Sybase Data Storage & Fragmentation

Software Gems Pty Ltd
Derek Asirvadem

V2.5
04 Sep 12

Introduction

SOFTWARE GEMS

Purpose

1 This *Software Gems* document defines the physical elements of a *Sybase* ASE database; assists in the understanding the terminology in the manuals, and the operation of ASE. Indeed, it overcomes the problem of abysmal manuals in that subject matter.

2 There is an awful lot of shallow, inaccurate, misleading and false information on the Internet. Unfortunately some of that false or misleading information is published by *Sybase*, both in the manuals, and on the web. This document is therefore rendered to provide full and complete information (albeit very condensed), such that the reader is no longer vulnerable to false or confusing information on the subject.

Structure

This document combines three closely related HTML documents into a single PDF, and resolves the links. It remains in three Parts, with a single numbering scheme (19 chapters) throughout (Levels are numbered in Roman numerals). When it is relevant, the section presents APL vs DPL/DRL LockSchemes separately. The definitions are Normalised, and cross-referenced. Virtually all objects can be selected, to open further detail.

Sybase Data Storage

The elements of data storage units, their relations, and their types. *This is a pre-requisite to the second part*.

Sybase Fragmentation

Definition & identification of the three distinct levels of fragmentation & the types within them; determination of each level/type; followed by chapters for each level/type

Education

- This document is actually a consolidated version of a selection of the Memory Tag pages from our courses.
  - We do not provide ordinary SQL and *Sybase* courses, there are many providers.
  - However, as true performance experts, we provide **specialist** *Sybase* Quality & Performance courses at both the DBA and Developer level, which allow you to take full advantage of your software investment.
  - We also provide high performance, standard-compliant Relational Database Design and education.
  - There is no substitute for formal, qualified education. Please inquire if you need further detail, or you have an interest in improving your *Sybase* performance or SQL coding.
- As such, they are detailed, very condensed and complete, but of course, the scope is limited.

Manual

These documents are provided to complement the *Sybase* manuals, and to correct them, as follows:

- they contain information that is not in the manuals (ie. they overcome the lack of information)
- where the manuals contain contradictory information, the correct version *only*, is provided, *the goal is to eliminate confusion and half-truths !*
- where misleading or false technical terms are used, *correct technical terms are used instead*
- they bring all the relevant information about a subject together, in one place

Document Status

What was once a few single pages made available on the web, due to interaction with the *Sybase* community, has been consolidated into a single document, and expanded. It remains a collection of diagrams from our course documents, a terse, condensed, diagrammatic style; rather than one of our usual polished final documents, that some of your have come to expect. Progress (adding diagrams and explanatory test) is made between assignments, based on questions and feedback received.

Version

V2.0 12 Sep 11 Consolidation of three previous docs; full exposition to 14 pages; first open publication; enabled HTML Image Map
V2.5 28 Mar 12 Data Storage (now 9p); Definition & Determination added (8p); Fragmentation (now 12p); PDF version (now 31p).
It is valid for *Sybase* ASE versions 12.5.4.x and 15.x. Yes.

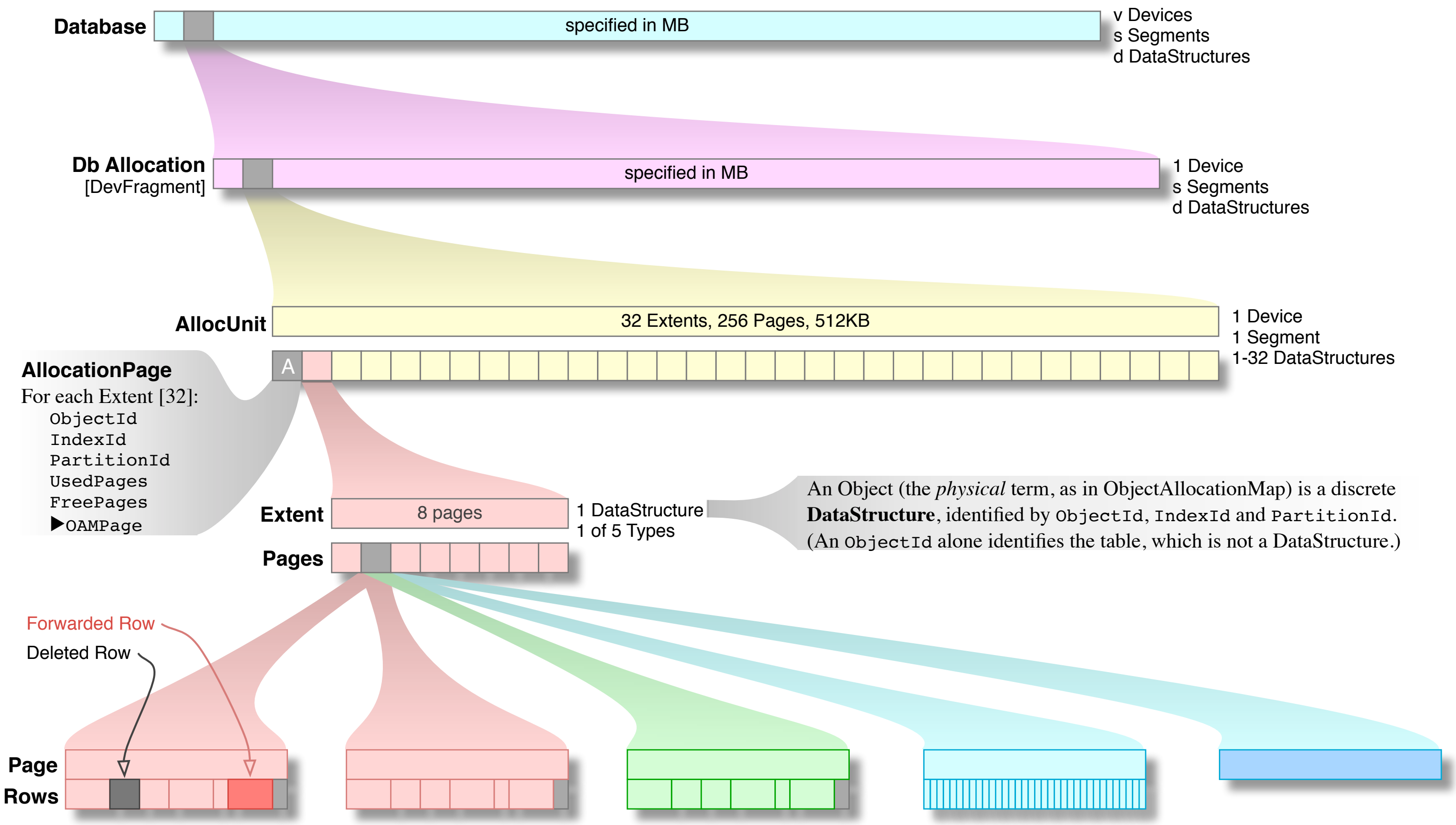
Copyright

Moral Right & Contact

The author is Derek Asirvadem, Information Architect and *Sybase* performance specialist, he is solely responsible for the content. He welcomes constructive commentary and answers questions for professionals (click the link at the bottom of the page).

1 Data Storage Unit

First we need to understand the different Data Storage Elements, what they contain, how they relate to each other, and their Units of Measure. This is presented in its natural hierarchy, from top to bottom, largest to smallest, and identifies the Pages used to control space management.



DOL Heap (Always)
- RowIds do not change
- DELETES Marked but not Removed
- Expanded Rows Forwarded
- Interspersed INSERTS at end
- No Clustered Index

APL Heap (When No CI)
- Rows shifted on Expand/ Contract/INSERT/DELETE
- Chronological Order
- INSERTS at end

Clustered Index
- Index/Row Order maintained
- Rows shifted on Expand/ Contract/INSERT/DELETE
- Heap eliminated
- Page Splits when Full for interspersed INSERTS

Nonclustered Index (including **Placement Index**)
- Index Entries & RowId
- Entries shifted on INSERT/DELETE

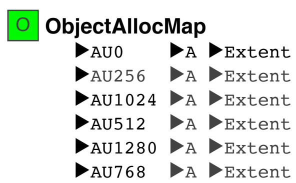
Text/Image Chain
- The entries are the content of a single Row/Column
- Allocated in units of Pages

1.1 AllocationPage

- The first page of each AllocationUnit contains the AllocationPage, it identifies:
  - the 32 Extents that it contains
  - the Physical DataStructure residing in each Extent (identified by `ObjectId, IndexId` and `PartitionId`)
  - pointers to the OAMPages of those 32 Physical Datastructure, and
  - the space available in each Extents, and in each Page of each Extent.

1.2 ObjectAllocationMap

- Just as the first Page of an AllocationUnit is the AllocationPage, the first Page of a DataStructure is the ObjectAllocationMap
- It contains a linked list of the AllocationUnits in which Extents belonging to the DataStructure reside.
  - The AllocationPage of each AllocationUnit is then interrogated to locate the Extent.
  - The AllocationPage identifies which Extents & Pages have free space. If such exists, this allows rows in the DataStructure to be placed close to other rows, however it is quite independent of rows in other DataStructures.
- If more than one Page is required for the OAM, a linked list of OAMs is provided
- While the OAM provides a second access path to the DataStructure, it is especially relied upon during Table Scans of DOL Heaps, since they do not have PageChains.



1.3 Other Control

In order to administer *Sybase ASE*, the above Data Storage units need to be understood, and they are covered in detail in the following pages. In order to complete the picture, however, there are two more Pages that are used to manage space efficiently (these are not expanded):

- **GlobalAllocationMap**
  Contains space usage bits (Used/Free) for all AllocationUnits in the database
- **PartitionControlPage**
  Each Partition has an additional Page identifying free space

2 DataStructure

SOFTWARE GEMS

This chapter introduces *Sybase* ASE DataStructures, again in logical order, and illustrates how they relate to each other.

The catalogue tables may be easier to understand if they had been named:
- ~~sysindexes~~ `sysLogicalStruct`
- ~~syspartitions~~ `sysPhysicalStruct`

1. **Table**
   - a Table has a single entry in `sysobjects WHERE type U`
   - the Primary Key is (`id`), as in `OBJECT_ID()` or `ObjectId`
   - a Table is a collection of Logical DataStructures
2. **Logical DataStructure**
   - each Logical DataStructure has a single entry in `sysindexes`, which defines its *logical* structure, keys, etc
   - the Primary Key is (`id, indid`), `indid` identifies the DataStructure Type
   - There are five types of Logical DataStructure (the APL Heap and DOL Heap are very different, as detailed in the next chapter):

`sysobjects.id/U`	Table				
LockScheme	DPL/DRL	APL		Any	
Logical DataStructure Type	DOL Heap • Always	APL Heap • Only when no CI	Clustered Index • Eliminates the Heap	Nonclustered Index	Text/Image Chain • one for all Text/Image columns in the *table*
Allowed	1	1 Heap xor 1 Clustered Index		249	1
`sysindexes.indid`	`0` means Heap (No CI)	`0` means Heap (No CI)	`1` means CI (No Heap)	`2 to 250` means NCI	`255` means Text/Image Chain

3. **Physical DataStructure**
   - each Logical DataStructure is rendered physically as one or more Physical DataStructures
     - the Heap or Clustered Index, which contains data rows, may be divided into several Physical DataStructures, called **Partitions**
     - the Nonclustered Index and Text/Image Chain are not Partitioned

`sysindexes.indid`	Partitions	Partitions	Partitions	Nonclustered Index	Text/Image Chain

   - each Physical DataStructure has a single entry in `syspartitions`, which defines its *physical* structure, **Data Storage** location, etc
     - *hence the silliness in the manuals that "unpartitioned objects have one partition"*
   - the Primary Key is (`id, indid, partitionid`)

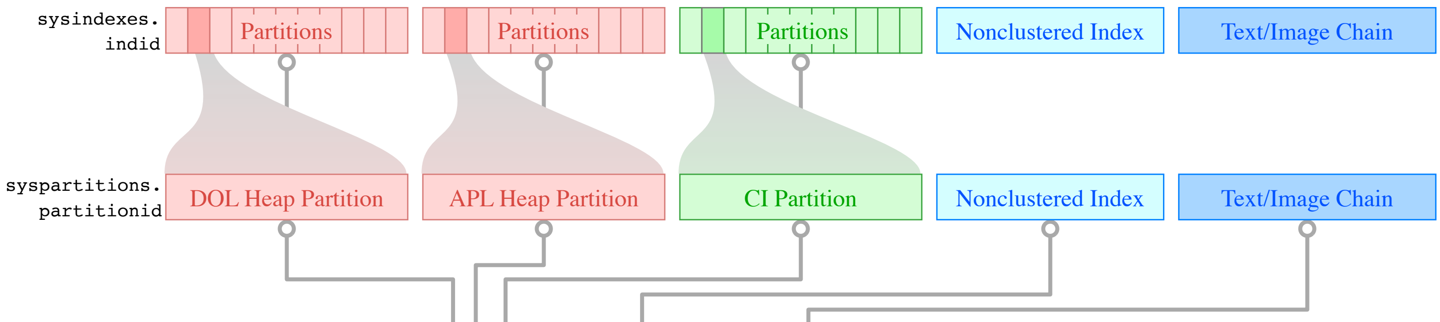
`syspartitions.partitionid`	DOL Heap Partition	APL Heap Partition	CI Partition	Nonclustered Index	Text/Image Chain

4. **Partitioned DataStructure**
   There are, therefore, five types of Physical DataStructure, and the Heap or the CI may be Partitioned.
5. In summary, a DataStructure is
   - an independent Data Storage structure that is
     - first, belongs to a Table (`ObjectId`)
     - second, one of five logical types (`IndexId`)
     - third, a physical structure, which may be a Partition (`PartitionId`)

During the discussion of logical or physical DataStructures, non-technical terms such as
'table',
'base table' and
'object-index pair'
are too ambiguous to be meaningful: those who use them are committed to your continued confusion.

