because the time needed to analyse and optimize I/O problems can be spared.
- Depending on the number of disks and the database size it could be a good idea to have several sets of fragment dbspaces (if a single fragment dbspace would become too big). This can especially be necessary if in combination with hardware striping there is only a relatively small number of fragments used (as described in section 3.6).

3.2 Detached indexes

When using the round robin fragmentation scheme, attached indexes would also follow this fragmentation scheme. For performance reasons it is not recommended to have an index fragmented by round robin. So the indexes must be detached. Detached indexes also provide the advantage that data and index pages do not interleave in a dbspace which reduces the amount of data to be read during a table scan.

In general, several detached indexes can be put together into a common dbspace. But this might cause a problem for large indexes because it is not possible to determine the extent size (or next extent size) of a detached index individually. Because the extent size for indexes as calculated by Informix is relatively small, this could result in extent overlapping and a high number of extents for each index contained in such a common index dbspace. In principle, each index could also be located to a dbspace of its own, but this causes an immense administrative overhead.

We therefore propose to put only large and continuously growing indexes into dbspaces of their own (called for example psap<indexname>) and to use common dbspaces for every 5 to 10 of the other detached indexes (called for example psapindex1 to psapindex<j>).

To avoid hot spots on the index dbspaces (resp. the disks where these dbspaces are allocated) each index dbspace must be distributed over as many disks as possible (in ideal case D disks). Several implementations of the above layout that did not distribute the index dbspaces over several disks showed hot spots on those disks containing the indexes. So these single disks became the restricting factor for overall I/O performance.

To distribute an index dbspace over several disks one might think of a distribution by simply allocating the dbspace's chunks on different disks, that is the first chunk on one disk, the second chunk on another disk and so on. But because these chunks are not filled up evenly as the fragments are (the chunk on the first disk will be filled up first, then the chunk on the second disk and so on) we will usually see hot spots on the current chunk because it holds the most recent and therefore most often accessed data (unless we choose very small chunk sizes).

The best method to distribute the index dbspace over several physical disks is the use of striping (also see section 2.2.1). As most operating systems or hardware vendors impose limits to the size and number of disks contained in a stripeset it might not be possible to achieve an index distribution over all available disks (as shown in figure 3.1). An alternative could be to set up several distinct stripesets each holding a set of index dbspaces and to distribute the indexes evenly over those stripesets.

3.3 Remaining dbspaces (Non-fragmented tables)

After all growing and heavily accessed tables have been taken out of their default dbspaces and fragmented into psapcust1 - psapcust<D>, the remaining standard dbspaces of table 2.1 should be uncritical with respect to performance and further growth. These dbspaces can be allocated in the same way as the index dbspaces are, that means they can be distributed over a large number of disks with the usage of disk striping.

In the future there will be no need to take a single table out of its standard dbspace unless fragmenting it. Even small tables that are heavily accessed can be fragmented if they turn out to be hot spots.

Under ideal circumstances these standard dbspaces should not be growing because all tables that grow over time should be fragmented. If a table contained in these dbspaces turns out to be growing too much, it can be reorganized into the fragment dbspaces psapcust<i>. This method ensures simple administration because usually only the fragment dbspaces need to be monitored for growth and eventually need to be extended. So this also reduces the danger of making bad allocation decisions for newly added chunks.

With respect to growth and I/O load the dbspaces should now look as shown in table 3.1.

3.4 Special dbspaces

The system dbspaces rootdbs, physdbs, logdbs, psapsystem as well as the temporary dbspaces can be treated alike all other dbspaces. Especially for logdbs it can be of advantage to have it striped across a large number of disks because with unbuffered logging it is heavily accessed by many small write requests.

An alternative for ensuring a high performance when writing logical logs would be to reserve some fast disks exclusively for logdbs. Because logdbs is very performance critical and because each transaction that modifies data writes the logs here, these should be the fastest available disks. If choosing this alternative, there should be no other dbspaces or filesystems located on these disks.

To speed up database operations that need temporary dbspaces, the number of tempdbs should equal the number of virtual cpu processes (CPU-VPs). To enforce parallel operations like sorting each CPU-VP can access a different tempdbs. The use of striping for the temporary dbspaces can provide an additional advantage because the I/O requests involved are served from different disks in parallel. Different tempdbs should reside on different stripesets.

Dbspace	Growing in Size	Read intensive	Write intensive	Use
PSAPBTAB	No	No	No	Application data
PSAPCLU	No	No	No	Cluster tables
PSAPDDIC	No	No	No	Data dictionary data
PSAPDOCU	No	No	No	Documentation data
PSAPEL40B	No	Initial	No	Compiled program data
PSAPES40B	No	No	No	Program sources
PSAPLOAD	No	Initial	No	Compiled program data
PSAPPOOL	No	No	No	Pool tables
PSAPPROT	No	No	No	Temporary objects
PSAPSOURCE	No	No	No	Program sources
PSAPSTAB	No	No	No	Master data
PSAPUSER1	No	No	No	User defined tables
ROOTDBS	No	No	No	Informix System
LOGDBS	No	No	Yes	Informix logical log
PHYSDBS	No	No	Yes	Informix physical log
TEMPDBS	No	Yes	Yes	Informix temporary data
PSAPSYSTEM	No	Yes	Yes	Informix system (since 4.5B)
PSAPCUST<i>	Yes	Yes	Yes	Fragment dbspaces
PSAPINDEX<j>	Yes	Yes	Yes	Index dbspaces

Table 3.1: Dbspaces with the new layout

If the database mirroring for rootdbs, physdbs and logdbs is turned on, the mirrors must reside on a different stripeset than the originals to ensure data security.

