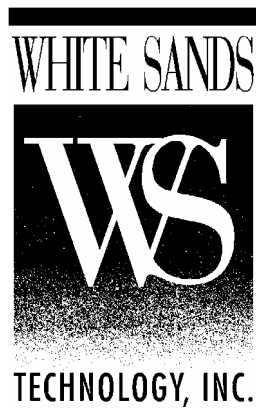


# Fragmentation and Database Performance

*A technical white paper by  
White Sands Technology, Inc.*



<http://www.whitesands.com/>

## **About the Authors**

White Sands Technology produces the ProActive DBA family of products and is a leader in the development of performance tuning, maintenance, diagnostics and disaster recovery solutions for Sybase ASE databases. Their home page on the Web is located at [www.whitesands.com](http://www.whitesands.com).

Also assisting with this white paper was John McVicker. John achieved near-legendary status in the Sybase community as a performance-tuning guru. He is currently a Systems Architect and Performance Manager within the Enterprise Systems Management division of Inventa, and was a Principal Consultant for over five years with Sybase Professional Services, the consulting division of Sybase, Inc. He can be reached via e-mail at [jmcvicker@yahoo.com](mailto:jmcvicker@yahoo.com) or [jmcvicker@inventa.com](mailto:jmcvicker@inventa.com).

## Table of Contents

<b>Preface.....</b>	<b>1</b>
<b>Introduction.....</b>	<b>1</b>
Why Be Concerned With Fragmentation? .....	1
<b>What is Fragmentation? .....</b>	<b>2</b>
Definition of Terms .....	2
Fragmentation Defined.....	3
<b>Types of Fragmentation .....</b>	<b>4</b>
Messy Page Chains .....	4
Poor Page Utilization .....	6
Extent Fragmentation .....	8
Row Fragmentation .....	9
<b>Dealing With Fragmentation .....</b>	<b>9</b>
Analyzing Effects of Deletes.....	10
Analyzing Effects of Inserts.....	10
Analyzing Effects of Updates .....	11
Preventing Extent Fragmentation .....	12
Capacity Planning for Defragmentation .....	12
Use Fixed-Length Row Sizes .....	14
Use Monotonically-Increasing Clustered Index Keys .....	14
Use Unique Clustered Index Keys .....	15
Large I/O Considerations .....	15
<b>The Big Picture .....</b>	<b>16</b>
<b>Conclusion .....</b>	<b>17</b>
<b>Index .....</b>	<b>18</b>

---

## Preface

This white paper is intended for DBAs and managers who must be concerned with keeping production Sybase database systems running at peak performance levels. It assumes a good degree of familiarity with Sybase database servers.

Microsoft SQL Server versions 6.5 and earlier share a common data structure with pre-11.9.2 versions of Sybase; thus, much of the information in this white paper applies to those versions of Microsoft SQL Server as well.

---

## Introduction

One of the last things client/server developers often think about is the long-term storage considerations of their database objects, namely tables.

The production-support DBA is going to support the application in its day-to-day usage, so it is up to the DBA to watch over active database objects such as disk devices, segments, table sizes, and index usage.

The developers work from a blueprint of the application as it will be used and create specific logical designs to match. However, the next step is to create a good physical design to support the application's logical design work.

The physical design steps should include making sure that the correct data types are selected for tables, ensuring that the proper normalization is done, and planning for the long-term growth and maintenance of the Sybase server and its databases.

### Why Be Concerned With Fragmentation?

During normal operations, the Sybase DBA must maintain a high performance RDBMS environment. This includes keeping service levels and performance ratings of the system, which include average response times of OLTP applications and batch jobs, as well as maintaining a high amount of availability of the system for the users.

While Sybase ASE and SQL Server are high-performance RDBMS engines, the database engine can perform only as well as the applications that are written for it allow.

*Good performance must be planned for during development.*

Standardized benchmarks such as TPC-C aside, well-written applications perform very well only if all pre-production application development is done with an eye towards long-term performance requirements of the business as well as the RDBMS engine that the application is written for.

The topics of defragmentation and performance-related aspects of table design have not been as mainstream with Sybase ASE and SQL Server as with other RDBMS products. Defragmentation of Sybase database tables is not considered a requirement by Sybase or by many Sybase DBAs, since tables do not have pre-set size parameters such as extent size and growth size factors found in other vendors RDBMS engines.

*Proper database design maximizes return on hardware investments.*

However, tables in Sybase databases can become fragmented in various ways. And, as many DBAs have discovered in recent years, the proper design of database tables and indexes, as well as the use of ongoing defragmentation steps,

will allow the Sybase DBA to maintain higher performance and retain the value of the hardware investment, rather than having to move to a new, higher level of hardware in order to maintain performance levels of an initially deployed production system.

Currently, most if not all RDBMS engines from various vendors do not automatically maintain compactness in their data structures, since the overhead of dynamic maintenance of data can be a burden on a busy production server.

Sybase has always provided a high-performance RDBMS server and continues to do so with its latest release—ASE version 12.5.