2 DataStructure

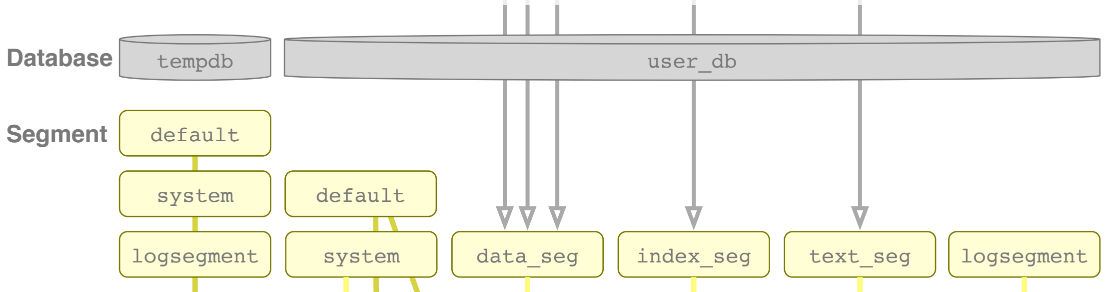
6. The five types of Physical DataStructure, the first three of which *may* be Partitioned, are located on Devices, which are identified by Segment:



2.1 Segment

A Segment [1] is a logical group of one or more Devices, within a database. A good Segment Plan has two fundamental purposes:

1. It allows DataStructures to be distributed for load balancing purposes:
   - separating the data (CI or Heap) of *a single table* from its related NCIs
   - separating the *different tables* within a Transaction
   - separating the Partitions of a table, in order to support full parallelism
2. It drastically reduces Level I and II Fragmentation, which would otherwise be massive.
3. Either a Logical DataStructure (all Partitions in the DataStructure) or a Physical DataStructure (a single Partition) may be **placed** on a Segment.
   - placing all the Partitions of a DataStructure on one Segment/Device has the same I/O contention as an unpartitioned DataStructure (shown)
   - placing each Partition of a DataStructure on a separate Segment/Device eliminates that contention, and maximises parallelism (not shown)



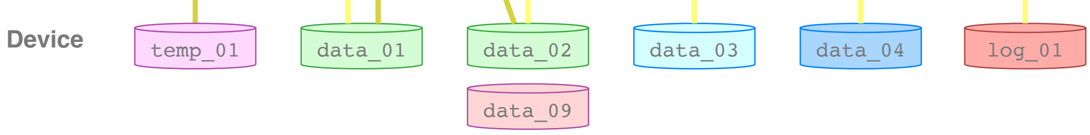
- `default` *is not a* `segment`*: it is the* `segment` *one has when one does not have* `segments`.
- *Much like* `public` *is not a* `group`*:* it is *the* `group` *one belongs to when one has no* `group`.
- *Or one is the number of* partitions *in an unpartitioned DataStructture*.

2.2 Device

A *Sybase* Device [1] is one of the following. Note that ASE *treats* it as a contiguous set of disk blocks:

- File
- Raw Partition
- Logical Volume (SAN or Volume Manager), which is a File or Raw Partition

Devices are server-level resources: part or all of a Device is allocated to a single Database.



- *Each Device is a separate I/O queue within the server*
- *It is therefore best to use neither too few Devices, nor too many, based on the size of each database*.

1. This is an introduction to Segments and Devices; it is not a full exposition.

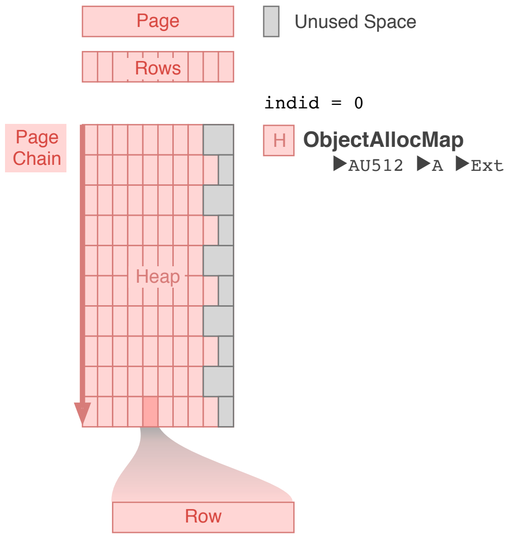
3.1 Heap

This chapter discusses the APL Heap and the DOL Heap, and their characteristics.

AllPage Locked

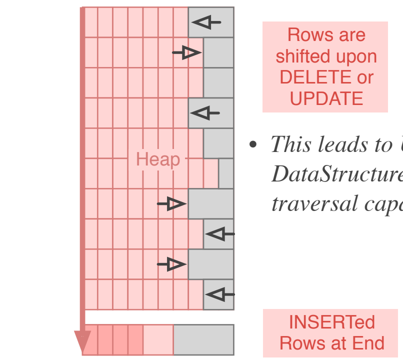
Heap (When No Clustered Index) Fresh

- All the Nonclustered Indices belonging to an APL table are **Clustered Index based** (RowIds may change); there are Heap-based (RowIds are static) only when the CI is absent
- The creation of a Clustered Index eliminates the Heap; dropping the CI returns the Heap
- This illustrates a Heap, which occurs only when the Clustered Index has been actively avoided
- Except when used as 'pipes' or 'queues', APL tables should always have a Clustered Index



- Table scans via PageChain
- `INSERTS` are placed at the end of the Heap
- Pages are kept trim; rows are contiguous
  - Rows within the Page are shifted upon `DELETE` and `UPDATE` (Row Expansion/Contraction)
- Row Expansion may cause it to be moved to the end of the Heap, changing the RowId)
  - If there are NCIs, the RowIds need to be updated

Heap (When No Clustered Index) Fragmented

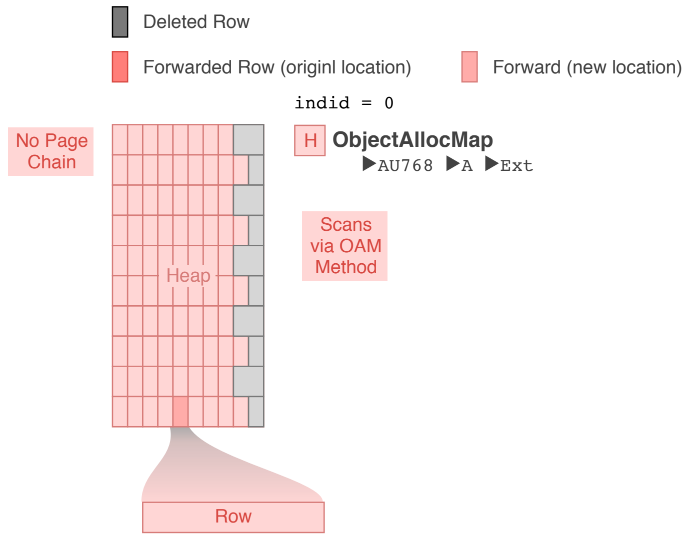


- *This leads to Unused Space, but the DataStructure retains its speed and traversal capability*

DataPage/DataRow Locked

Heap (Always) Fresh

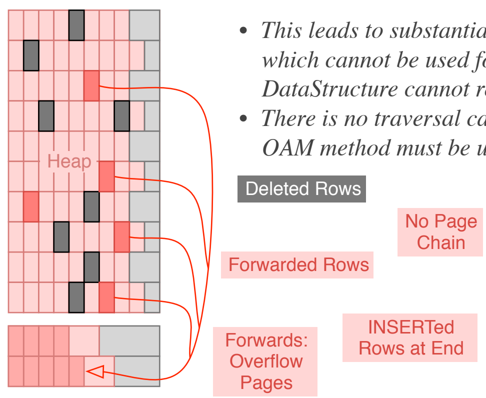
- DOL tables always have a Heap
- RowIds do not move, they are **Static** (except during `REORG` of course)
  - All the Nonclustered Indices (including the Placement Index) belonging to a DOL table are **Heap (or Static RowId) based**
  - It is a mistake to view the DOL Heap as PI based, since all the NCIs (including the PI) are *dependent on the Heap*, not other the way around. The NCIs cannot change because the Heap cannot be changed.
- By design, the Heap and any NCIs (including the PI) are logically and physically separated, in order to reduce dependencies



- Table scans via OAM method only
- RowIds do not change
  - Deleted rows are marked for delete but not deleted (they are deleted, and the space is reclaimed, during `REORG` or aggressive Garbage Collection)
  - If space is available in the current Page or Extent of the Heap (as a result of reserving same), the Forwarded Row or interspersed `INSERT` is placed there; otherwise (the usual case) it is placed at the end of the Heap. The intended and actual locations are nowhere "near" the original location and nowhere "near the Placement Index, refer to section [**8.3**] and [**9.5**]. Forwards accumulate in **Overflow Pages**.
  - When a row is Forwarded, the NCIs (including the PI) must access the original location, to obtain the forward address, then access the Forwarded Row.
  - Contracted Rows are *not* repatriated

Heap (Always) Fragmented

- *This leads to substantial Unused Space, which cannot be used for new rows; the DataStructure cannot retains its speed*
- *There is no traversal capability, the OAM method must be used*

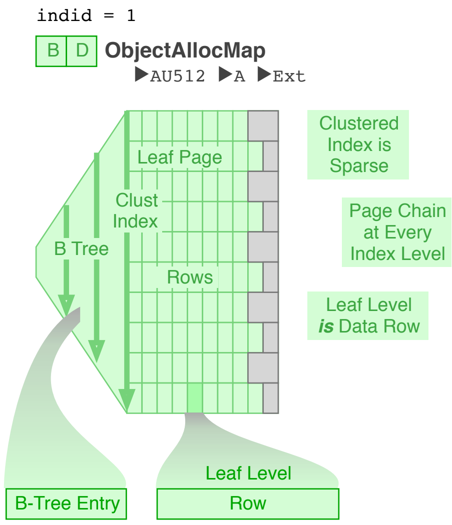


3.2 Clustered Index

This section discusses the Clustered Index, and its characteristics.

AllPage Locked

Clustered Index (Heap Eliminated)



- The Index B-Tree is **clustered** with the data rows, into a single DataStructure
  - The Leaf level of the B-Tree **is** the data row (put another way, there is no Leaf level, the B-Tree is *clustered* with the data rows)
  - Creation of the CI eliminates the Heap; dropping the CI returns the Heap
  - One less logical Read on every access
  - There are still two OAMs to allow independent access
- All the DataStructures belonging to an APL table are **Clustered Index based**
  - Index order = Row Order
  - Rows are distributed as per Index Key, and remain so
- Designed for
  - **Relational Keys** (compound or composite keys)
  - **Range Queries**
- `INSERTS` into Key location:
  - For Interspersed `INSERTS`, if the page is full, a Page Split is necessary, and the RowIds (in the split Page) which are referenced in any NCIs must be updated
- Pages are kept trimmed
  - On Expand/Contract/`INSERT/DELETE` Rows in the CI may be shifted within a Page, without additional overhead, maintaining free space in the page

- According to the Relational Model, rows in a table must be unique. The Clustered Index is designed for Relational tables, and to be unique, and therefore should be
  - Non-unique keys cause **Overflow Pages** .