3.5 Further important design issues

3.5.1 Extent size

Because it might be extremely hard or even impossible to reorganize a very large fragmented table or even a complete dbspace containing several fragmented tables, the extent size needs special attention. It must be avoided under all circumstances that any table comes near the limit of 256 extents (per table fragment). The extent growth of all tables in the fragment dbspaces psapcust<i> must be monitored on a regular basis, especially during the first time after implementing this layout.

To monitor the number of extents you can use transaction DB02 and choose button "Space Statistics". If some table shows a large number of extents, the next extent size must be adapted. If the number of extents is continuously growing although the next extent size is correctly set, this can be caused by insufficient freespace. If there is not enough continuous freespace available, the table will be extended by extents that are smaller than the next extent size and thus result in frequent extensions.

First and next extent sizes of all table must be set carefully. The first extent size should equal to the currently known tablesize. For fragmented tables the first extent size equals the expected tablesize divided by the number of fragments. This ensures that continuous disk space will be allocated for that table. The next extent size must be set according to the expected table growth to avoid excessive extent growth and extent overlapping.

3.5.2 Numbering of Chunks

If possible, chunks with consecutive numbers should not be placed on the same physical device. Otherwise the database cleaners will interfere with each other when doing a checkpoint because the cleaners work on the chunks in ascending order.

3.6 Database Layouts using hardware striping

Figure 3.1 shows a database layout on a storage system without hardware striping. For storage devices that implement hardware striping, the layout looks a bit different. The number of fragments will equal the number of stripesets, that means one fragment psapcust<i> will be allocated to one stripeset. Depending on the size of those fragment dbspaces it can be necessary to use not only one set of fragment dbspaces psapcust1 - psapcust<n> but two or more such sets (e.g. paspcust1_1 - papcust1_<n> and psapcust2_1 - psapcust2_<n>) and to divide the tables to be fragmented between these sets. Each set of fragment dbspaces should span all disks (resp. stripesets).

Each table should belong to exactly one set of fragment dbspaces.

Figure 3.2 shows a sample layout on a storage system using vertical hardware striping (striping on controller level) with one set of fragment dbspaces. Figure 3.3 shows a layout on a storage system using horizontal hardware striping (striping across controllers) with two sets of fragment dbspaces.

Index dbspaces, remaining dbspaces and special dbspaces will be allocated in the same way as described in sections 3.2 to 3.4.