However, Sybase has also made advances in dynamic maintenance of the environment, starting with the Housekeeper task added in Sybase SQL Server System 11. This task helps keep database checkpoint times much lower than they were in older versions of SQL Server, and, starting with ASE version 11.9.2, the housekeeper task performs automatic, background cleanup of certain aspects of DOL (data-only locked) tables

Let's hope this is only the first step in a more proactive approach to the DBA's maintenance chores, such as automated data defragmentation housekeepers and index statistics gatherers.

This white paper illustrates how important table defragmentation is to achieving high performance in Sybase database servers. This importance is magnified in Sybase 11.0 and later versions, due to the availability of large I/O, and in Sybase ASE 11.9.2 and later which include the ability to perform OAM-based table scans on DOL tables using extent I/O. However, earlier versions also show increased performance following proper table defragmentation.

---

## What is Fragmentation?

At this point, we assume the reader knows the basics of Sybase data storage internals. Also, most readers of this paper (DBAs, rather than developers) will be interested in production Sybase database support issues.

However, developers may also be interested in the capacity planning and fragmentation-causing topics here.

For more detailed information on Sybase database structure, you can refer to the resources listed below:

- White Sands Technology, Inc.'s ProActive DBA User's Manual, Chapter 2 (Overview of SQL Server Data Storage)
- Sybase Adaptive Server Enterprise Performance & Tuning Guide, Chapter 3 (Data Storage) and Chapter 4 (How Indexes Work)
- Sybase Internals (Kirkwood, John; International Thomson Computer Press, 1996), Chapter 8 (Storage)

### Definition of Terms

The following section lists basic data structures and other terms that will be mentioned in this white paper.

*Sybase data storage-related terms.*

#### **Page**

Sometimes known as a block, a page is 2KB in size (or larger, up to 16K, in

Sybase ASE 12.5 and later versions). A data or index page holds one or more rows of data, and rows cannot span pages—that is, a row must exist on a single page.

### **Extent**

A group of 8 contiguous pages (or 7 pages if the first page number of the extent is a multiple of 256, where the first page in the extent is reserved for the allocation map page).

### **Extent Reclaim**

Occurs when all data pages on an extent become unused; the database extent is removed from use by its owning table and placed back into the free list of extents in the database.

### **Reserved Page**

A page within an extent which has been assigned to a specific table, but does not necessarily contain data yet.

### **Data Page**

A page within an extent which actually has rows on it and is not empty.

### **Unused Page**

A page which does not have rows on it but is a reserved page.

### **Overflow Page**

A type of data page maintained for nonunique clustered indexes, overflow pages hold duplicate index rows.

### **Cache Buffer**

A database page which is resident in the database server's cache.

### **Segment**

One or more logical chunks of a database typically used for table placement on specific disk drive(s).

### **Seek Time**

Time it takes a disk drive head to move to and start reading a requested piece of data.

## **Fragmentation Defined**

**Fragmentation** is defined as any condition which causes more than the optimal amount of disk I/O to be performed in accessing a table, or causes the disk I/Os that are performed to take longer than they optimally would.

Optimal performance of `SELECT` queries occurs when data pages are as contiguous as possible within the database, and the data pages are packed as fully as possible.

Note that achieving optimal `SELECT` performance can be at odds with maximizing performance of `INSERT`, `UPDATE` and `DELETE` statements, which will be discussed later in this paper (see page 6).

**Defragmentation** can be defined as the set of operations that a DBA performs to an RDBMS to remove any wasted space and make the storage space contiguous and well-ordered within individual database rows, pages, extents, segments, databases, and cache memory, all of which leads to better system resource usage and performance.

---

# Types of Fragmentation

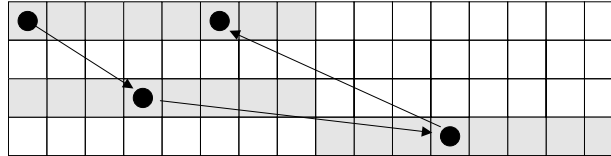
Fragmentation can manifest itself in different ways in the database. The different types of fragmentation are described in the following sections.

## Messy Page Chains

This condition occurs when the pages within the page chain of a table or index are not contiguous and in sequential order within the database. Messy page chains cause excessive disk seeking during full table or index scans and range queries (partial scans).

The drawing below illustrates discontinuous pages within a page chain:

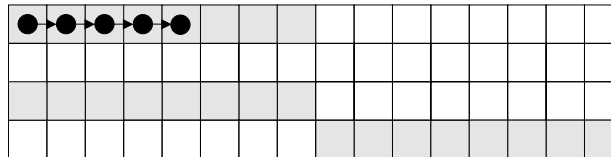
*Fragmented page chain.*



As you can see, the pages within the page chain are not physically contiguous (next to each other); thus, to read the pages in the page chain involves seeking the disk heads to each new page in the chain—a very time-consuming operation.