A man and a woman are meant to be married; together they achieve more than each achieves separately. Implementing APL tables without a Clustered Index, is analogous to a divorced couple. Likewise, there is no fidelity in non-unique Clustered Indices .

DataPage/DataRow Locked

(None)

- Despite the demanded "clustered" syntax, there is no such thing as a DOL "clustered" index or DOL "clustered" table. The DataStructure addressed in is fact a **Placement Index**.
- There is nothing remotely like the Clustered Index available for DOL tables.

Confirmation

If anyone suggests that DOL "clustered" indices do exist, run this **simple query** on a database that has both APL Clustered Indices and DOL "clustered" indices. Study the **DataStructure** chapter, along with the report, and ask them why, as far as *Sybase ASE* internally is concerned:

- Clustered Indices always appear *without* a Heap
- Heaps always appear *without* a Clustered Index
- Placement Indices are Nonclustered Indices
- Placement Indices always appear *with* a Heap (which means they are two separate Logical, and therefore Physical, DataStructures)

Such persons evidently have little technical knowledge os *Sybase*.

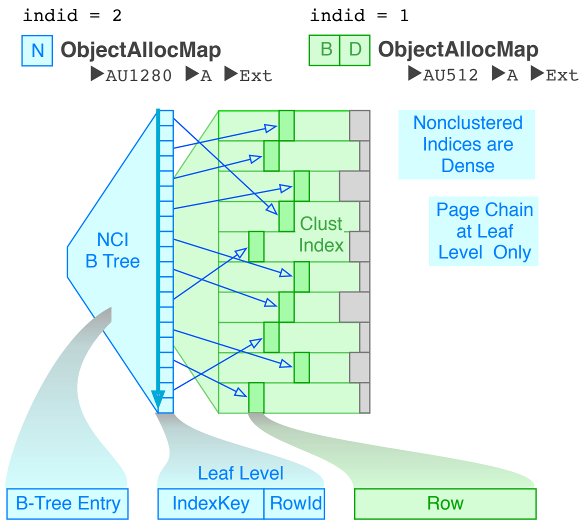
All the technical evidence from all the functions and catalogue components, is consistent. Even a simple query demonstrates the truth. It can be extended to show other items as desired.

3.3 Nonclustered Index

This chapter discusses the Nonclustered Index, and its characteristics under the different LockSchemes.

AllPage Locked

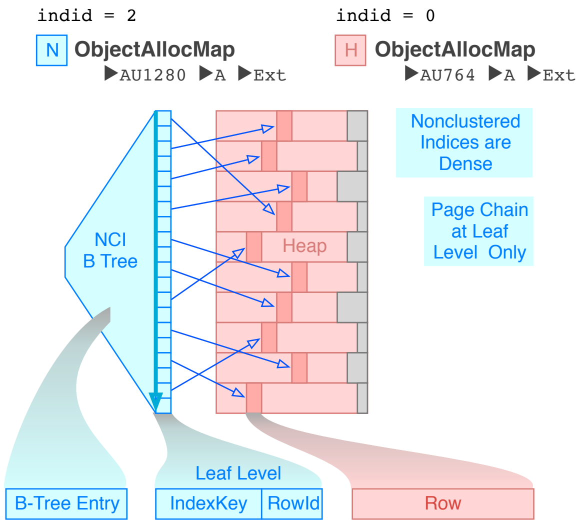
Nonclustered Index



- If there is no space available in the NCI for interspersed `INSERTS`, the Index Page must be split.
  - This disturbs the PageChain
- The NCI conatins the RowId in the CI; when the row moves (as the CI is re-ordered and kept trim), the NCIs need to be updated.

DataPage/DataRow Locked

Nonclustered Index



- If there is no space available in the NCI for interspersed `INSERTS`, the Index Page must be split.
  - This disturbs the PageChain
- The **Placement Index** is a Nonclustered Index, with a couple of additional attributes.
- The NCI conatins the RowId in the Heap; the rows do not move, and so there is nothing to update in the NCIs (including the PI). This is better stated as, in order to eliminate updating the NCIs, the rows in the Heap are designed to be static.

3.4 Placement Index

This chapter discusses the Placement Index, and its characteristics.

AllPage Locked

(None)

There is no equivalent on the APL side. A rough equivalent would be:

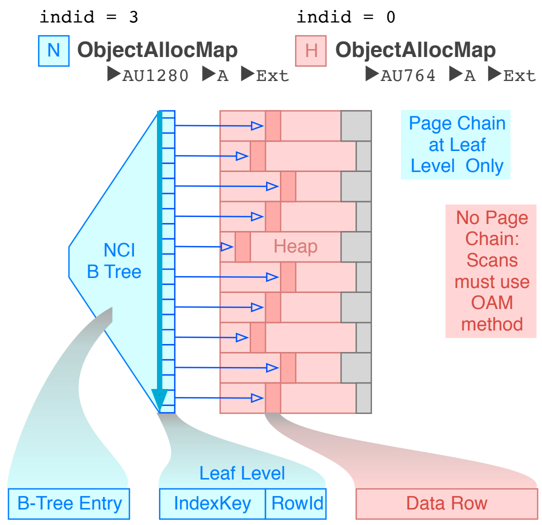
- a **Heap** (ie. where a Clustered Index has been actively avoided, thereby crippling it).
  - but even then the APL Heap has a PageChain, providing faster scans
- plus a **Nonclustered** Index

*The **Placement Index** is not comparable to a Clustered Index, which is available only for APL*

- *It has no clustering (as per the definition of that term since 1984); the B-Tree is not clustered with the data rows, forming a single physical DataStructure; it remains a separate DataStructure to the Heap*
- *There is no such thing as a DOL "clustered" Index*
- *The use of the term "clustered" Index in relation to DOL tables is therefore incorrect, confusing, and fraudulent.*
- *The correct term, as per some, but not all, Sybase documentation, is Placement Index*
- *Unfortunately, to address the Placement Index or the Heap, one is required to use the "clustered" syntax. Talk about forced confusion.*

DataPage/DataRow Locked

Heap (Always) & Placement Index Fresh



DOL tables always have a **Heap**. They may have a single Placement Index. It is a **Nonclustered Index** (there is no structural difference), a separate DataStructure to the Heap, with two additional criteria:

1. It identifies the **initial placement** of rows in the Heap
2. Any settings made, such as placement `ON segment` and `FILLFACTOR`, apply to the Heap as well.

As such, its relationship to the Heap is *slightly* closer than that of other NCIs, but that does not constitute clustering ala Clustered Index; a term which existed before its advent;. Note that they are separate by design.

- This initial row placement is not maintained under:
  - interspersed `INSERTS`
  - `DELETES` and
  - `UPDATES` that cause Row Expansion
- The Index & Heap remain two separate DataStructures; two OAMs
  - Two Logical Reads on every access (via any NCI, including the PI)
- Key order in each NCI is maintained, but Row order in the Heap *cannot* be maintained
  - The **Heap is Static RowId based**.
  - Other than to rebuild the Heap, there is no value in a Placement Index
- **Range Queries** are not possible, since it is not a Clustered Index (there is no order to the Heap, and it does not have a PageChain).
- Ideal for non-relational Keys (surrogates, monotonic)

DOL tables have an additional **third level of Fragmentation**, they get fragmented at this level very quickly, and require regular `REORG`. The above illustrates a fresh, unfragmented Heap and Placement Index; section [18] illustrates a fragmented Heap and Placement Index.

Deeper Understanding, Less Irrelevant Work

Consider this. Since:

- Given that Range Queries are not supported, there is ***no value*** in the Placement of rows in the Heap, or maintaining the order of the rows
- Whatever placement is obtained by `DROP/CREATE INDEX`, is lost as soon as ordinary DML commences

therefore the placement intended by the Placement Index is actually quite irrelevant, and can be dispensed with. This merely eliminates the confusion, and the small mountain of false expectations heaped upon it.

The issue that remains, that does matter, is fragmentation, since it hinders Asynch Pre-Fetch and Large I/O efficiency and consumes unused space. When the Heap becomes fragmented enough to warrant it, de-fragment it by creating *and dropping* a Placement Index (realising its the fleeting value, which is to identify some order *when rebuilding the Heap*). This method is usually much faster than `REORG REBUILD`, even though the `WITH SORTED_DATA` qualifier cannot be used, since the data in the Heap is not in any order.

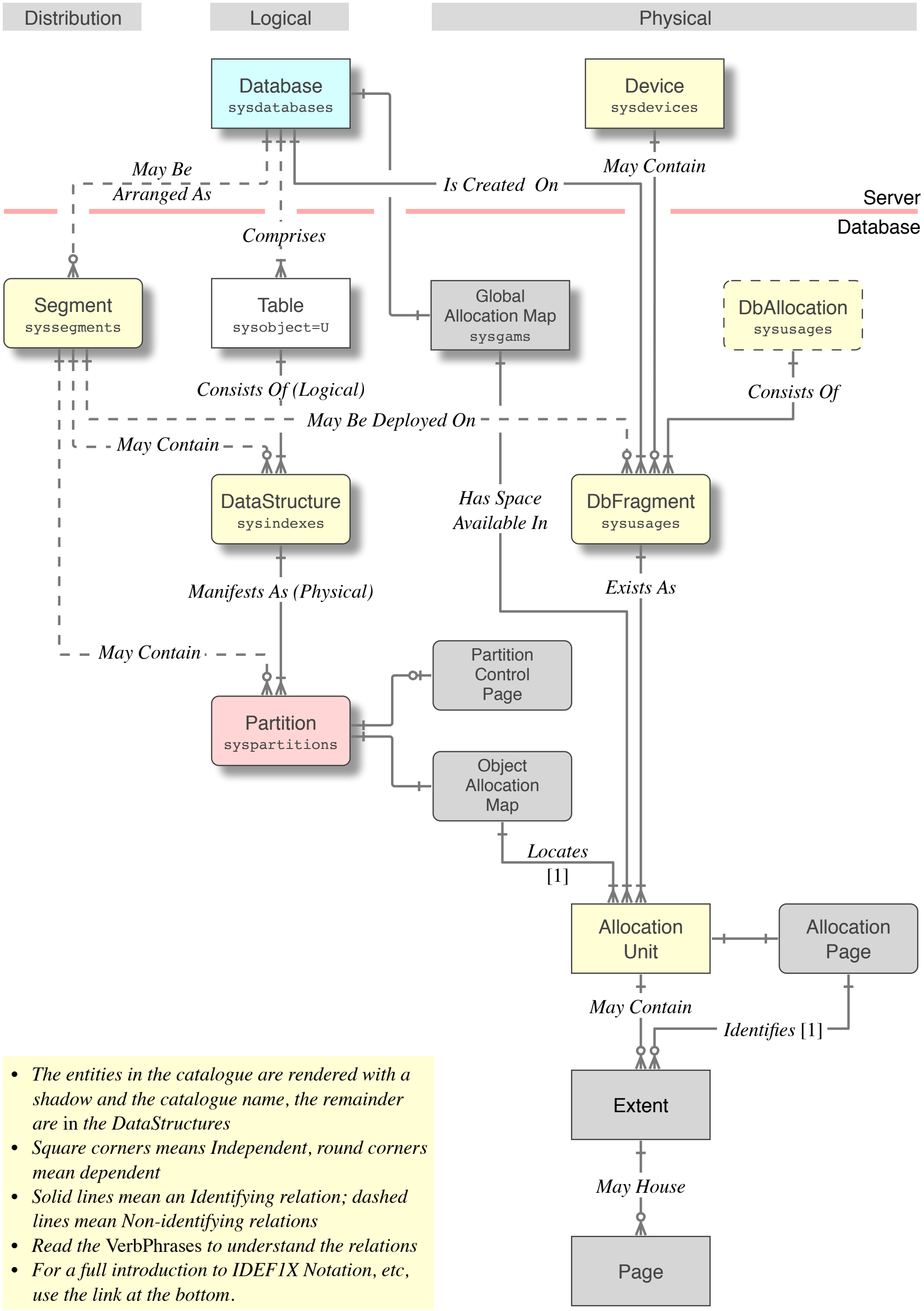
Placement Index Key

Since Range Queries cannot be supported, and the order cannot be maintained, the index that is chosen for the Placement Index is actually quite irrelevant. The candidate index that explicitly identifies, or implies, a chronological order is best, since it groups the most frequently updated rows away from the least frequently updated rows.