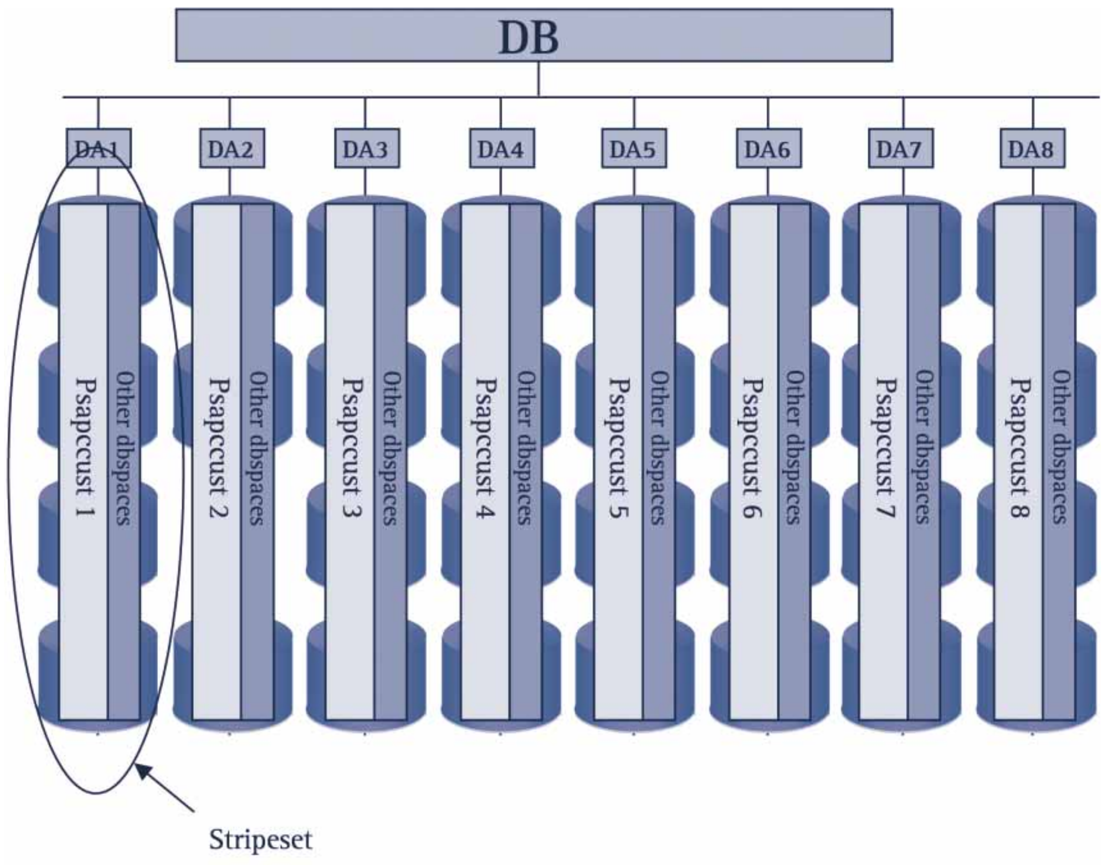


Figure 3.2: Optimal data balancing with vertical hardware striping

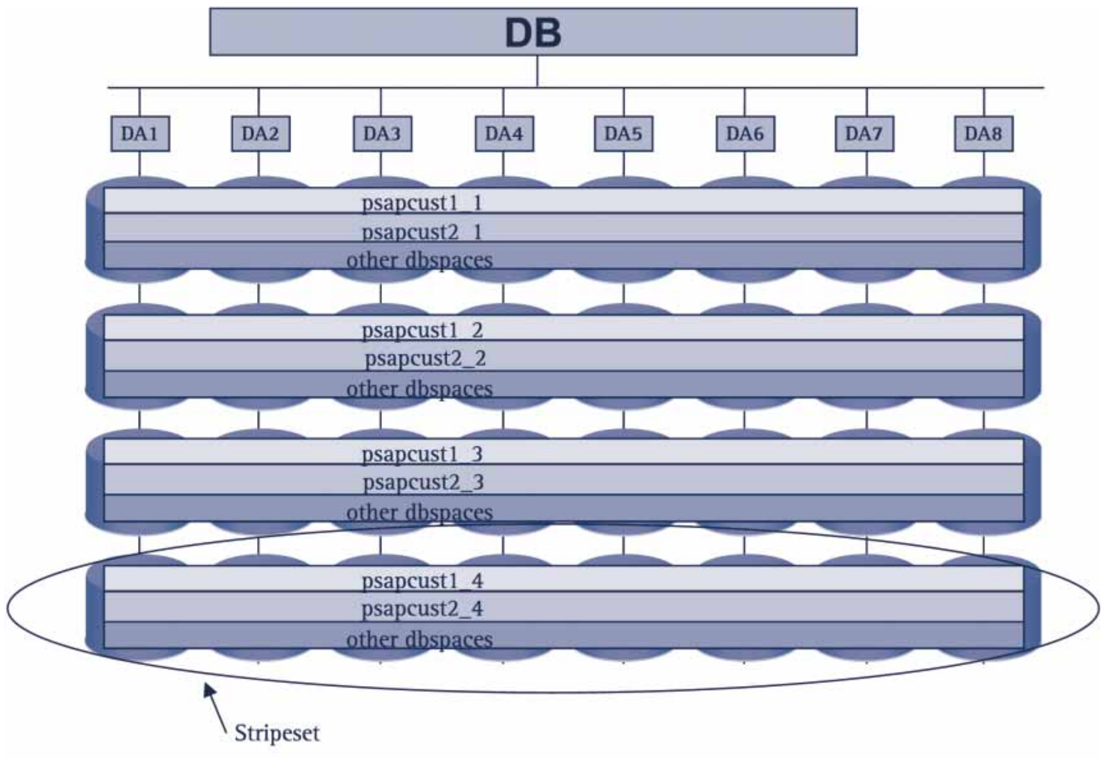


Figure 3.3: Optimal data balancing with horizontal hardware striping

4 Implementation of data balancing

4.1 General process

To implement the above layout we must distinguish between the situation of a system before start of production and an already productive system.

The best and easiest way to implement a new database layout is before start of production. A number of tables expected to fullfill the conditions described above can be identified and created in psapcust right from the beginning. If, after start of production, some additional tables turn out to be candidates for fragmentation these can be easily reorganized afterwards.

For a production database a migration strategy has to be worked out and tested. In principle, there are two strategies to change the layout of an existing database:

1. If the migration shall be done ”inplace” on the same storage device enough free space is needed to migrate even the largest tables. To achieve an optimum layout, each disk must provide some free space to hold the new fragment dbspaces psapcust<i>. Then all tables that are to be reorganized can be fragmented one at a time into the new dbspaces. This process can be stopped and at any time be restarted for additional tables. This method only requires short (but possibly many) periods of downtime.
2. Move the database to a different storage system. This strategy is applicable for example if a new storage system shall be used because the old one became too small to hold all data. Another example would be an altered hardware strategy e.g. providing new functionality like high availability solutions. In some cases this will also be the only possible method because the prerequisites for the first method cannot be met. Under these circumstances the new database layout will be built up on the new storage device before actually migrating the data during an offline period by using a database link (“insert into select”-method).

A very important thing when (re)creating a table with fragmentation is the correct lock mode ”row” which must be given to the ”create table” statement (the default lock mode of Informix is ”page”, as opposed to the lock mode used by SAP). This can be guaranteed by using the command ”dbschema -ss” when extracting the table’s schema.

The following sections describe some possibilities on how to implement fragmentation for selected tables. The necessary steps are only given in rough details. For more detailed information please see the Informix manuals.

4.2 Table reorganization with SAPDBA

To implement fragmentation with SAPDBA only the method ”reorganization of a single table’ can be used. If you want to implement fragmentation for a large number of tables this is not a very practical way. Please use the newest version of SAPDBA. The database must be in no-logging mode, that means the R/3 system must be down. There must be enough temp-space available.

If an already fragmented table shall be reorganized with SAPDBA, the fragmentation will be preserved.

Process:

- choose menu path Reorganization ➠ Reorganize Single Table. Provide the table’s name and choose method “Insert into select from”.
- adapt the first and next extent sizes, these must already be calculated to meet the requirements of the fragmented table (the first extent size of one fragment must be given here). Do not change the dbspace name or locking mode.
- SAPDBA will generate the table’s schema. This must be edited manually before letting SAPDBA execute the scripts. The schema can be found in directory /informix/<SID>/sapreorg. Edit the file <table>.schema.table and replace the table’s dbspace name by the names of the fragment dbspaces (e.g. replace the phrase ”in psapbtab” by ”fragment by round robin in psapcust1, psapcust2, ..., psapcust<D>”). Edit the ”create index” commands to create detached indexes (e.g. add the phrase ”in psapindex”). Replace a primary key constraint by the corresponding index (for R/3 releases prior 4.0B).
- let SAPDBA execute the scripts

Advantages:

- Easy and comfortable method because SAPDBA

takes care of the reorganisation process including view and index creation as well as update statistics
- Parallel reorganisation of different tables is possible by opening several SAPDBA sessions (take care of a suitable first extent size !)
- Parallel reads with PDQ possible if the table to be reorganised is already fragmented (if PDQ is set for the SAPDBA session)

Disadvantages:

- Inserts are done sequentially
- A single SAPDBA session cannot work on a list of tables to reorganise

4.3 Table reorganization with "Init Fragment"

This method simply uses the "alter fragment .. init" statement to convert a non-fragmented table into a fragmented table. This statement can also be used to modify the fragmentation strategy and to change the number of fragments of a fragmented table. It can also serve to convert a fragmented table into a non-fragmented table.