The page chain shown below, on the other hand, is in optimal order with no disk seeking required to move from page to page within the page chain—the I/Os can be performed sequentially, with no disk seeking:

*Properly organized page chain.*



A random I/O (defined as an I/O involving a disk seek) typically takes 10-20 times longer than a sequential I/O of the same size; thus, messy page chains can harm performance to a much greater degree than other types of fragmentation in objects on which full or partial scans (i.e. table scans or range queries) are performed.

Messy page chains can be caused by any of the following:

- Page splits due to inserts to a clustered index which is set up to spread randomly across the table;
- Page splits due to row growth (i.e. a row being updated to a larger row size);
- Pages being removed from the page chain due to heavy deletes, which leaves the page chain discontinuous between the pages surrounding the deleted page.

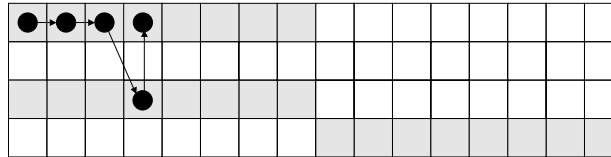
Data-only locked (DOL) tables in Sybase ASE 11.9.2 and greater do not utilize page chains in the data layer (heap) of tables. However, indexes (both DOL pseudo-clustered and nonclustered) utilize page chains, and messy page chains can harm performance for full or partial scans of these indexes, such as in covered queries which use the index.

As new database pages are allocated to satisfy new storage required by a table (e.g. due to inserts or page splits caused by rows which grow in size), the new

pages are allocated first from a list of unused pages within the table, and then from new free extents within the database segment.

However, the grabbing of random unused pages creates non-contiguous database pages spread out over the segment of the database, as you can see in the diagram below:

*Page splitting causes messy page chains.*



The third and fourth pages used to be contiguous within the page chain, but after the page was split, there is now a break in the contiguity of the page chain.

The randomness of data pages causes larger seek times when following a page chain during operations such as a table scan, index scan, and range queries. Larger seek times translate to slower response times for both long-running queries as well as OLTP operations.

*Measuring Page Chain Randomness*

In order to help you understand random page ordering, you can refer to ProActive DBA Visual Space Manager's (herein referred to as "Visual Space Manager") Page Chain Order report, which shows you the effects of poor page ordering and its effects on performance at various I/O sizes (2KB to 16KB). Visual Space Manager also includes higher-level reports which summarize your tables' and databases' overall fragmentation levels.

As an alternative, the DBCC `pglinkage` command can be used to traverse the data pages of the table, so you can get an idea of how well or poorly the page chain is ordered.

Microsoft SQL Server 6.0 and greater offers the DBCC `showcontig` command, which provides a report of certain types of fragmentation on a table or index.

To run this command, you must use an administrator's account such as `sa` and also have the `sybase_ts_role` granted to you. Typically, the DBA of the database server should be the one using the command.

First, use the query below to obtain the starting page number of the table's data page chain:

```
/* show print output */
dbcc traceon(3604)
go

/* find the starting page # of the table's heap or CI */
SELECT name, first
FROM sysindexes
WHERE indid in (0,1)
AND id = object_id(your_tablename)
go

name          first
----          -
Customer      14092
```

Next, issue the DBCC `pglinkage` command as shown below to walk the page chain of the table:

```
/*
```



```

* dbcc pglinkage (dbid, start_pg#, number_pgs,
*               print_option, search_for, search_order)
*
* dbid - the database id of the database being reviewed
* start_pg# - first page in the heap or CI (see above)
* number_pgs - how many pages to read (0=all)
* print_option - 0 = show count of pages read
*               1 = show last 16 pages read
*               2 = show all pages read
* search_for - page # to stop at (0=end of table)
* search_order - 0 = follow previous page pointer,
*                1 = follow next page pointer
*/
go

/* Example - show all page numbers, starting from first,
continue to end of table */

dbcc pglinkage (5, 14092, 0, 2, 0, 1)

go

```

This command will be time-consuming for large tables; look to use a limiting `number_pgs` value, such as 5000.

You will see the page numbers for all pages within the page chain, and you can determine how random the page linkage is and plan for table defragmentation accordingly.

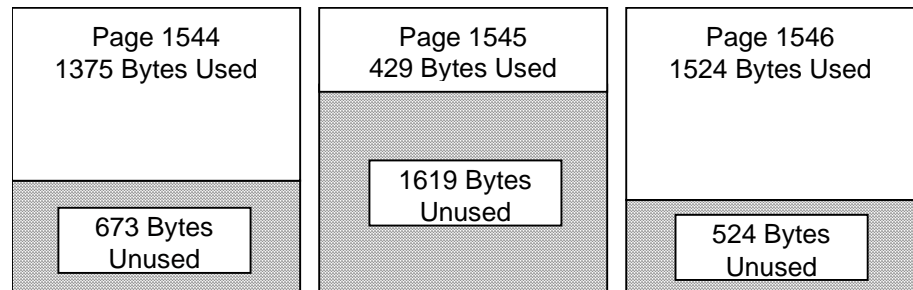
For a perfectly linked table, the pages in each extent should be linked together so that the page numbers increase by 1 within each extent. Every extent has a starting page number which is evenly divisible by 8.