4 Data Model • Catalogue

SOFTWARE GEMS

A formal Relational Data Model is the best way to understand data, and its relations. This chapter presents the entities in the catalogue that pertain to Data Storage elements, in terms of a formal Data Model (Entity Relation level), rendered in IDEF1X. Specifically, it shows the catalogue in which information about each Data Storage Unit is stored.



Devices and Databases are server objects

The GAM is a server level entity, however it is located in the Database catalogue

The Database Allocation is the collection of Database Fragments

The size of Database Fragments is automatically set, based on the `ALTER DATABASE` *request versus the space availability and location. The smaller the Fragment, the more the database is fragmented at Level I.*

As discussed above, the physical manifestation of a DataStructure is one or more Partitions.

A Page or Extent number that is divisble by 256 is an AllocationUnit, containing an AllocationPage and up to 32 Extents.

A Page number that is divisble by 8 is an Extent, containing a single DataStructure. Contrary to the manuals, an Extent contains only one DataStructure

The atomic unit of Storage, and of I/O. Asynch Pre-Fetch can read an Extent or AllocationUnit in a single request.

- *The entities in the catalogue are rendered with a shadow and the catalogue name, the remainder are in the DataStructures*
- *Square corners means Independent, round corners mean dependent*
- *Solid lines mean an Identifying relation; dashed lines mean Non-identifying relations*
- *Read the* VerbPhrases *to understand the relations*
- *For a full introduction to IDEF1X Notation, etc, use the link at the bottom.*

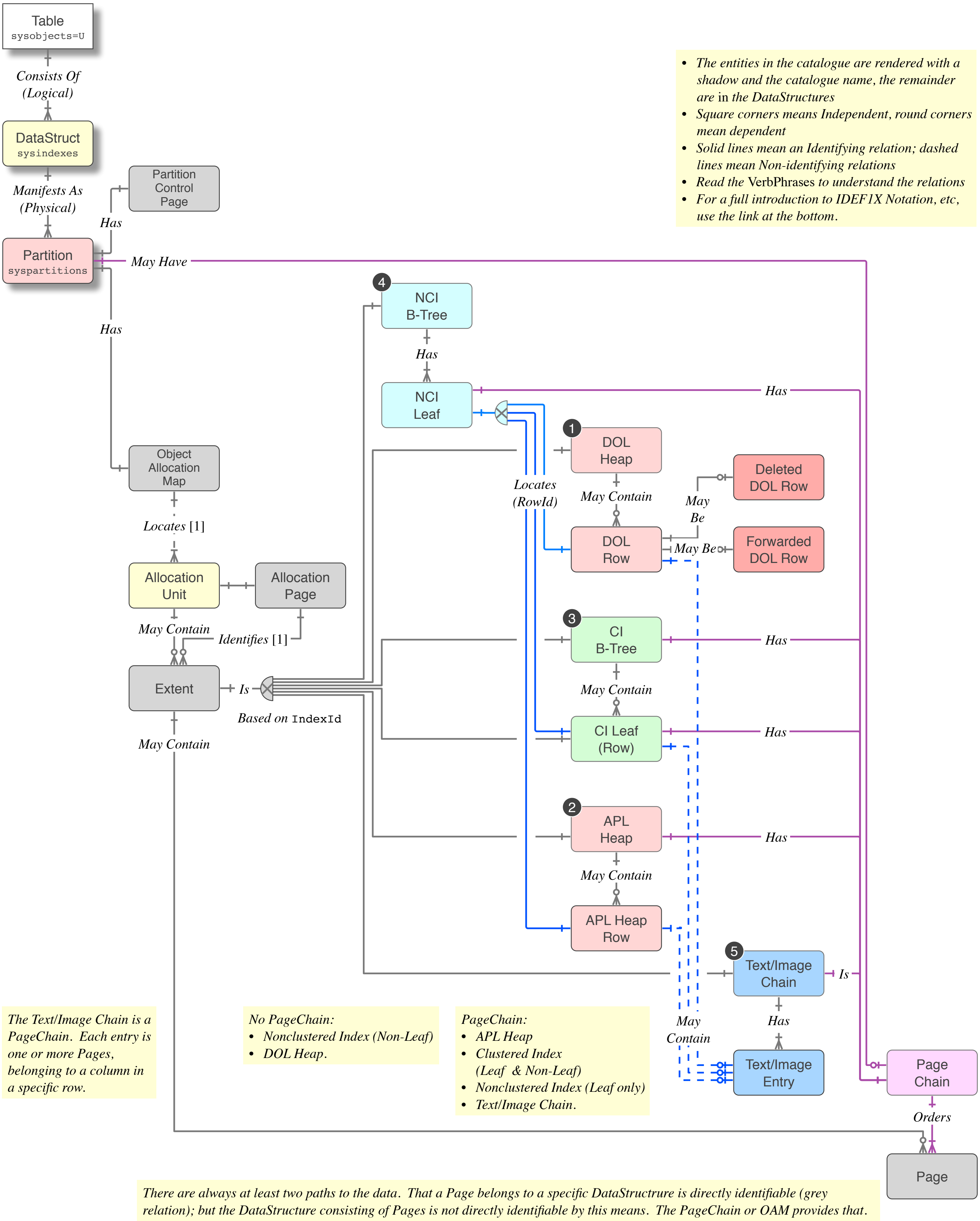
This models the normal case: exceptional cases, such as the mandatory `logsegment`*, which may or may not be correctly deployed, are not differentiated.*

IDEF1X Notation

1. Additionally: *Has Space Available In*.

5 Data Model • DataStructure

This chapter exposes the five types of DataStructures, starting from the catalogue, in terms of a formal Data Model (ER level).



IDEF1X Notation

1. Additionally: *Has Space Available In*.

6 Definition

SOFTWARE GEMS

This document defines and discusses all aspects of Fragmentation, in substantial detail (albeit condensed) as it occurs in *Sybase ASE*.
The document is laid out as follows:

- this introduction, containing **definitions** and approach
- the **impact** of fragmented DataStructures
- Definition of every **Type** of Fragmentation, within each of the three Levels
- four sections identifying how Fragmentation can be **determined** accurately, and without confusion, fully detailed
- a section on **evaluation** of the various determinants
- an additional section of issues relating to **Partitioned** DataStructures
- eleven sections discussing the different Types of Fragmentation within each Level, fully illustrated and discussed

In particular, the level of detail provides information so that Fragmentation can be fully understood and therefore prevented, and leads up to why common methods of correcting Fragmentation do not work. Put another way, the detail identifies why Fragmentation must be addressed using an overall approach, at all three levels, if substantial performance gains are sought. It is not a point problem, and therefore point solutions do not apply.

Understanding the **Data Storage** structures that *Sybase* uses, is a pre-requisite to understanding Fragmentation.

- A table does not exist physically, it exists as a collection of **Physical DataStructures**: when a query is executed, it is the DataStrucures that belong to the table that are accessed. In order to administer tables efficiently, the DataStructures and how they are accessed, must be clearly understood.

Level

The three Levels of Fragmentation are quite **independent** of each other, and can be differentiated easily. It is quite possible for a DataStructure to be fragmented at one Level and free of Fragmentation at another Level: indeed, each Level requires quite different **correction** operations, and they affect only that Level. The highest performance is obtained when all three levels are addressed.

Frequency

The frequency of correction operations for each Level, is also different:

- Level III de-fragmentation (`REORG REBUILD` or `DROP/CREATE CI or "CI"`) is required weekly at a minimum.
- Level II is dependent on
  a. whether a good Segment plan has been implemented, and
  b. the turnover within the DataStructure.
  The frequency required varies from monthly to annually. A good Segment plan and a well designed Clustered Index may well eliminate the need for de-fragmentation altogether.
- Level I de-fragmentation is required once, if it is done properly. It provides
  a. the basis for reduced fragmentation at Level II
  b. reduced frequency of Level II de-fragmentation operations, because it renders the correction operations at Level II more permanent.

It is normal to de-fragment a DataStructure at Level II because it is demanded presently, but to leave a full de-fragmentation operation of Level I to a separate maintenance window, addressing many DataStructures together, because it requires reasonable planning and the scripts require testing, etc.

What it is Not

Administrators are sometimes confused by the masses of misinformation either available on the internet, or presented by Storage Teams who are avoiding work, or hardware salesmen who are selling something on the false basis that it will result in less work for the DBA. To address this, it is important to understand what Fragmentation is not:

- ***Hardware Striping equals Fragmentation***
  The SAN (or Logical Volume Manager) and *Sybase ASE* are completely independent of each other. ASE *treats* the Logical Volume as a contiguous series of disk blocks. Whether the LV is striped or not is irrelevant to ASE; Fragmentation; performance; etc. Striping affects only the speed of the LV within the hardware unit. De-fragmentation operations within ASE reclaims performance within ASE.
- ***If you use a SAN, you don't need Segments***
  See above. Total lack of technical ability and logic. *My father works 50 hours a week, therefore your father does not need to work*.
- ***Partitions equals Fragmentation***
  When the Partitions of a table (Physical DataStructure) are placed on several Devices or Segments, for performance purposes, by design, it is *distribution* not *fragmentation*, and the result is substantially different to the fragmentation that occurs when there is no design.
- ***Data Distribution equals Fragmentation***
  Substantial performance can be gained in Relational tables when the Key (usually composite Keys) is used to distribute the data [1], and therefore decrease contention. That is again, by design, and space must be reserved for interspersed `INSERTS`. Such reserved space is not the same as unused or waste space, which cannot be used for interspersed `INSERTS`.

What it Is

Level I

Database Fragmentation: the unplanned or unconscious occupation of space, and the disturbed contiguity, of DataStructures across the Database.

Level II

DataStructure Fragmentation: the unplanned or unconscious occupation of space, and the disturbed contiguity, within the DataStructures.

Level III

Page Fragmentation: the unplanned or unconscious occupation of space, and the disturbed contiguity, within the DataStructures, in systems that have been implemented quickly and without OLTP Standards or Relational technology.

1. That is not possible in record filing systems, where surrogate keys (single-column; monotontic) are used across the board.

6.1 Impact

This document is written for the qualified *Sybase* Database Administrator, and the subject is Fragmentation. As such, it does not detail how the I/O subsystem; disk resources; caches and their configuration; etc, operate. It is expected the the reader understands all that, and therefore appreciates the relevance of maintaining DataStructures in an un-fragmented state. However, there are basic features within *Sybase ASE*, that are commonly unappreciated and therefore unused. It is a shame that in many sites, *Sybase* operates at a mere fraction of the speed that it is capable of.

Two such features that are fundamental to *ASE* delivering great speed when accessing the DataStructures, are described here.

Asynch Pre-Fetch

is the **mechanism**, the set of methods that enables *Sybase ASE* to read large amounts of data, *in anticipation of* the query requirement.

- Asynch Pre-Fetch reads:
  - Min 8 Pages (1 Extent)
    - (Min for Covered NCI Scan is 2 Rows)
  - Max 256 Pages (1 AllocationUnit or 32 Extents)
    - (the first Page/Extent uses 2K I/O, due to it being an AllocPage)
- Asynch Pre-Fetch is requested for Table Scans, Range Scans, Covered NCI Scans, DBCC, and Recovery
- It has a self-modulating **Look-Ahead Set** which:
  - prevents it from saturating the I/O subsystem, and
  - prevents it from reading large numbers of Extents or Pages that will not be used.

  The modulation is based on the extent of success/failure of previous APF attempts on the DataStructure.
- Due to ASEs brilliant architecture, Asynch Pre-Fetch operates independent of the Caches and PoolSizes, and concerns itself with Buffers; and subsequently, Pages used.