Process:

- detach all indexes because otherwise they will remain attached and also be fragmented by round robin. This can be done in advance before the actual table reorganisation and is supported by SAPDBA.
- adapt the next extent size using an "alter table" statement (very important because the first extent size cannot be changed)
- execute the statement "alter fragment on table <table> init fragment by round robin in psapcust1, psapcust2, .. psapcust<D>"
- update statistics with SAPDBA

Advantages:

- Easy and comfortable method because
- Executed transactionally, returns an error message if something goes wrong (e.g. not enough space available)
- Faster than the other methods above
- Can be automated be scripts

Disadvantages:

- First extent size can not be changed, it is taken from the source table's definition
- Next extent size must be set adequately using the "Alter Table" statement to avoid excessive extent growth
- Table might be destroyed if the reorganisation is somehow interrupted

4.4 Table reorganization with "Detach Fragment"

This method is applicable for an already fragmented table that shall be reorganized (e.g. when the number of fragments shall be altered). The original fragmentation will be revoked and the data will be inserted into a newly fragmented table.

Process:

- generate the database schema including server specific syntax (dbschema-ss). Create one file containing the create table statement and another file containing all other statements.
- drop indexes and constraints on the table in question
- detach all fragments of that table, starting with the last fragment (order is important so that the table can be reassembled later on !). Use the statement "alter fragment .. detach fragment" as described in the manuals.
- after detaching all fragments there will be D separate tables (if the number of fragments was D) named for example mara_D to mara_2.
- rename the original table with the remaining data (e.g. from mara to mara_1)
- edit the schema and recreate the table using the desired degree of fragmentation
- use parallel insert processes to insert the data from the D small tables into the new table
- edit the schema for index creation to include the dbspace for the detached indexes (e.g. add the phrase "in psapindex"). A table constraint must be created as index.
- recreate indexes and views
- update statistics with SAPDBA
- remove the partial tables

Advantages:

- Reads and inserts can be done in parallel which reduces the total time needed for reorganization

Disadvantages:

- Very complex method
- Can only be used for already fragmented tables

5 Parallel Database Query

Parallel database query (PDQ) is an Informix feature that provides parallel operations on fragmented tables. Together with the database layout described above this is an ideal means to speed up administrative tasks as well as selected application processes.

PDQ can be switched on for a database session by setting the environment variable PDQPRIORITY to a value N between 1 and 100 or by issuing the SQL command SET PDQPRIORITY N. N should be set carefully to avoid all PDQ resources being granted to only few queries. For example if PDQ is set to 100 all other queries requesting PDQ resources would have to wait for the first query to finish ! For a detailed description of the PDQ resources and their allocation see the Informix Performance Guide.

Attention: Do not generally switch PDQ on for the database because this would dramatically reduce OLTP performance and query throughput, just switch it on for special tasks or selected statements and do not forget to switch it off afterwards.

5.1 PDQ and administrative tasks

Administrative tasks that can be sped up by PDQ include index creation and update statistics. As these tasks are usually executed during times when the system shows no or only little load, PDQ can be set to the maximum value of 100 to achieve the optimum performance gain.

The time needed for oncheck on fragmented tables can be linearly reduced by starting one oncheck process for each of a table's fragments. PDQ is not needed here.

Please see the appendix for examples on how to optimize these tasks.

5.2 PDQ and Open SQL queries

PDQ cannot be used for index access, it is only applicable to speed up full table scans. This works for select, update and delete operations but not for inserts. For a single full table scan with PDQ set to 100 runtime can be improved to less than 1/10 of the runtime without PDQ.

Under these premisses a full table scan with PDQ can be faster than an index access that was faster than a scan without PDQ (depending on the selectivity of the query). For larger result sets a full table scan with PDQ will be faster because the data pages are scanned sequentially whereas an index scan might read the same pages several times. To enforce a full table scan for a single select statement from within an ABAP report the "client trick" (for R/3 releases < 4.5A and Informix releases < 7.30) or the database hint "+full" (from R/3 releases >= 4.5A together with Informix releases >= 7.30) can be used. Please note that both methods should only be used for carefully selected cases and always imply a modification to the original code. See appendix 7.2.5 for an example.

PDQ can be switched on by issuing a "set pdqpriority" command with exec sql before the Open SQL statement. Please make sure that PDQ is switched off again immediately after the statement by issuing a "set pdqpriority off" command with exec sql. If you forget to switch off PDQ it will be in effect for all future processes executed in that R/3 work process (even if that is a completely different report executed by a completely different user !). In general PDQ must not be switched on for an R/3 session to avoid a negative impact on overall system performance !

Starting with R/3 release 4.6C there will be a second alternative to use PDQ for a specially selected Open SQL statement. The kernel hint "+ PDQ(n)" switches on PDQ for an Open SQL statement and switches it off after its execution.

An example for using PDQ in an ABAP report can be found in appendix 7.2.4.

Remark: The PDQ level that was set for the execution of a statement is saved together with the statement in the statement cache. So if a statement is already contained in the cursor cache and will be executed a second time with PDQ set to another value than before, this will have no effect and the statement will be executed with the PDQ level of the first execution. This problem does not arise if PDQ was switched on by using the R/3 kernel hint "+ PDQ(n)". That will result in a completely new statement and will have no impact on subsequent statements without PDQ.