Ideally, the extents will be contiguous as well, so that the page numbers will all be sequential (with the exception of breaks for allocation map pages, which occur every 256 pages).

## Poor Page Utilization

When pages are less than fully used, the part of each page that is unused constitutes a form of fragmentation, since the table's or index's rows are no longer packed together as tightly as they could be.

The drawing below illustrates poor page utilization:



The effect of poor page utilization is that more disk I/Os are required in order to read a table of a given size, due to the wasted space on each page, whereas fewer disk I/Os would be required if the pages were packed more fully with data.

Poor page utilization is caused by the following conditions:

- Random delete operations, which leave individual pages in use but not completely full of rows;
- Page splits due to inserts or updates, which leave each page approximately half-full;
- Row shrinkage of rows that contain nullable or variable-length columns;
- Placement of duplicate clustered index rows onto overflow pages, which are likely not to be fully utilized;
- Using a fillfactor of less than 100 when rebuilding an index on a read-only table or very large, lightly modified table; this will leave gaps within database pages which are unlikely to become filled in;
- Using very large row sizes, which prevent rows from being able to occupy all the space on a page; for example, if rows are 800 bytes each, then only two rows can fit on a 2K page, leaving approximately 400 bytes unused on every page.

*Fillfactor settings are used to intentionally alter page utilization levels.*

When you build a clustered index on an APL table using a low fillfactor setting, it results in fewer rows on the data pages than could otherwise fit.

This can have both advantages and disadvantages, and it is up to the DBA to weigh the benefits and drawbacks to determine the optimal fillfactor setting.

The main reason to use a fillfactor lower than 100% is so that, when performing randomized inserts across the entire table, rows that are inserted will not cause page splits immediately—in other words, each page will, on average, have room for a certain number of new rows to be inserted before a page split must occur.

Over time, of course, page splits will inevitably occur as individual pages fill up with newly-inserted rows, but using a fillfactor when you create or rebuild a clustered index can delay the onset of page splitting, in effect buying you some time before it becomes cost-effective to rebuild the index again.

On the other hand, setting the table's fillfactor too low results in excessive wasted space on pages, which negatively impacts the performance of `SELECT` queries against the table due to the extra disk I/Os that are required to read the selected rows.

These tradeoffs can be summarized as follows:

Fillfactor Setting	Pro	Con
Higher Fillfactor	Best utilization of page space.	Random inserts more likely to result in page splits, harming performance of inserts and updates, and causing excessive disk seeking.
Lower Fillfactor	Page space is less effectively utilized, meaning more disk I/O is required for large <code>SELECT</code> queries.	Inserts spread across the table are less likely to cause page splitting.

In general, page splitting harms performance to a greater degree than having a certain amount of unused space within rows and pages, because the extra disk I/O and disk seeking incurred by page splits outweighs the slight additional cost of performing extra disk I/Os (if those I/Os are sequential) due to lower page space utilization. Thus, use of a fillfactor lower than 100% should be strongly considered for those tables that will receive a moderate to high degree of random inserts across the range of the clustered index.

If the table's clustered index is **not** designed so as to spread out inserts across the table's data pages, then the wasted space which exists due the low fillfactor setting only results in fewer rows being locked while page locks are in effect, but will confer no benefits in terms of reducing page splitting. In such cases, a fillfactor of 100% is recommended unless you need to use a lower setting in order to reduce contention due to page locking.

The DBA must also understand that using a low fillfactor can also result in more rows being considered by the query optimizer during query optimization, and may even lead to indexes not being chosen on tables due to the larger amount of data pages within the table.

In general, we recommend that a fillfactor of either 100 or at least 90 percent be used unless carefully planned. However, this formula may not apply to smaller, highly-accessed tables, because points of contention (blocking) should be addressed with a fillfactor or the use of the dirty-read feature within Sybase.

Applications built around the use of low fillfactors should probably be redesigned based on the newer features of SQL Server 11.0 and later, or use of row-level locking features available starting with ASE 11.9.2 should be considered.

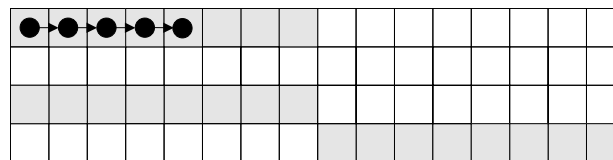
Visual Space Manager's Object Information Report lets you view a table's page utilization level in detail, so you can keep an eye on page splitting and other causes of poor page utilization.

## Extent Fragmentation

Similar to page fragmentation, extent fragmentation occurs when the extents of a table or index are not contiguous with the database. This can occur due to:

- Random deletes throughout a table, which leave some of the pages in an extent unused while the extent itself is still reserved as part of the table's space allocation.
- Deletes on ranges of contiguous rows within the table, causing one or more entire extents to become deallocated, thus leaving a gap between the surrounding extents of the table or index;
- Interspersion of a table's data extents with its own indexes' extents or with the extents of other tables, resulting in excessive disk seeking.