Large I/O

refers to the **resources** used by Asynch Pre-Fetch. When large Buffers are requested, the (a) specifc Cache and (b) the available PoolSizes come into play. The integrity of resident Buffers may cause denials: Pages or Extents within the requested Buffer may already exist in a smaller PoolSize; the PoolSize requested may not be present; etc. Therefore Large I/O statistics relate to Caches and PoolSizes (not Buffers).

The impact of fragmentation is usually a subjective issue: people are used to a certain level of response from their queries, when the database contains a somewhat higher population than it did during the initial testing, the response slows down. It is an awareness that is quite real, but unscientific.

- the loss of speed is certainly the result of naïve server installation and configuration, and a lack of planning and configuration at the Device and Segment levels
- that loss of speed is not necessary: the server and its resources can be configured, such that response does not slow down with population, even with very large tables [2]
- that subjectivity is relevant only in the absence of science and knowledge; chapter [7] details the accurate determination of fragmentation, such that science and knowledge can de used instead of subjectivity
- the initial value of that subjective sense of speed is actually quite low (since the query did not enjoy the benefit of proper configuration, and thus the use of Asynch Pre-Fetch and Large I/O), and therefore the users are in reality comparing 'slow' with 'very slow' on the scale of possible speed; they have never enjoyed 'fast' and they do not know what they are missing.

Level I

Correcting Level I Fragmentation returns great speed to the DataStructures, due to enabling Asynch Pre-Fetch and Large I/O to their maximum extents. It allows *Sybase* to operate at the 'fast' end of the possible speed spectrum. Further, it contains and therefore reduces the extent of Level II Fragmentation [3].

Level II

Most DBAs are aware of some of the aspects of Level II Fragmentation, and how to correct it. There are some traps for young players, as detailed in chapter [9], ignorance of which will cause de-fragmentation operations to be very transient, to have no persistence. However, without an awareness of Level I, the baseline speed is 'slow' and the frequency of de-fragmentation operations is increased.

Level III

This is mainly the consequence of storing unnormalised spreadsheets in a database container, as opposed to storing Normalised Relational tables. One has to live with the consequences of such actions, and deal with the myriad problems, such as fragmentation of a new order; frequent and offline maintenance of DataStructures; reduced concurrency (increased contention); increased number of locks; etc.

Performance & Tuning

- APF is generally automatic (one need not do anything to invoke it)
- Large I/O is possible if a large PoolSize is configured for the Cache
- Resources for both APF and Large I/O are fully configurable, monitored in detail, and can be tuned at several levels.
- Sysmon reports statistics for both the APF mechanism and the Large I/O resources.
- the low usage of these facilities is always due to fragmentation at Level I or Level II or both. Correcting that fragmentation returns great speed to the DataStructures.

2. Contrary to most articles on the web, *Sybase* is quite capable of high speed on very large tables. Archiving history data onto a separate database; the consequent requirement to modify code (to look in two places for one thing); the maintenace of an archive database; the loss of DRI, are all quite unnecessary.
3. Software Gems provides a *High Performance Sybase Configuration*, that ensures the server is operating as the highest levels of performance. We also provide a complete Device & Segment [re-]configuration, such that Level I issues are eliminated. Both on a fixed price, guaranteed result basis.

Intro Unit DataStruct Defn II Determ III Determ I AllocUnit I Segment II PageChain II Unused III Page

6.2 Fragmentation Type

SOFTWARE GEMS

It is convenient when the Type identifies the exact *location* of the Fragmentation within the Database or DataStructure; other forms of identifying the Type are confusing. In order to fully understand the three Levels of Fragmentation, and types of Fragmentation within each Level, let us look at the best and worst scenarios in each Level and Type. Your DataStructures will be either one or the other, there is no 'in-between'; however, after correction operations using an overall plan have commenced, the DataStructures will move into that 'in-between' zone.

Level	Location/Type	Applies		Condition	Result	Correction
I	AllocationUnit • AllocUnits across the Db • Extents within AllocUnits	APL	DOL	**Best** • AllocUnits of a DataSructure spread across the smallest range • Extents of a DataSructure spread across the fewest AllocUnits [10.3] • Each AllocUnit contains the fewest DataStructures	• Highest level of APF: AllocUnits; Extents as required • LIO structures heavily used • Fewest I/Os required to read the DataStucture	
				Worst • AllocUnits of a DataSructure spread across the largest range. • Extents of a DataSructure spread across the most AllocUnits; on the most Devices; across the database [8.3]. • Each AllocUnit contains the most DataStructures	• Loss of APF • LIO Structures not used • More I/Os required to read the DataStucture	• Separate the tables within a transaction • Separate DataStructs in a table from each other • Rebuild DataStructs in a *fresh* location
II	PageChain	APL	[4]	**Best** Contiguous PageChain [12.1]	• Level I modulated [5] • No interrupts during scans	
				Worst Disturbed PageChain, spread across Extents & AllocUnits [12.2]	• Level I modulated [5] • More interrupts during scans	
	OverflowPage • Duplicate Rows	APL		**Best** No Non-unique CIs	Prevention of insanity	
				Worst High percentage of duplicated CI 'keys' [13]	Additional I/O for duplicated 'keys'	Implement an Unique CI [6]
	OverflowPage • Forwards		DOL	**Best** No Forwards	Substantially faster queries	
				Worst High percentage of Forwards [13, 17]	Additional I/O for Forwarded rows	Fxed length rows or `REORG REBUILD` or `DROP/CREATE "CI"`
	Unused Space Extent [7]	APL	DOL	**Best** No Unused Pages per Extent	• Level I modulated [5] • Highest level of APF &	
				Worst High percentage of Unused Pages per Extent [14]	• Level I modulated [5] • APF & LIO scaled back	`DROP/CREATE CI` or `"CI"`
	Unused Space Page [7]	APL	DOL	**Best** No Unused space per Page	• Level I modulated [5] • Fewest I/Os required	
				Worst High percentage of Unused space per Page [15]	• Level I modulated [5] • More I/Os required	`DROP/CREATE CI` or `"CI"`
III	Page (Heap)		DOL	**Best** No Forwards & Deletes	• Level I modulated [5] • Fewest I/Os required	
				Worst High percentage of Forwards & Deletes [17]	• Level I modulated [5] • Additional I/O for Forwards & Deletes • Creates Unused Space/Page	`REORG REBUILD` or `DROP/CREATE "CI"`

4. The DOL Heap (containing the data rows), has no PageChain; all scans must use the OAM method
5. The same Result identified at Level I, modulated to the scope identified by Location/Type (the row).
6. Duplicate rows (Keys) are illegal in Relational Databases.
7. It is a good practice to plan and allocate extra space it the Pages and Extents of the DataStructure that contains the data rows, to allow for interspersed `INSERTs`; such planned space is not considered unused. Unused Space is specifically the space consumed that is unplanned or unconscious.

7 Determination

This chapter explains how Fragmentation at each Level and Type (explained in the previous chapter), for each type of DataStructure can be **determined accurately**, and evaluated. The next three sections provide information specific to each of the three Levels of Fragmentation; the fourth section identifies issues relating to Partitions.

7.1 Determination I

There are no *Sybase* facilities for identifying Level I Fragmentation, it requires proprietary code, such as our **HelpSpace** or **PhysicalSpace** utility, the report of which is shown here.

This section is for Customers only

7.2 Determination II Space

First, we will examine the **basic space metrics** related to Level II Fragmentation of the Logical DataStructures, summarising the underlying Physical DataStructures (Partitions) to the logical level. For non-patitioned DataStructures, this is all that is required. A simple query from `sysindexes`, which identifies each Logical DataStructure, is required [1,2].

Table					DataStructure			1			2						
Table	**Lck**	**Row**	**Fwd**	**Del**	**Struct**	**IndexName**	**Idx_KB**	**Unused**	**Used_%**	**Data_KB**	**Unused**	**Used_%**	**LGIO**	**SPUT**	**DPCR**	**IPCR**	**DRCR**
TestBase_APL	APL	2,000,010			Clst	UC_SecurityId	508	96	81.1	89,020	124	99.86	99.96	93.74	99.99		
					NC1	U__Name	75,720	38	99.95				98.92			99.64	81.85
TestBase_APL_Heap	APL	80,000			Heap					3,660	100	97.27	99.62	93.63	99.87		
					NC2	U__Name	3,056	28	99.08				99.08			99.69	81.68
TestBase_APL_Loc	APL	2,000,000			Clst	C__SecurityId	512	100	80.47	88,968	78	99.91	99.99	93.75	100.00		
					NC1	U__SecurityId	22,048	22	99.9				99.69			99.90	100.00
TestBase_DPL	DPL	2,105,177	0	309	Heap					105,768	3,056	97.11	100.00	94.19			
					NC1	U__SecurityId	51,672	230	99.55				26.02			5.25	90.63
					NC2	UP_Name	133,868	40	99.97				30.74			24.91	92.45
TestBase_DRL	DRL	100,000	0	0	Heap					4,896	16	99.67	100.00	94.17			
					NC1	UP_SecurityId	1,326	16	98.79				100.00			100.00	100.00
					NC2	U__Name	3,984	30	99.25				99.65			99.88	0.05

	Statistic	Requested For (DataStructure)	Returns
1	Unused Space/Index	Clustered Index (B-Tree) [3] Nonclustered Index	Unused pages in the B-Tree portion of the CI Unused pages in the NCI
2	Unused Space/Data	Heap Clustered Index (Data) [3]	Unused pages in the Heap Unused pages in the Data portion of the CI

- The `RESERVED_PAGES()` function returns the number of Pages reserved for the DataStructure. If the `partionid` is not supplied, all Partitions in the DataStructure are summarised. Multiplying this value by `@@PAGESIZE` returns bytes, which can then be divided into kilobytes or megabytes.
- Space for each DataStructure is allocated on an Extent basis (eight Pages); the Extent cannot be used by other DataStructures. Thus it is reserved.
- The value returned is of course, whole Pages.
- The `DATA_PAGES()` function returns the number of Pages in the DataStructure that contain data. If the `partionid` is not supplied, all Partitions in the DataStructure are summarised.
- Subtracting `DATA_PAGES()` from `RESERVED_PAGES()` yields unused Pages.
- Dividing them yields the percentage used.

1. For DOL tables, on the physical plane, a Heap DataStructure always exists. Additionally, a separate **Placement Index** (falsely named "clustered") DataStructure may exist. Such DataStructures are quite different to the single Clustered Index dataStructure. This is reflected in the catalogue, and is easily confirmed in any report, such as the example.
2. The information in the example reports, and much more, is provided in our **HelpIndex/HelpPartition** utilities.
3. The Clustered Index DataStructure has both B-Tree and Data components: the Pages reserved and the Pages used can be obtained for the B-Tree portion and the Data portion of the Clustered Index, separately.

7.3 Determination II DerivedStat

Second, we will examine the **Derived Statistics** provided by *Sybase* that relate to Level II Fragmentation of the Logical DataStructures, again summarising the underlying Physical DataStructures (Partitions) to the logical level. A simple query from `sysindexes`, which identifies each Logical DataStructure, is required [1 2].