5.3 Monitoring PDQ

The correct execution of PDQ queries and the resources allocated for their execution can be monitored with onstat.

“onstat -g mgm” shows the resources allocated by the memory grant manager. This includes the amount of decision support memory used, the number of currently active parallel scans as well as limitiations in effect for the PDQ resources.

“onstat -g ses” shows PDQ sessions (parallel threads)

The query plan of a query executed in parallel also indicates this by the expression ”parallel, all fragments”.

6 Database extension

One advantage of the proposed layout is the simplicity in dealing with future database growth. Under the assumption that all growing tables are located in psapcust, no extensions of other dbspaces than psapcust or psapindex become necessary. To ensure the possibility of reorganizing a fragmented table there should always be at least as much freespace left in each fragment dbspace psapcust<i> as the size of the largest table’s fragment is.

If the database needs to be extended because of (unexpected) table growth, the well-planned database layout might be endangered. If the positive effects implemented above are destroyed, performance will suffer and can easily lead to I/O bottlenecks.

If there is enough free space left on each disk, an extension can be easily done by adding a new chunk to each fragment dbspace. Otherwise an extension can only be done by adding new physical disks. Unless the number of new disks equals the number of fragments, an extension will always imply a serious restriction to the initial layout. But this problem is not unique to the layout presented here - similar problems always arise if a database must be extended by adding new disks. Regardless of the layout used, new disks are always endangered of becoming hot spots because new chunks are always added to these disks.

Note:

As all possibilities presented below imply severe disadvantages with respect to the initially planned optimal database layout, an alternative to be considered is to use a fully equipped storage system right from the beginning. The disk space that is not yet needed can be left unused in a way that the amount of unused space is the same on all disks. If more space is needed later on, each fragment can be extended onto its own disk without running into any of the above mentioned disadvantages. If finally that fully equipped storage system runs out of space, a second (fully equipped) storage system could be used to extend the layout.

If new physical disks must be added to provide the needed space you should always add a fixed set of disks. Never add just one or a few single disks because these will definitely become a hotspot. The minimum number of disks to add depends mainly on physical conditions. On the physical level there are two recommended possibilities of how to add new disks to a storage system:

- Horizontally:
  Extension by a multiple of C disks, that means adding one (or more) disks to each controller (see figure 6.1).
- Vertically:
  Adding one (or more) new controllers with n disks each (see figure 6.2).

From the database point of view, there are also two ways of providing more disk space to a table:

- Adding new chunks (see figure 6.1):
  Extensions are usually done by adding a new chunk (rawdevice) to the dbspace that holds the "overflowing" table. For a fragmented table this means that each of the fragments belonging to the table must be extended. For the above layout this involves adding D chunks, one to each psapcust<i> dbspace.

  Disadvantage:
  Chunks of several different fragments will now be located on the same physical disk, that means recent data might concentrate on the new disks. If the new disks are added vertically (new controller) this might also introduce a higher load on that controller. This situation could be avoided by adding D new physical disks so each of the D fragments can be extended onto its own disk.

- Adding new fragments (see figure 6.2):
  A second alternative is to add new fragments to a fragmented table by using the "Alter fragment .. add" clause. These new fragments could reside on one newly added physical disk each. Because Informix does not provide some kind of rebalancing mechanism for new fragments, these new fragments will hold fewer data than the original fragments (only newly inserted rows will be distributed round robin over the increased number of fragments). This implies that a table reorganization will probably be necessary.

  Advantage:
  Still equal distribution of recent data over all available physical disks, but only as long as the initial fragments are not yet filled up completely.

  Disadvantage:
  New fragments will be smaller unless a reorganization is done. If the initial fragments are filled up completely, there will be hot spots on those disks added later on. A necessary reorganization can be done for example as described in section 4.4 (as long as there is enough freespace left in each fragment-dbspace).

Figure 6.1 gives an example of physical disks being added horizontally while the database is extended by adding new chunks to each fragment. Figure 6.2 gives an example of physical disks being added vertically while the database is extended by a corresponding number of fragments. Although not displayed here, the two other combinations (horizontally / new fragments or vertically / new chunks) are also possible.

If one of the index dbspaces needs to be extended, this can be done by defining a new stripeset that holds a new index dbspace (e.g. as shown in figure 6.1 or 6.2).

Because larger tables always imply longer access times, the following recommendations should be obeyed to ensure long term performance:

- Use data archiving to keep table growth limited.
- Use reports to delete obsolete data where available.

See http://sapnet.sap.com/data-archiving for information on potentially large tables and their archiving objects respectively their corresponding reports to delete old data.