The diagram below illustrates extent fragmentation:



The areas highlighted in gray represent the extents allocated to a particular table in the database. As you can see, the table's extents are not contiguous within the

database, but rather are intermingled with the extents of other objects and with empty extents.

Extent fragmentation is very common in Sybase. It occurs, for example, when two or more tables are built in parallel, thus causing new extent allocations to alternate between the two tables; or, when a table which has indexes is loaded with data, new extent allocations alternate between the table's data layer and its indexes, causing the extents for each object to become interleaved with the other(s).

You can reduce the amount of extent interleaving on tables by allocating selected tables to their own segments in the database, so that when the tables grow, new extent allocations will not be interleaved with the allocations of other tables.

## Row Fragmentation

This type of fragmentation occurs when columns in a tables are wider than needed to store application data. For instance, if a customer table has a column of CUST\_NAME with a datatype of CHAR(30) NOT NULL, then any right-padding spaces in this column would be considered as fragmentation since they take up space but do not apply to the application specifically.

Changing this column to a VARCHAR(30) NOT NULL datatype would cause only the usable data to be stored, but not the trailing spaces. This is more space efficient, and the developer could even plan more proactively for larger names and use VARCHAR(255) NOT NULL as the datatype since only the name is stored and not the spaces.

The tradeoff in making this change is that if the customer name changes to a longer string, page splitting can occur, which results in lower page utilization as well as page chain fragmentation.

In general, page splitting harms performance to a greater degree than a certain amount of unused space within rows and pages, so for columns that are likely to grow in size, it is often better to use a fixed-length datatype which will not grow, rather than a variable-length datatype whose data can grow and cause a page split. In this case, the extra space used by the fixed-length column harms performance less than the extra I/Os and inevitable disk seeking caused by page splits.

But, it may make more sense to use variable-length datatypes for columns that will not grow or change in size once the rows have been inserted. This will realize the space-saving benefits of variable-length columns, yet will not allow row growth or shrinkage which can increase the fragmentation level in the table.

---

## Dealing With Fragmentation

Now that we understand the different types of fragmentation that can occur, let us turn to what can be done to reduce the occurrence of fragmentation in your databases, and how to cope with fragmentation if and when it does occur.

With the proper planning during the application development and table design phase, and by planning appropriate maintenance schedules for ongoing database maintenance, DBAs can achieve the proper levels of space efficiency and reduced fragmentation that will result in optimal performance of the database as a whole.

The following sections list techniques you can use as part of your strategy for keeping fragmentation-related performance degradation at a minimum.

## Analyzing Effects of Deletes

The design of tables in Sybase should include the analysis of whether random deletes to the table will occur. Since Sybase databases only reclaim space at the extent level, not the page level, the DBA and developer should understand the condition of page fragmentation.

When a row is deleted, its space is not utilized by new inserts unless the same, or a very similar, clustered index value is used. For tables without clustered indexes (i.e. heap tables), new inserts are only applied to the last data page of the table. Unused deleted row space within data pages on tables without clustered indexes is never reclaimed.

The effect of this is to reduce overall page utilization in the table, which means more disk I/Os are required in order to scan a given number of rows in the table, than would otherwise be required if the pages were as full as possible.

## Analyzing Effects of Inserts

As discussed earlier (see page 7), inserts to APL tables with clustered indexes can cause page splitting, which is a serious degrader of performance. Analysis should be done to determine what level of inserts will occur against the larger and/or more heavily used tables in the database, and where those inserts will occur.

If the inserts will occur primarily at the end of the table, it is usually best to use a higher fillfactor setting (or even 100%), and use the `dbcc tune(ascinserts)` setting to prevent 50-50 page splitting (see below).

Tables with random insert points mixed with random delete points will experience the most fragmentation if a clustered index is used to cause the random inserts. This is because page splits occur on data pages when no more room is found on the page but the clustered index value falls between two existing values.

*Over time, random inserts cause "50-50" page splitting.*

When this occurs, the Sybase server creates two half-full pages out of the original page, and the new row being inserted is placed on one of the two split pages.

Subsequently, the data that used to occupy 2048 bytes (on a single page) now consumes 4096 bytes in two pages with about 50 percent full data pages. Over time, if the table is inserted into and deleted from either heavily or lightly, the table becomes heavily fragmented at the database page level.

When using database tables with clustered indexes that allow for all inserts to go to the last database page, the DBA can allow for the use of the `DBCC TUNE` feature with the "ascending inserts" option, to allow the page split to occur at the end of the data page, rather than the middle of the data page. The following table is one that could benefit from using `DBCC TUNE`:

```
CREATE TABLE Customers
  (Customer_Number NUMERIC(10,0) IDENTITY
   NOT NULL, Customer_data...)

CREATE INDEX cidx ON Customers (Customer_Number)
```

As new records are added to this table, application developers will expect the rows to be added in numerically increasing order based on the identity attribute of the `Customer_Number` column.