Table					DataStructure												
Table	Lck	Row	Fwd	Del	Struct	IndexName	Idx_KB	Unused	Used_%	Data_KB	Unused	Used_%	3 LGIO	4 SPUT	5 DPCR	6 IPCR	7 DRCR
TestBase_APL	APL	2,000,010			Clst	UC_SecurityId	508	96	81.1	89,020	124	99.86	99.96	93.74	99.99		
					NC1	U__Name	75,720	38	99.95				98.92			99.64	81.85
TestBase_APL_Heap	APL	80,000			Heap					3,660	100	97.27	99.62	93.63	99.87		
					NC2	U__Name	3,056	28	99.08				99.08			99.69	81.68
TestBase_APL_Loc	APL	2,000,000			Clst	C__SecurityId	512	100	80.47	88,968	78	99.91	99.99	93.75	100.00		
					NC1	U__SecurityId	22,048	22	99.9				99.69			99.90	100.00
TestBase_DPL	DPL	2,105,177	0	309	Heap					105,768	3,056	97.11	100.00	94.19			
					NC1	U__SecurityId	51,672	230	99.55				26.02			5.25	90.63
					NC2	UP_Name	133,868	40	99.97				30.74			24.91	92.45
TestBase_DRL	DRL	100,000	0	0	Heap					4,896	16	99.67	100.00	94.17			
					NC1	UP_SecurityId	1,326	16	98.79				100.00			100.00	100.00
					NC2	U__Name	3,984	30	99.25				99.65			99.88	0.05

Statistic	Requested For (DataStructure)	Returns	
			Meaningless & Confusing [4]
3 `LGIO` Large I/O Efficiency	Heap	Page/Extent/AllocationUnit contiguity of Heap	
	Clustered Index	Page/Extent/AllocationUnit contiguity of CI	
	Nonclustered Index	Page/Extent/AllocationUnit contiguity of NCI	
4 `SPUT` Data Space Utilisation	Heap	Density of data rows per data page	
	Clustered Index	Density of data rows per data page	
	Nonclustered Index [5]		*Does not apply*
5 `DPCR` Data Page Cluster Ratio	Heap/APL	Density of data per page in the Heap, via PageChain	
	Heap/DOL [6]		*Does not apply*
	Clustered Index	Density of data per page in CI order	
	Nonclustered Index [7]		*Does not apply*
6 `IPCR` Index Page Cluster Ratio	Heap [8]		*Does not apply*
	Clustered Index [9]		*Does not apply*
	Nonclustered Index	Density of index pages in NCI order	
7 `DRCR` Data Row Cluster Ratio	Heap [10]		*Does not apply*
	Clustered Index [11]		*Does not apply*
	Nonclustered Index	Density of data rows in NCI order	

- The `DERIVED_STAT()` function returns five statistics for four of the five types of DataStructure [12]. Again, if the `partitionid` is not supplied, all Partitions in the DataStructure are summarised.
- Unfortunately, `DERIVED_STAT()` does not operate the way `RESERVED_PAGES()` and `DATA_PAGES()` operate: The Clustered Index is treated as a whole unit, values for the B-Tree and the data cannot be obtained separately.
- Further, instead of returning Null for requests that are not applicable, it returns interesting values or fixed values (0% or 100%), which lead to confusion [4]. The cells for meaningless figures are empty in the example report.
- Note that the function (all five statistics) return fairly exact information, at the row or intra-page level, whereas `RESERVED_PAGES()` and `DATA_PAGES()` returns whole Pages.

---

4. Display of *meaningless* figures causes great confusion, and invites comparison with *meaningful* figures, eg. `DPCR` for a DOL Heap (fixed 100%, meaningless) cannot be related to or be compared with `DPCR` for an APL Heap (meaningful) which can be addressed, in order to achieve close to 100%. Administrative time is wasted in correlating such figures and trying to make sense of them; decisions that may be made on the basis of such confusion are consequently irrelevant and meaningless. It is therefore better to avoid displaying meaningless figures, and to focus on the meaningful figures alone.
5. **Data Space Utilisation** Data is contained in either the Heap or the Clustered Index only, therefore `SPUT` applies to them alone, the fiigure for the NCI (always 0%) is meaningless.
6. **Data Page Cluster Ratio** The DOL Heap does not have a PageChain; data *page* access is via the OAM only; the figure (always 100%) is meaningless (space may well be poorly utilised); use `LGIO` or `SPUT` instead. It is not comparable with the `DPCR` of the APL Heap or CI.
7. `DPCR` is relevant for fetching data *pages*, which reside in the Heap or the Clustered Index only. It does not apply to the Nonclustered Index, since it is used to access data *rows*; data *pages* are never fetched via that structure. The Nonclustered Index (including the Placement Index) does not support **Range Queries**, only the Clustered Index does, and there it does fetch *pages*.
8. **Index Page Cluster Ratio** is relevant for fetching index *pages*; it applies to the Nonclustered Index. There are no *index* pages in the Heap; the figure (always 0%) is meaningless; refer to `IPCR` of the relevant NCI.
9. Index pages in the Clustered Index are not provided separately; the figure (always 0%) is meaningless; use `DPCR` instead.
10. **Data Row Cluster Ratio** is relevant for fetching data *rows*; it applies to the Nonclustered Index, since it is used to fetch data *rows*. It does not apply to the Heap since access to it is for *pages*, via the PageChain (APL) or the OAM (DOL). The figure (always 100%) is meaningless: for APL, use `DPCR` instead; otherwise, refer to `DRCR` of the relevant Nonclustered Index.
11. `DRCR` does not apply to the Clustered Index. Since the data rows in the Clustered Index are maintained in index order, by definition the `DRCR` is 100%. The figure is meaningless: for APL, use `DPCR` instead; for DOL, there is no Clustered Index, refer to `DRCR` of the relevant Nonclustered Index.
12. The function does not provide statistics for the Text/Image chain.

7.4 Determination III

SOFTWARE GEMS

Third, we will examine the **Forwarded** and **Deleted** row counts that relate to Level III Fragmentation of the Logical DataStructures, which occur in DPL/DRL lockschemes only. This applies to the Heap, and is in addition to, not instead of, `LGIO` and `SPUT` (which are explained in [7.3]). Again summarising the underlying Physical DataStructures (Partitions) to the logical level. A simple query from `sysindexes`, which identifies each Logical DataStructure, and `systabstats.forwrowcnt` & `delrowcnt` is required [1, 2].

Table			8	9	DataStructure								3	4			
Table	**Lck**	**Row**	**Fwd**	**Del**	**Struct**	**IndexName**	**Idx_KB**	**Unused**	**Used_%**	**Data_KB**	**Unused**	**Used_%**	**LGIO**	**SPUT**	**DPCR**	**IPCR**	**DRCR**
TestBase_APL	APL	2,000,010			Clst	UC_SecurityId	508	96	81.1	89,020	124	99.86	99.96	93.74	99.99		
					NC1	U__Name	75,720	38	99.95				98.92			99.64	81.85
TestBase_APL_Heap	APL	80,000			Heap					3,660	100	97.27	99.62	93.63	99.87		
					NC2	U__Name	3,056	28	99.08				99.08			99.69	81.68
TestBase_APL_Loc	APL	2,000,000			Clst	C__SecurityId	512	100	80.47	88,968	78	99.91	99.99	93.75	100.00		
					NC1	U__SecurityId	22,048	22	99.9				99.69			99.90	100.00
TestBase_DPL	DPL	2,105,177	0	309	Heap					105,768	3,056	97.11	100.00	94.19			
					NC1	U__SecurityId	51,672	230	99.55				26.02			5.25	90.63
					NC2	UP_Name	133,868	40	99.97				30.74			24.91	92.45
TestBase_DRL	DRL	100,000	0	0	Heap					4,896	16	99.67	100.00	94.17			
					NC1	UP_SecurityId	1,326	16	98.79				100.00			100.00	100.00
					NC2	U__Name	3,984	30	99.25				99.65			99.88	0.05

	Statistic	Requested For (DataStructure)	Returns
8	Forward [13]	DOL Heap	Variable length rows that have been transferred to another location
9	Delete [14]	DOL Heap	Rows that are marked for deletion

- `systabstats` contains one row for each Physical DataStructure, which means the columns must be summed to produce a Logical level report.
- Execute `sp_flushstats` before querying the table.
- **Forwards** and **Deletes** apply to the DOL Heap only.
- DOL tables always have a Heap, wherein the row resides. The Heap is **Static RowId based**. The space allocated for Forwarded rows (which consume the space of two rows) and Deleted rows (which consumes the space of one row), cannot be re-used for interspersed `INSERTS`.
- Since Forwards and Deletes do not apply to APL tables (row expansion is performed in-place and deletion is immediate), the relevant cells are empty in the example report.
- Space can be reclaimed via `REORG` or `DROP/CREATE "CLUSTERED" INDEX` (there is no Clustered index for DOL tables, but the syntax is required).

7.5 Evaluation

a. The three sets of metrics (Unused Space; Derived Statistics; Forwards & Deletes) regarding Fragmentation of a DataStructure must be taken together; any single metric should not be evaluated alone.

b. Similarly, all the DataStructures that belong to a table should be evaluated together. This should be done in the context of the actual usage: certain queries require single-row data (via an index); covered queries require access across an entire index; yet others would require table scans. Knowledge of how the data is accessed, and the DataStructures that are used to support that access, is essential to relevant administration.

c. In addition, the actual speed of the DataStructures belonging to the relevant tables must be monitored: timing records (for either a controlled test or an actual production sample at certain times of day, ensuring the same configuration and cache settings) must be kept, so that they can be compared before and after de-fragmentation operations.

- The value of any particular de-fragmentation operation must be confirmed: there is no point in performing operations that do no provide a benefit.
- The length of time between de-fragmentation operations, when speed is regained, and the point where the DataStructure has deteriorated enough to warrant the operation being repeated, should be recorded. If Level I Fragmentation is addressed, the frequency of such operations is substantially reduced.

d. Likewise, `sysmon` reports covering the period of the day should be maintained, or MDA data should be captured at relevant intervals. This is very important because it will allow you to tune the structures at an overall level (rather than on a DataStructure basis).

- The most important indicator of Fragmentation is that the **Asynchronous Pre-Fetch** capability that is built into the server, and the **Large I/O** resources that have been configured, are not used. Denying these facilities cripples the speed of *Sybase*.

13. For each Forwarded row, two row 'slots' are consumed: the first for the original location, the address of which is fixed, and cannot be moved; and the second for the forwarded location, which contains the expanded data row.
14. Deletes are not physically removed from DOL Heaps until `REORG` is executed.

7.6 Determination Partition

The above reports view the Logical DataStructures, and that is quite adequate for initial inspection, before further inspection is warranted. It is the end point for non-partitioned DataStructures. For Partitioned DataStructures [15], the Physical DataStructure must be inspected. The determination of Level II & III Fragmentation is only slightly more complex, it requires a simple query from `syspartitions`, which identifies Physical DataStructures, and `systabstats.forwrowcnt` & `delrowcnt` [1 2].

Table		DataStructure		Partition														
Table	Lck	Struct	IndexName	Partition	Row	Fwd (8)	Del (9)	Idx_KB (1)	Unused	Used_%	Data_KB	Unused (2)	Used_%	LGIO (3)	SPUT (4)	DPCR (5)	IPCR (6)	DRCR (7)
TestBase_APL	APL	Clst	UC_SecurityId	[1]	496,821			128	24	81.25	22,126	42	99.81	99.94	93.74	99.98		
				[2]	496,195			126	24	80.95	22,080	26	99.88	100.00	93.75	100.00		
				[3]	496,091			128	26	79.69	22,080	30	99.86	100.00	93.74	100.00		
				[4]	510,903			126	22	82.54	22,734	26	99.89	100.00	93.75	100.00		
		NC 1	U__Name					75,720	38	99.95				98.92			99.64	81.85
TestBase_APL_Heap	APL	Heap		[1]	20,891						960	30	96.88	100.00	93.60	100.00		
				[2]	20,245						928	28	96.98	100.00	93.73	100.00		
				[3]	20,007						910	20	97.80	100.00	93.67	100.00		
				[4]	18,857						862	22	97.45	100.00	93.54	100.00		
		NC 2	U__Name					3,056	28	99.08				99.08			99.69	81.68
TestBase_APL_Loc	APL	Clst	C__SecurityId	data_1	494,325			128	26	79.69	21,998	28	99.87	100.00	93.75	100.00		
				data_2	493,200			128	26	79.69	21,934	14	99.94	100.00	93.75	100.00		
				data_3	493,920			128	26	79.69	21,966	14	99.94	100.00	93.75	100.00		
				data_4	518,555			128	22	82.81	23,070	22	99.90	100.00	93.75	100.00		
		NC 1	U__SecurityId					22,048	22	99.90				99.69			99.90	100.00
TestBase_DPL	DPL	Heap		[1]	571,980	0	94				28,748	840	97.08	100.00	94.18			
				[2]	494,252	0	49				24,998	884	96.46	100.00	94.19			
				[3]	508,744	0	90				25,540	718	97.19	100.00	94.19			
				[4]	530,201	0	76				26,482	614	97.68	100.00	94.19			
		NC 1	U__SecurityId					51,672	230	99.55				26.02			5.25	90.63
		NC 2	UP_Name					133,868	40	99.97				30.74			24.91	92.45

- The columns have been re-arranged to clarify the DataStructure hierarchy and to make sense. The various row counts, space usage, and derived statistics are shown at the Partition (Physical) level, where it is actually located.
- The Heap and the Text/Image Chain are not named. Where the Partition is not explicitly named, an ordinal number is used to identify it (rather than the default Partition name, which is made up from the long and unusable `partitionid`).
- This example report lists Partitioned tables. It shows *all* DataStructures relating to each Partitioned table, in one place, in order to avoid having to examine two reports.
- `TestBase_DRL` is not Partitioned, thus it is absent from this report.