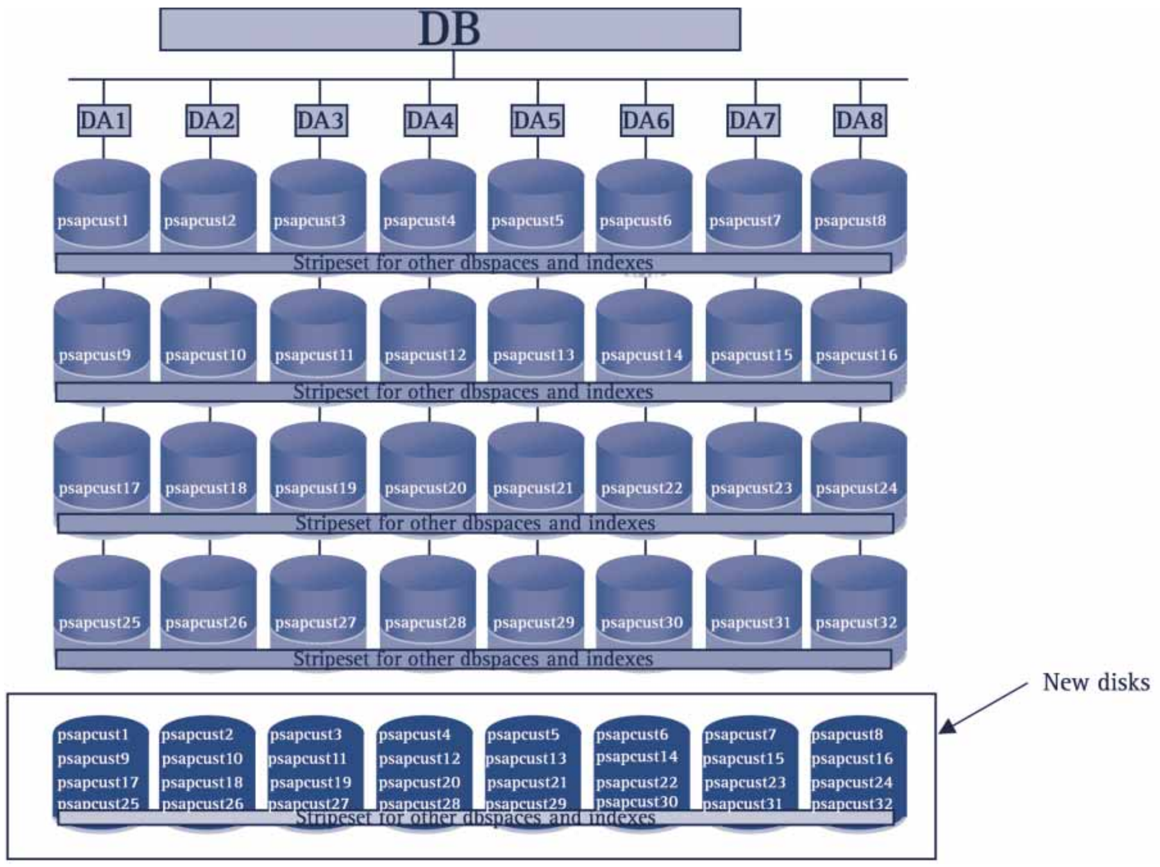

DB
DA1
DA2
DA3
DA4
DA5
DA6
DA7
DA8
psapcust1 … psapcust32
Stripeset for other dbspaces and indexes
New disks
psapcust1 psapcust9 psapcust17 psapcust25
psapcust2 psapcust10 psapcust18 psapcust26
psapcust3 psapcust11 psapcust19 psapcust27
psapcust4 psapcust12 psapcust20 psapcust28
psapcust5 psapcust13 psapcust21 psapcust29
psapcust6 psapcust14 psapcust22 psapcust30
psapcust7 psapcust15 psapcust23 psapcust31
psapcust8 psapcust16 psapcust24 psapcust32

Figure 6.1: Horizontal extension with adding new chunks

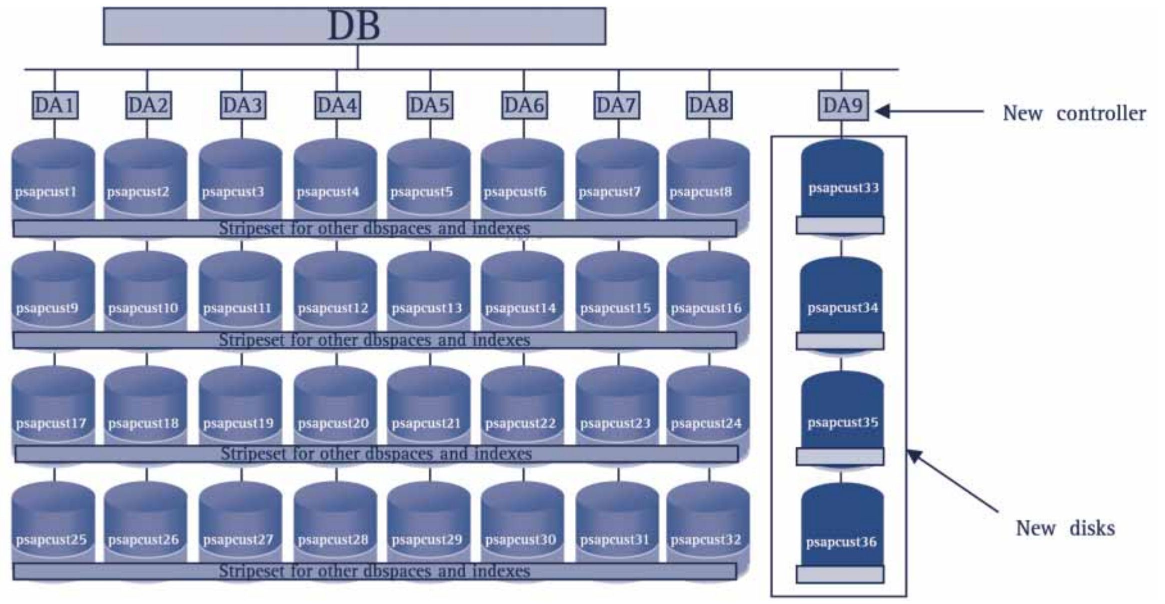


Figure 6.2: Vertical extension with adding new fragments

7 Summary

Data Balancing with Informix

Objectives	
• Same number of fragments for all tables • Allocate each fragment to a different physical disk • Fragment all growing and all heavily accessed tables • Distribute data over all disks	• Use 1 set of fragment dbspaces for all fragmented tables • Detach indexes into special index dbspaces • Distribute index dbspaces over as many disks as possible
Advantages	**Disadvantages**
+ I/O load evenly distributed ▪ over all disks ▪ over all controllers + No hot spots + Exploits whole capacities of disk subsystem + Parallel Database Query possible + Speed up administrative tasks + Less administrative overhead + Easier monitoring + Easy to extend (as long as there is freespace available) + No waste of disk space	– Overhead for single table accesses – Indexes cannot be distributed over all disks (unless a stripeset can span all disks) – Performance might suffer if extension by new disks becomes necessary
Important to note	
• Excessive extent growth must be prevented → Extent growth must be monitored → Adapt extent sizes if necessary → Add new space early enough to allow correct extent allocation • Enough free space must be preserved → Do not let the disks fill up completely → Add new disks early enough • Fragmentation imposes an overhead on table accesses → Do not fragment too small tables	