Using the following command in Sybase versions 11 and higher, the DBA can force page splits to occur at the offset where the row would be inserted:

```
DBCC TUNE(ascinserts, 1, Customer_Number)
```

To turn off ascending inserts, the following command would be issued:

```
DBCC TUNE(ascinserts, 0, Customer_Number)
```

It is best to use this command when the data in the table is inserted in sequential order based on the clustered index, or if the clustered index of the table is a composite index and the **last** column increases monotonically.

In the latter case, even though not all inserting is done at the end of the table, if enough of the inserts will be in sequence at a small number of insert points in the table, it can be beneficial to force page splits to occur at the end of the page. Some fragmentation will result, but overall, less fragmentation will occur than if pages were split using the normal “50-50” method.

**NOTE**—This command is a **runtime-only** command and must be re-issued upon dataserver startup. The DBA may want to create a startup batch which is run following dataserver startup to set certain tables up for ascending inserts. In addition, this command can be used prior to a large BCP operation.

Sybase 11.0 as well as prior versions all benefit from defragmenting to remove database page fragmentation. More data on a database page means the possibility of fewer logical and physical I/Os during database disk usage, as well as lower requirements of data cache size. Data cache can keep more rows of data in memory with fewer cache buffers (data pages in memory) if the pages are more fully packed with rows.

## Analyzing Effects of Updates

Like random inserts, updates which cause rows to grow in size can cause page splitting on APL tables. On DOL tables, these updates can cause another problem known as *row forwarding*, where Sybase relocates the grown row on a new database, but leaves a marker at the row’s old location which indicates the new location of the row.

Both of these conditions harm performance by causing extra disk I/O and disk seeking, and thus developers and DBAs should work together to avoid these conditions as much as possible.

You can prevent rows from growing in size by using only fixed-length, non-nullable column datatypes. Obviously, this is not feasible in all situations, but DBAs and developers alike should be aware of the tradeoffs in using variable-length and nullable datatypes.

Variable-length types can save storage space if used wisely; yet, they do incur additional overhead, which can end up wasting space if they are used indiscriminately.

But perhaps the worst problem of all is that, when a variable-length column grows in size, it can cause a page split, which itself not only takes much longer to execute, but also causes fragmentation, leading to poorer performance for subsequent queries that have to scan that part of the table.

Thus, special care should be taken to ensure that variable-length columns are used only where the following conditions are met:

- There is a significant spread between the average and maximum column length; and,
- The maximum column length is reasonably large (e.g. 5-10 characters or more), so the variable-length column overhead does not outweigh the potential space savings.

## Preventing Extent Fragmentation

When a table undergoes a variety of operations, data pages may become empty within an extent of the table. The extent continues to store data on some of its pages, while other pages are marked as unused (these pages show up in the output of `sp_spaceused` under the `unused column`).

Since these unused pages are never reclaimed by Sybase unless new inserts are done which cause a new page requirement, they contribute to the overall unused space of the table. These pages add up over time, and use disk space which otherwise would be available for other tables.

Extents are deallocated from a table when all data pages of the extent become unused. As the last row of a data page is removed, the extent is reviewed to see if that page is the last one in the extent with data on it, and if so, the whole extent is removed from the table's allocation map and returned to the free list of extents within the database.

Larger amounts of unused pages for a table should be reclaimed by the database in order to provide the pages to other tables on the database segment. The operations of defragmentation will reclaim this space. With the large-block I/O offered starting in SQL Server 11.0, extent fragmentation becomes much more significant a performance problem, because entire extents with unused pages are read into cache but only those pages with data on them will be utilized by the server during the query that fetched the extent into cache.

## Capacity Planning for Defragmentation

When planning a Sybase database environment, most DBAs and development teams plan for expected data sizes as well as affording space for fault tolerance in the form of mirroring or RAID configurations.

However, space considerations for table defragmentation are often overlooked. Typically, capacity planning should include allocating database segments to defragmentation areas to allow tables to move from one database segment to another when defragmentation occurs.

For instance, the following tables show you how to allow for space planning for a large table.

Database Name	Size
Customer_History	5GB

Segment Name	Size	Usage
history_seg_01	1GB	Customer_History data only
history_seg_02	1GB	Customer_History data only
ncidx_seg_01	1GB	Nonclustered indexes only
ncidx_seg_02	1GB	Nonclustered indexes only
default	500MB	
system, log	500MB	

Table	Typical Size	Initially Assigned to Segment
Customer_History	800MB	history_seg_01

The steps to defragment this table would be as follows:

- Drop Customer\_History non-clustered indexes
- Drop Customer\_History clustered index
- Rebuild Customer\_History clustered index on history\_seg\_02 (if on 01 today)
- Use fillfactor of 100% (plan accordingly, depending on usage)
- Rebuild Customer\_History non-clustered indexes on ncidx\_seg\_02 (if on 01 today)
- For nonclustered indexes, use a fillfactor of 90% (plan accordingly, depending on usage)
- Optionally execute DBCC Tune(ascinserts, 1, tablename)
- Execute sp\_recompile "Customer\_History"

The effects of properly allocating space for defragmentation are:

- The nonclustered indexes are kept separate from the data layer of the table (heap or clustered index), thereby reducing fragmentation at the data layer;
- The data layer and the nonclustered indexes can be rebuilt onto a clean segment (either \_01 or \_02, whichever it is not currently using), so the page allocations will be perfectly contiguous with no extent interleaving fragmentation.