	Statistic	Requested For (Partition)	Returns	
				Meaningless & Confusing [4]
1	Unused Space/Index	Clustered Index (B-Tree)	Unused pages in the B-Tree portion of the CI	
2	Unused Space/Data	Heap Clustered Index (Data)	Unused pages in the Heap Unused pages in the Data portion of the CI	
3	`LGIO` Large I/O Efficiency	Heap Clustered Index	Page/Extent/AllocationUnit contiguity of Heap Page/Extent/AllocationUnit contiguity of CI	
4	`SPUT` Data Space Utilisation	Heap Clustered Index	Density of data rows per data page Density of data rows per data page	
5	`DPCR` Data Page Cluster Ratio	Heap/APL Heap/DOL [6] Clustered Index	Density of data per page in the Heap, via PageChain Density of data per page in CI order	 *Does not apply*
6	`IPCR` Index Page Cluster Ratio	Heap [8] Clustered Index [9]		*Does not apply* *Does not apply*
7	`DRCR` Data Row Cluster Ratio	Heap [10] Clustered Index [11]		*Does not apply* *Does not apply*
8	Forward [13]	DOL Heap	Variable length rows that have been transferred to another location	
9	Delete [14]	DOL Heap	Rows that are marked for deletion	

- `syspartitions` and `systabstats` each contains one row for each Physical DataStructure (Partition).
- Execute `sp_flushstats` before querying the tables.
- Only the DataStructure *that holds data rows*, either the Heap or the Clustered Index, is Partitioned; the Nonclustered Index and the Text/Image Chain are not Partitioned.

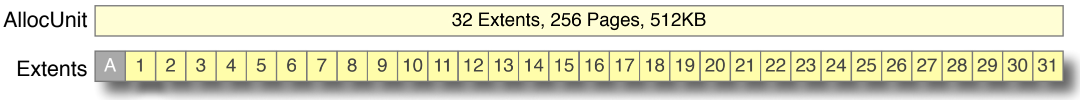
15. Partitioning (if implemented correctly at all resource levels) provides massively increased performance, improved concurrency (if OLTP Standards are implemented), and substantially reduces maintenance and de-fragmentation windows, because Partitions can be administered individually, or a needs basis.

8 I Allocation Unit

This part of the document identifies Level I Fragmentation: AllocationUnits within the Database (Allocations) and Extents within AllocationUnits. It is provided in three sections:

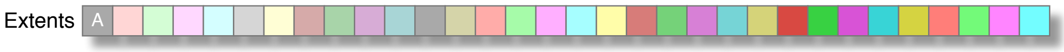
- AllocationUnit basics
- Why Drop/Create does not return Asynch Pre-Fetch and Large I/O, and
- Prevention of Level I fragmentation, the use of Segments.

8.1 Fresh



This shows the result of loading a single DataStructure into an *empty* **AllocationUnit**, and creating the Clustered Index, with `SORTED_DATA` if the CI was just droppped. The **Extents** are contiguous within the AllocationUnit; **Asynch Pre-Fetch** and **Large I/O** are fully operational. Even if the order was not sequential, and the PageChain was not linear, these facilities remain fully operational; the Look-Ahead set is not scaled down.

8.2 Fragmented

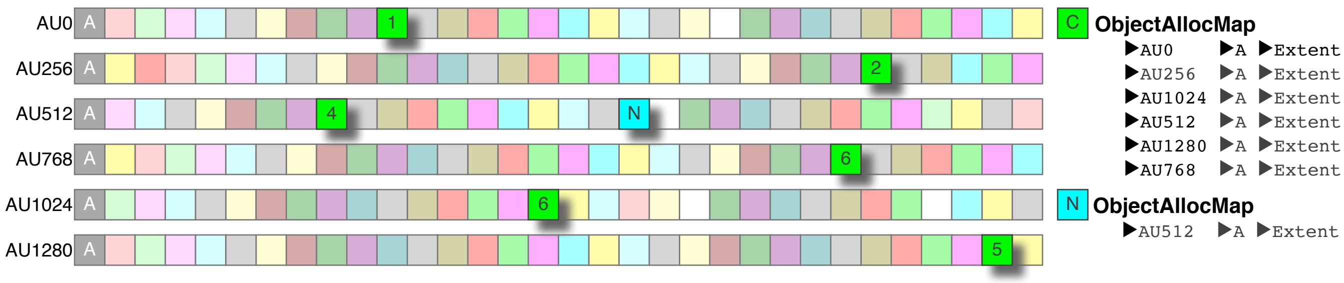


Where Segments are *not* understood and used, as in most sites, the reality is somewhat different. Since the Extents of up to 32 DataStructures (*physical* objects) can be located in an AllocationUnit, and all tables were loaded by *concurrent* `INSERTS`, the AllocationUnits each end up with Extents belonging to 32 different DataStructures. The Extents are fragmented within the AllocationUnits, and the AllocationUnits are fragmented across all Devices.

- Where 128 DataStructures are loaded, they are all fragmented across four AllocationUnits, etc.
- The `INSERTS` to all tables contend for the few currently active AllocationUnits, creating an **AllocationUnit Hotspot**.
- Further, the `INSERTS` to each table contend with its own Nonclustered indices: if a nominal table has 3 Nonclustered indices, that would be 32 tables with their NCIs, resulting in 128 DataStructures, across four AllocationUnits.

ASE correctly identifies that Asynch Pre-Fetch & Large I/O (multiple Extents, up to an entire AllocationUnit, at Level I) is not worth attempting. In such circumstances, drop/create Clustered Index, while de-fragmenting the DataStructure *within itself* (Levels II & III), does nothing to improve the established fragmentation at the AllocationUnit level (I): once it is set, it is set for life (refer next page), until Segments are used along with fresh Allocation Units.

8.3 DataStructure Perspective



This shows the Extents of a typical table, comprising two DataStructures:

- a single Clustered Index (containing data and index Pages) or a DOL Heap (data Pages only) in green,
- and one Nonclustered Index (index Pages only) in blue. (The DOL Placement Index is an ordinary Nonclustered Index, a separate DataStructure, some distance away from its data Heap, although on the same Segment.)
- The Pages within each DataStructure are some distance apart from each other. Here they cross Allocation Units.
- For DataStructures that have a PageChain, it is disturbed (the numbers show the sequence); it traverses AllocationUnits.

The first page of each AllocationUnit is the AllocationPage. The first page of a DataStructure contains its **ObjectAllocationMap**, that perspective is on the right. It is a list of all AllocationUnits that contain the DataStructure. The AllocationUnit is then interrogated via its AllocationPage to find the Extents that belong to the DataStructure.

An Object (*physical* term, as in ObjectAllocationMap; and which is unfortunately different to `OBJECT_ID()`, etc., which is a *logical* term) is a **DataStructure**, one of:

- Clustered Index (APL Only)
- Heap (DOL: always, APL: only when there is no CI)
  - the DOL Heap and APL Heap are very different
- Nonclustered Index (DOL Placement Index is NCI)
- Text/Image Chain

- *The web is full of mis-information, and shallow information.*
  - *Single-vendor sites are censored, and exclude robust discussion of technical issues related to their offerings; they have their commercial agenda.*
- *There is no substitute for actual experience, or for diligently verifying that you have actually accomplished what you set out to.*
- *Fragmentation at every level shown here, is easy to identify.*
  - *The success, and ease of correction, depends on your skills and understanding of this information: this is published free to assist you in that regard.*

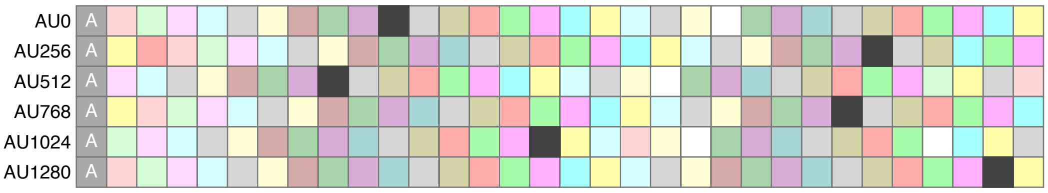
Intro   Unit   DataStruct   Defn   II Determ   III Determ   I Segment   II PageChain   II Unused   III Page

9 I Drop-Create

9.1 Common De-Fragmentation Issue

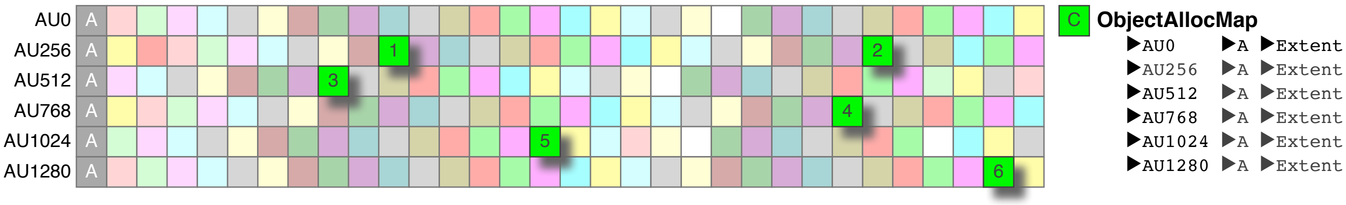
This chapter discusses some of the issues relevant to typical de-fragmentation exercises, and the limitations of `DROP/CREATE CLUSTERED INDEX`. Many DBAs de-fragment their DataStructures by performing the full complement of the three steps identified here, and puzzled: while the table is significantly faster, Asynch Pre-Fetch and Large I/O are not reurned. The DataStructure concerned is either a Clustered Index or a DOL Heap, the before image is illustrated in [8.3].

9.2 BCP-Out, Drop



The data has been `bcped-out`, and the table has been truncated, or the table is dropped and recreated. As long as Segments are not used to place the table on different Devices, or separate groups of tables, this sequence applies.

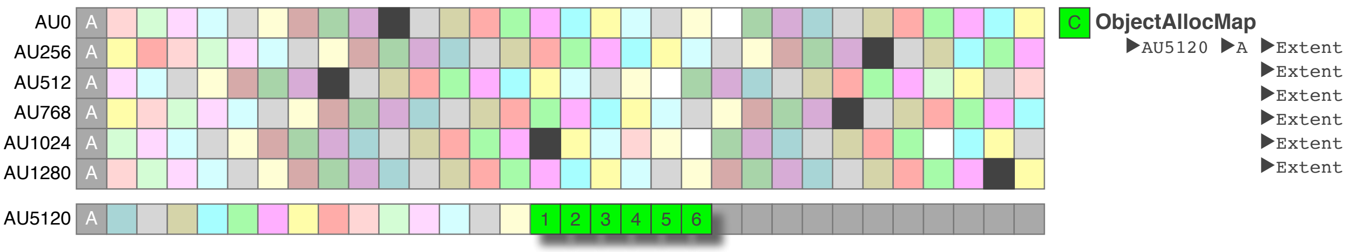
9.3 BCP-In, Create Clustered Index Sorted Data



When the data is `bcped-in`, it is placed in the available Extents, most likely the recently evacuated ones (assuming unload/load is performed when the database in not in use). Certainly, the DataStructure is de-fragmented within its own Extents and Pages (Levels II & III). However, if proceeding with one or a few DataStructures at at time; the Extents de-allocated will be re-used; they were fragmented at Level I before; and they remain so. Asynch Pre-Fetch & Large I/O (multiple Extents, up to an entire AllocationUnit, at Level I) is still not possible. Although advised by many Sybase identities, this is a common mistake; at any rate, its effect is temporary, and it needs to be repeated.

If `SORTED_DATA` is used, which does not re-write the data Pages, the Extents remain in their location.

9.4 Drop, Create Clustered Index



The distilled requirement, is simply to create the Clustered Index *without* the `SORTED_DATA` option; this re-writes the data Pages to a new location. Which makes the `bcp-out/bcp-in` unnecessary. However, the original DataStructure space, which is released at the end of the process, will be used for whichever Clustered Index is created *next*, as shown in section [9.6].