8 Appendix

8.1 Example database layout

This section gives a detailed example of a performant database layout. Let us assume that we have 8 controllers with 4 disks each. We do not have the possibility of hardware striping. We can use the layout scheme of figure 3.1 which could then look like figure 7.1.

We can use all disks for table fragmentation, that is we can have 32 fragments. The fragment dbspaces will be numbered consecutively from 1 to 32 and allocated to a different physical disk each (dbspaces psapcust1 to psapcust32). The data of all fragmented tables is evenly distributed over 32 disks. So the accesses to that data is also distributed over 32 disks and there will be no hotspots on single fragments or disks. With this method we also have access to 32 physical disks at the same time when doing a full table scan with PDQ.

For the indexes and the other remaining dbspaces we set up 4 stripesets with the help of software (operating system) striping. Each stripeset spans 8 physical disks which are connected to different controllers. Each of the remaining standard dbspaces will be allocated to one of these 4 stripesets in a way that the stripesets will be filled almost evenly.

For the detached indexes we can create 4 index dbspaces (psapindex1 to psapindex4), each allocated to one of the 4 different stripesets. The tables indexes are distributed over these index dbspaces such that they are filled quite evenly.

The system dbspaces are also allocated to the 4 stripesets. The original and mirror of one system dbspace are always located on a separate stripesets.

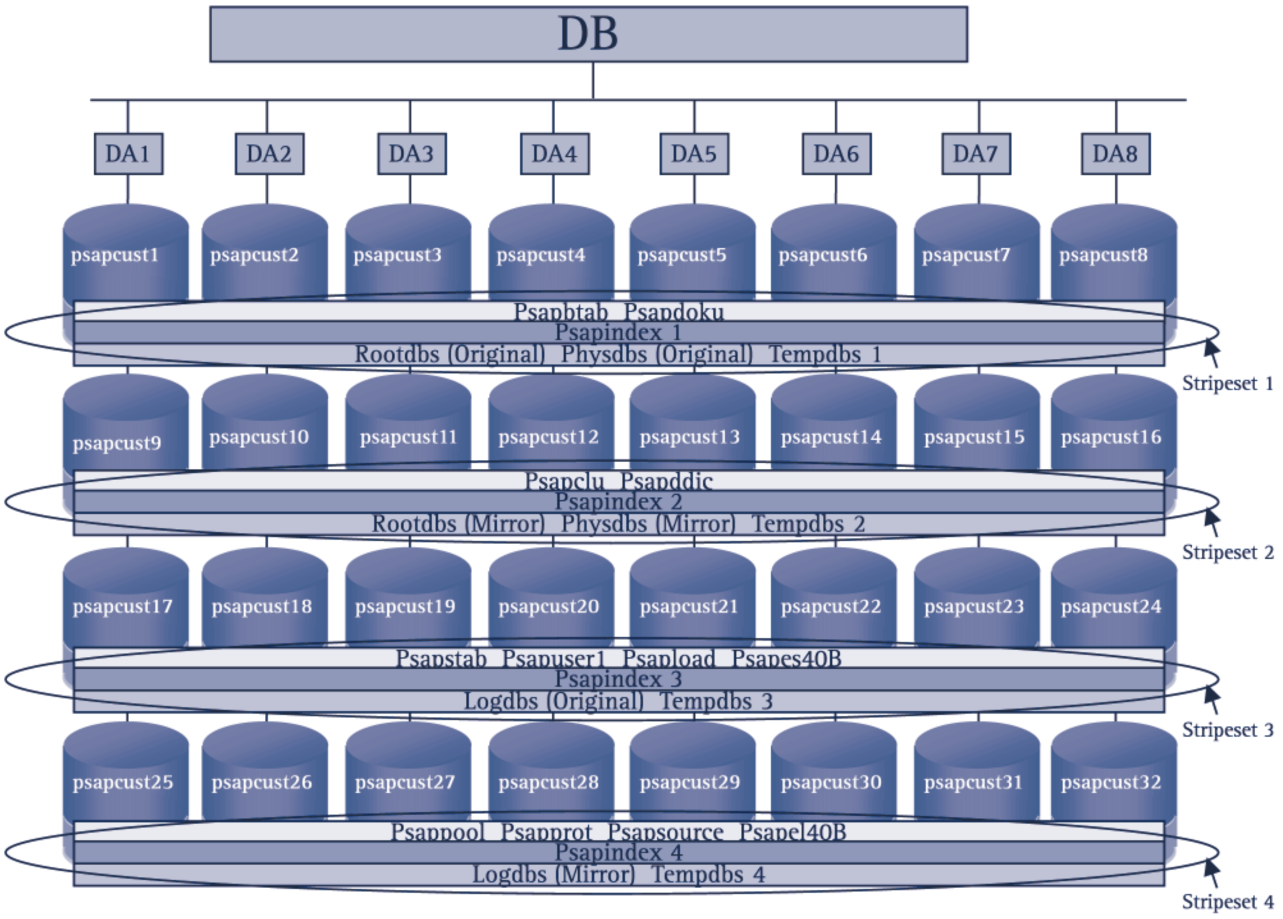


Figure 7.1: Example disk layout

8.2 Examples for the usage of Parallelism

All tests were done on the following system environment:

- SUN Enterprise 450
- 4 CPUs (248 MHz)
- 1 GB Memory
- 4 Disk-Controller SCSI Ultra Wide

The following table(s) where used

- MARD (Material Master)
- 1.6 Mio rows
- 2 versions :
  - non-fragmented, detached indices
  - fragmented in 8 dbspaces, detached indices
- each fragment on different physical disks

8.2.1 PDQ for index creation

Executed SQL command (on UNIX level) create index mard_test on mard (mandt,pstat,minbe,lagpr) in psaptest

Results

- nonfragmented table;
  PDQ=0 ➠ runtime 2 min 45 sec
- fragmented table;
  PDQ=100 ➠ runtime 0 min 16 sec

(Relevant UNIX/Informix settings:

```
DS_TOTAL_MEMORY=100MB

PSORT_NPROCS=4)
```

Performance gain: factor 10

8.2.2 Parallel oncheck on fragmented tables

Normally when you have a nonfragmented table you can do an oncheck by executing a command like this:

```
oncheck -cID <database>:<table>
```

For a fragmented table you have the advantage to start an oncheck for each fragment of a table at the same time. You can do this as follows:

```
oncheck -cID <database>:<table>,<fragment_dbspace1> &
oncheck -cID <database>:<table>,<fragment_dbspace2> &
oncheck -cID <database>:<table>,<fragment_dbspace3> &
oncheck -cID <database>:<table>,<fragment_dbspace4> &
...
```

So all fragments of the table can be checked all at once what can result in a performance gain up to factor n (n = number of fragments).

8.2.3 PDQ for update statistics

Executed SQL command (on UNIX level)

```
sapdba -updstat -t mard -threshold 0 -XXL
```

Results

- nonfragmented table PDQ=0
  Logfile:
  ```
  12:14:43  -------
  12:14:43   loglevel=2
  12:14:43  -------
  12:14:43  [generated by the incredible SAPDBA
            V4.6B]
  [SNIP]
  summary...
  12:29:32  [1 times successf. LOW on whole table
            (0 unsuccessf.)]
  [SNIP]
  ```

  ➠ runtime: 14 min 49 sec

- fragmented table PDQ=100
  Logfile:
  ```
  12:31:08  -------
  12:31:08   loglevel=2
  12:31:08  -------
  [SNIP]
  12:36:53  summary...
  12:36:53  [1 times successf. LOW on whole table (0
            unsuccessf.)]
  [SNIP]
  ```

  ➠ runtime: 5 min 45 sec

Performance gain: factor 2.6

8.2.4 PDQ in ABAP reports (e.g. period closing)

The following example for using PDQ in an ABAP report is an update command which is used in the period mont closing report in R/3. This is a very time consuming report which updates table mard of the material master.

```
update
   mard
set
   vmlab     =     labst ,
   vmuml     =     umlme ,
   vmins     =     insme ,
   vmein     =     einme ,
   vmspe     =     speme ,
   vmret     =     retme ,
   lfgja     =     :bukrs_tab-jahr ,
   lfmon     =     :bukrs_tab-monat
where
       ( mandt is not null and mandt = :sy-mandt )
and
       lfgja        = :bukrs_tab-vorjahr
and
       lfmon         = :bukrs_tab-vormonat
and
      werks in ( select werks from t001w where bwkey in
                     ( select bwkey from t001k where
                     bukrs = :bukrs_tab-bukrs ) )
```

Results:

- nonfragmented table;   PDQ=0
  ➠ runtime 50 min 39 sec
- fragmented table;   PDQ=100

```
   exec sql.
       set pdqpriority 100
   endexec.

   update ...

   exec sql.
       set pdqpriority 0
   endexec.
```

   ➠ runtime 10 min 43 sec

Performance gain: factor 4.7

8.2.5 Enforcing full table scans

As PDQ can only be used with full table scans it can sometimes be necessary to force the optimizer to a full table scan instead of an index scan. This can be done with an optimizer hint or with the ”client trick”.

a) Example for the database hint ‘+full’ in ABAP (>= 4.5A and >= 7.30)

```
SELECT * FROM MARD %_HINTS INFORMIX
   ‘FULL(MARD)’.
```

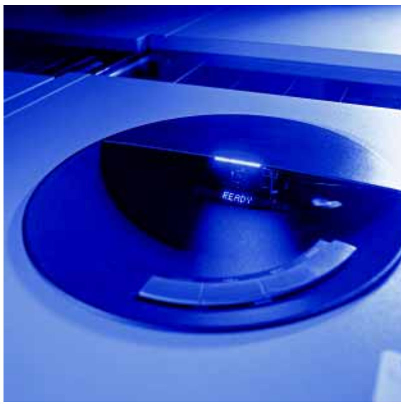
b) Example for the ‘Client trick’

```
data: xmandt(5) value ‘%   %’.
write sy-mandt to xmandt+1(3).
select  mandt matnr werks lgort from mard client
   specified
into i_mard
where  lgort = ‘0001’  and
        werks in (‘0001’)  and
        mandt like xmandt.
endselect.
```

SAP AG
Neurottstrasse 16
69190 Walldorf, Germany

Mailing address
69189 Walldorf, Germany

SAP information service
Tel. +49 (180) 5 34 34 24
Fax +49 (180) 5 34 34 20



www.sap.com

You can find this and other current literature on our home page in the media centers for each subject.

50 038 258s (0003) Printed on environmentally friendly paper.