When rebuilding a clustered index, consider the space requirements within the segments being used for the new clustered index. You will need space to accept new data pages which will hold the rows being moved from the older clustered index.

Typically, the space requirements will be equal to or less than the current size of the table (for example, as shown in the data column of sp\_spaceused for the table).



The reason the space requirement may be less is that unused page space may exist in the table—for example, if a table's clustered index currently uses 500MB, it may only require 400MB after defragmentation.

A good estimate is based on the first clustered index rebuild done on the table. Compute the row count and divide by the number of pages shown in the data column of `sp_spaceused` after the clustered index is created. Use that to judge the average number of rows per page and continue to use that average number for further space estimates when rebuilding clustered indexes. The steps would be:

- Defragment the table to pack the pages as fully as possible
- Run `sp_spaceused tablename`
- Compute `rowtotal / data / 2`
- Catalog the result as the table's average rows-per-page value

When defragmenting a non-clustered index, you simply need to drop and recreate it. Many times, it will be smaller than its original size, as reported by:

```
sp_spaceused table_name, 1
```

## Use Fixed-Length Row Sizes

If you try to avoid as much as possible variable-length datatypes and datatypes which allow NULL values which later change to non-NULL values, you will prevent row size growth, which can cause page splitting on updates or row forwarding on DOL tables—i.e. fragmentation.

Row size growth can also lead to overflow page usage. If a duplicate row cannot fit back onto the page it was updated on, a new page will be created for the page split, causing data page fragmentation.

For smaller `VARCHAR NOT NULL` datatypes, consider `CHAR NOT NULL`. A rule of thumb could be to use `CHAR` for 10-byte and smaller character columns, and `VARCHAR` for ones larger than 10 bytes, where column size is not deterministic.

Visual Space Manager's Row Information Report lets you examine a table's average row size versus its defined minimum and maximum row size. If the average size is close to the maximum, consider replacing variable-length columns with fixed-length equivalents.

## Use Monotonically-Increasing Clustered Index Keys

Create tables with clustered indexes which increase in value over time—such as identity-based clustered index columns or a date/time value based on the current date and time. This will help reduce fragmentation due to page splitting because new rows will always be added to the end of the table.

Some DBAs do not like to do this because they consider the last page of the table a point of contention which will cause blocking. This is only a problem if users have multi-step transactions which take excessive amounts of time to insert onto the last page of a table.

Applications may be able to have such transactions broken out into multiple transactions if the unit of work is not critically dependent on such bundling of transaction activity.

If many users are updating the last page of the table, the typical point of contention is the speed of transaction log writes during the commit phase of the

transaction. Such contentions can be alleviated with faster transaction log disks or even Solid State Devices for log segments.

Also, in version 11.5 and greater, proper sizing of the PLC (also called ULC, or user log cache) can improve transaction commit throughput, thus alleviating last-page contention on clustered indexes with monotonically-increasing index key values.

## Use Unique Clustered Index Keys

Where possible, you should only use unique-valued clustered indexes to keep overflow page usage down.

When a significant number of duplicate key values exist in a clustered index, Sybase creates overflow pages to hold the duplicate rows. This lowers space efficiency and index performance for two reasons:

- The duplicate rows are not part of the B-tree index structure and thus require extra disk I/O for the server to locate these rows within the index;
- The overflow pages are often not packed very fully, and may contain only one or several rows; this reduces overall page utilization and cache efficiency, and increases fragmentation.

The best way to eliminate overflow pages in a clustered index is to create the clustered index on a unique column or columns in the table. As an alternative, you can often reduce the level of overflow pages in a clustered index by adding one or more columns to the index to increase the level of uniqueness of the index.

## Large I/O Considerations

Starting with Sybase SQL Server version 11.0, multiple pages can be read at one time into I/O pools configured in the default cache and any DBA-defined data caches.

Because the ability to effectively use large I/O depends on the data page chain being contiguous, the DBA should pay attention to the output of `sp_sysmon` in SQL Server 11.0.1 and greater.

*Measure large I/O effectiveness with `sp_sysmon`.*

Running `sp_sysmon` at 10 minute intervals while large I/O operations are being done will give the DBA an idea of how efficient the page chains are. The numbers to review are the comparison between the large I/Os that are done and number of pages used.

If the percentage of pages used is below 60 percent, consider examining the table's fragmentation level—for example, via Visual Space Manager's Fragmentation Summary Report, or by using the `DBCC pglinkage` function to determine the number of contiguous pages in the table; then, plan a table defragmentation for those tables and indexes being scanned with larger I/Os that could benefit from defragmentation.

Visual Space Manager's Fragmentation/Performance Report shows the effects of fragmentation in terms of disk seeking and increased disk I/O levels when using regular 2K disk I/O versus the larger 16K I/O size. This can help you identify tables that could benefit from the larger I/O size, as well as tables that could effectively utilize larger I/O if defragmented.

---

## The Big Picture

With the proper amount of table defragmentation, the Sybase DBA can proactively keep a database server environment operating in top condition.

Sybase continues to review operating procedures of ASE and may offer built-in table defragmentation operations in future version of ASE.

However, in today's customer environments, the database can only offer those features which are available today.

In our experience, it is not uncommon for defragmentation to reclaim database space of 30-40% or more. For example, one customer saw savings in the 6-7 GB range following defragmentation of 14GB of tables which had not had their indexes rebuilt in over a year.

Subsequently, overall user performance was enhanced on their Sybase 11.0 system, especially for table scans using 16KB I/Os.

Other customers have reported performance increases of between 5-50% or more in batch and on-line processes running against Sybase databases after defragmenting critical tables in heavily-used databases.

With the data pages in a contiguous format, all eight pages of each extent could be used following a 16KB I/O, and disk seeking was reduced during table scans due to the extents now being contiguous.

Many of the latest disk drive arrays and controllers contain read-ahead caches. Such caches may perform more physical I/O than requested if they "predict" that additional data will be required—for example, for a 2KB I/O request, a read-ahead cache may read 16KB or 32KB from disk.

This means that a properly defragmented table will allow for a much higher throughput for such caching environments since many of the reads done following an initial 2KB read will be satisfied by the disk controller's RAM cache, instead of wasting (discarding) the pre-read blocks of data.

It can allow random read rates which might typically occur at 100KB to 200KB per second to be increased to over 1MB and sometimes 2MB per second, even when performing 2 KB I/Os during a table scan on Sybase 11.0.x and older versions.

Sybase versions 11.0 and greater can also utilize the cache better if the cache does pre-reads of data blocks of 16KB or larger.

Finally, Database Consistency Checker (DBCC) execution times are dramatically improved by defragmentation of tables.

Checking a table with DBCC incurs reads of all data pages within a table, similar to a table scan.

Speeding up of DBCCs allows for more end-user availability to the tables as the shared locks placed on the tables being checked do not occur for as long a period. Checking of non-clustered indexes is also sped up by defragmenting them along with the data portion of the table, though some sites can skip running DBCCs against non-clustered indexes and, in the event of a problem within the non-clustered index, the index can be rebuilt.

---

## Conclusion

Hopefully you now have a better understanding of what causes fragmentation in Sybase, and the various options you have at your disposal for dealing with it when it occurs.

As we have seen, with the right tools and the right information, you are no longer at the mercy of fragmentation.

For further information on the White Sands Technology, Inc. products mentioned in this white paper, including ProActive DBA Visual Space Manager, contact us at:

Voice: 1-818-702-9200

Fax: 1-818-702-9100

Web: [www.whitesands.com](http://www.whitesands.com)

Email: [sales@whitesands.com](mailto:sales@whitesands.com)

Finally, we at White Sands would like to thank you for reading this white paper, and we hope you find it beneficial in tuning and maintaining your Sybase databases.

If you have any feedback regarding this white paper, feel free to contact us using any of the above methods.

---

# Index

- Bulk copy, 11
- Clustered indexes
  - duplicate values in, 7
  - monotonically-increasing, 14
  - randomizing inserts in, 8
  - unique, 15
- Datatypes
  - fixed vs variable-length, 9
- DBCC
  - improving performance of, 16
  - pglinkage command, 5
  - showcontig command, 5
  - tune command, 10
- Defragmentation, 3
  - capacity requirements, 12
- Deletes
  - effects on fragmentation, 4, 7, 8, 10
- Disaster Recovery Toolset, 17
- Disk I/O
  - sequential vs random, 4
- Disk seeking, 3
- DOL tables, 4
- Duplicate values
  - in clustered index, 7
- Extent, 3
  - fragmentation, 8
- Fillfactor, 7, 8, 10
- Fragmentation defined, 3
- Inserts
  - effects on fragmentation, 7, 10
- Large I/O, 15, 16
- Messy page chain, 4
- Overflow pages, 3, 7, 14
  - preventing, 15
- Page chain
  - fragmentation in, 4
- Page splits, 4, 7, 8, 10, 11
- Page types, 2
- Page utilization, 6
- ProActive DBA, See "Visual Space Manager"
- Random I/O, 4
- Row fragmentation, 9
- Row size
  - and fragmentation, 7
  - growth, 4
  - preventing growth in, 11
- Segment, 3
- Sequential I/O, 4
- Updates
  - effects on fragmentation, 7, 11
- Variable-length datatypes, 9, 11, 14
- Visual Space Manager, 5, 8, 14, 15, 17
- White Sands Technology, Inc.
  - contacting, 17