`bcp-out/bcp-in` is effective only when the entire database, or at least a large groups of tables, are de-fragmented together. Otherwise, aa new location can be specified by creating a new Device and identifying a new Segment on it.

Drop-Create and Sorted_Data

- `DROP/CREATE CLUSTERED INDEX` in its unqualified form rewrites all data Pages, and the PageChain (if the structure has one); the operation requires 125% of the space used, at the new location.
- When it is qualified `WITH SORTED_DATA`, the data Pages are not re-written, which means fragmentation is not corrected. It is extremely fast, especially in 15.0
- To correct fragmentation without losing the speed, use `WITH SORTED_DATA`, along with `FILLFACTOR` and/or `RESERVEPAGEGAP`, which forces the data Pages to be rewritten.

DPL/DRL Lockscheme

- The `WITH SORTED_DATA` qualifier cannot be used with DOL tables, because the order of the data in the Heap is not maintained (there is no Clustered Index).
  - The exception is that *fleeting moment* immediately following a full `REORG` or `DROP/CREATE "CLUSTERED" INDEX` (using the syntax demanded to address the Placement Index), in which case the rebuild is not required.

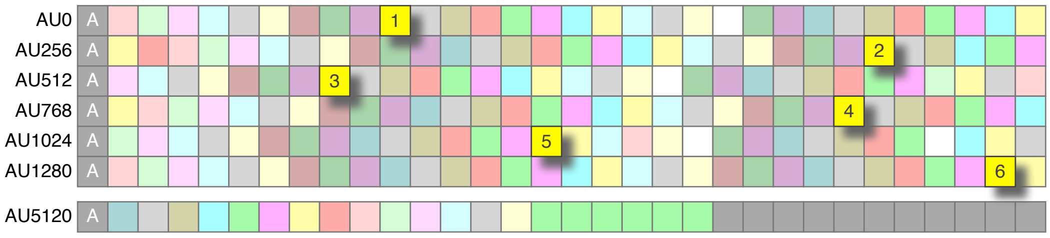
9.5 Drop, Create Placement Index

P ObjectAllocMap
▶AU5120 ▶A ▶Extent
▶Extent

H ObjectAllocMap
▶AU5120 ▶A ▶Extent
▶Extent
▶Extent
...

For DOL tables containing more than a few Extents, even immediately following a careful de-fragmentation exercise (`DROP/CREATE "CLUSTERED" INDEX` in *fresh* AllocationUnits), although the Heap is initially contiguous, since the Heap and Placement Index are two separate DataStructures, except for the first few Pages, the index and data Pages are substantially removed from each other.

9.6 Create Clustered Index/Next Clustered Index



C ObjectAllocMap
▶AU0 ▶A ▶Extent
▶AU256 ▶A ▶Extent
▶AU512 ▶A ▶Extent
▶AU768 ▶A ▶Extent
▶AU1024 ▶A ▶Extent
▶AU1280 ▶A ▶Extent

The *next* Clustered Index created takes up the fragmented Extents which were vacated by the *previous* Clustered Index (green) when it was re-written to a new location.

There really is no substitute for Segments.

DPL/DRL Lockscheme

- For DOL tables, once the Pages and Extents in the Heap are reasonably full, unless space is reserved for interspersed `INSERTs` and row expansion, it is not possible for rows to be placed "near" each other (as intended by the Placement Index); logically sequential rows or Pages could be hundreds of megabytes apart.
- Further, the index Pages in Placement Index and the related data Pages in the Heap could be hundreds of megabytes apart (while remaining "on the same Segment", `default` or otherwise).

10 I Segment

10.1 Normal Growth

Refer to section [**2.1**] for introduction to Segments; this chapter discusses the value of Segments in reducing or eliminating Fragmentation.

The use of **Segments** allows groups of tables to be stored together, and thus separated from competing table groups, on discrete **Devices**. This shows the AllocationUnits of:

- 6 Segments `Data1` through `Data6` (table groups, base colours) used for the Clustered Indices of 18 tables (distinct shades)
  - for the purpose of explanation, the Devices may well be named `Data1` through `Data6` as well
- 2 Segments `NC1` and `NC2`, for all their Nonclustered Indices (an arbitrary 3 Nonclustered Indices `A`, `B`, `C`, per table is shown).

																																		ObjectAllocMap (CI)
Data1	AU0	A	1	1	1	2	3	2	2	4	5	3	3	6	7	4	4	8	9	5	5	10	11	6	6	12	13	7	7	14	15	8	8	O O O ▶AU0
Data2	AU256	A	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6	6	6	7	7	7	8	8	8	9	9	9	10	10	10	11	O O O ▶AU256
Data3	AU512	A	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6	6	6	7	7	7	8	8	8	9	9	9	10	10	11	11	O O O ▶AU512
Data4	AU768	A	1	1	1	2	3	2	2	4	5	3	3	6	7	4	4	8	9	5	5	10	11	6	6	12	13	7	7	14	15	8	8	O O O ▶AU768
Data5	AU1024	A	1	1	1	2	3	2	2	4	5	3	3	6	7	4	4	8	9	5	5	10	11	6	6	12	13	7	7	14	15	8	8	O O O ▶AU1024
Data6	AU1280	A	1	1	1	2	3	2	2	4	5	3	3	6	7	4	4	8	9	5	5	10	11	6	6	12	13	7	7	14	15	8	8	O O O ▶AU1280
NC1	AU1792	A	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C					
NC2	AU2048	A	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C					

Where **Segments** are *not* used, all data is placed in the `default` Segment. Since all Objects are loaded via *concurrent* `INSERTS`, the Extents are fragmented within the AllocationUnits, and the AllocationUnits are fragmented across all Devices. That case, unfortunately quite common, is illustrated in sections [**8**] and [**9**]. The illustration abovee shows exactly the same quantity of DataStructures and Extents that are shown in those sections, the numbers continue to identify Extent number within the DataStructure. The above illustrates the result of all tables being evenly, and concurrently, `INSERTED` into.

The use of Segments provide three major advantages:

1. Reduction of fragmentation, due to more Extents belonging to fewer DataStructures being placed on each Allocation Unit
   - thus Level I de-fragmentation operations are reduced, if not eliminated.
   - Asynch Pre-Fetch & Large I/O (multiple Extents, up to an entire AllocationUnit, at Level I) is now reasonably possible, it is worthy of consideration to the Optimiser.
2. Substantially increased performance, due to:
   - enhanced *concurrent* `INSERT` speed, for several reasons, primarily because the:
     - the tables required in each transaction are separated from each other, on separate Segments, and
     - Nonclustered Indices are separated from their data (Clustered Index or DOL Heap), on separate Segments
     - onto many Device queues.
3. The absence of Segments results in a few current Allocation Unit Hotspots, on **one** (the current) Device, despite many Devices being available. Such hotspots are eliminated.

10.2 Fragmented

																																		ObjectAllocMap (CI)
Data1	AU0	A	1	1	1	3	4	2	3	6	7	3	2	9	10	4	8	12	13	5	4	14	2	6	6	5	8	7	5	11	15	8	7	O O O ▶AU0
Data2	AU256	A	1	1	1	2	2	2	4	3	4	3	4	5	7	5	7	5	6	8	6	7	10	11	8	3	8	9	6	9	10	9	10	O O O ▶AU256
Data3	AU512	A	1	1	1	3	2	2	4	3	4	6	4	5	7	5	7	10	6	8	11	7	11	2	8	3	5	9	6	8	10	9	9	O O O ▶AU512
Data4	AU768	A	1	1	1	3	4	3	2	5	6	4	3	8	10	6	4	12	13	7	5	14	2	2	6	7	9	5	7	11	15	8	8	O O O ▶AU768
Data5	AU1024	A	1	1	1	3	4	3	2	6	7	5	3	8	10	7	4	11	12	2	5	2	5	4	6	9	13	6	7	14	15	8	8	O O O ▶AU1024
Data6	AU1280	A	1	1	1	2	4	2	3	5	7	3	4	8	10	4	7	11	12	5	8	14	15	6	2	3	6	7	5	9	13	8	6	O O O ▶AU1280
NC1	AU1792	A	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C					
NC2	AU2048	A	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C					

This shows the same group, eventually fragmented at Level I under interspersed `INSERT/DELETE` activity (`UPDATE` only causes Row migration or Page splits when the columns are variable), which would cause PageSplits, etc; the resulting fragmentation is depicted. Where even simple Segment plans are used, fragmentation can be substantially reduced; where carefully considered Segment Plans are used, Level I de-fragmentation operations can be avoided altogether. Even though fragmented, Asynch Pre-Fetch & Large I/O are fully enabled (although slightly less efficient than when not fragmented).

Note also that since the AllocationUnits are laid out *initially* as per [10.1], the structures are essentially immune to becoming fragmented. Therefore what is shown here is the result of *extreme* interspersed `INSERT/DELETE` activity, *and* over a long period.

The effect of de-fragmenting single tables (ie. at the DataStructure level, as and when required, via `DROP/CREATE CLUSTERED INDEX`, to correct Level II fragmentation as illustrated above, without requiring unload/reload, produces [10.1] for the subject DataStructure. Since each new DataStructure takes up the Extents of the previous DataStructure, and that latter was unfragmented for the most part; the *sequence of Extents* is corrected. However, that is not the completely contiguous, as shown next.

10 I Segment

10.3 Fresh

Data1	AU0	A	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data2	AU256	A	1	2	3	4	5	6	7	8	9	10	11	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10
Data3	AU512	A	1	2	3	4	5	6	7	8	9	10	11	1	2	3	4	5	6	7	8	9	11	1	2	3	4	5	6	7	8	9	10
Data4	AU768	A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Data5	AU1024	A	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data6	AU1280	A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
NC1	AU1792	A	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C				
NC2	AU2048	A	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C				

ObjectAllocMap (CI)

O	O	O	▶AU0
O	O	O	▶AU256
O	O	O	▶AU512
O	O	O	▶AU768
O	O	O	▶AU1024
O	O	O	▶AU1280

The effect of de-fragmenting most or all the tables in each **Segment** is illustrated here. Of course, Each Segment can be de-fragmented as and when necessary; all Segments do not need to be de-fragmented at the same time. Where Segments are not used, none of this is possible.

11 Level I Fragmentation Summary

To summarise the types of fragmentation covered in Level I:

- AllocationUnits are fragmented across the Database, preventing Asynch Pre-Fetch & Large I/O (multiple Extents up to an AllocationUnit at Level I).
- Extents are fragmented across the AllocationUnits, preventing Asynch Pre-Fetch & Large I/O (multiple Extents up to an AllocationUnit at Level I).
- Such fragmentation can be greatly reduced by implementing Segments, since it limits the physical range of DataStructures.

Further, Segments increase performance by separating DataStructures that compete or contend with each other.

Segment Limit

For large databases, the 29 Segment limit poses an obstacle, which must be worked around by loading tables in tranches. At the least, when Clustered Indices are rebuilt to address Level II Fragmentation, they can be rebuilt singly, and in place, and without the vulnerability illustrated in chapter [**9**].

DPL/DRL Lockscheme

- Placement Indices and Heaps, which are separate DataStructures (although on the same Segment), are not explicitly illustrated here; a single pair is illustrated in [**9.5**].
- Sites that use such tables generally do not use Segments, and thus all DataStructures in the entire database is fragmented across the single `default` Segment. Florists call this "striped", and wonder why it is slow; engineers call it retarded.

Surrogate Key

- A monotonically increasing value, such as an `IDENTITY` column, creates myriad problems, which do not occur with true Relational keys.
  - `IDENTITY` columns are fine for prototype systems (development). Due to the many attendant restrictions they impose on ordinary maintenance task, they must not be allowed in production systems.
- It *creates* an `INSERT` HotSpot on the last Page, and guarantees contention.
- The hotspot exists for both APL and DOL tables, with the latter being slightly faster.
- A monotonically increasing key is the worst candidate for a Clustered index: choose a Key that distributes the data, and therefore eliminates the hotspot.
- Contact the author for alternative, high performance methods.

Mythology

Based on the naïve belief that *The SAN Does Everything*, and in substitution of knowledge or technical examination:

- **Myth** that fragmentation does not matter when a SAN is used, because the volumes are striped (effectively "fragmented").
- **Myth** that Segments are not required where a SAN is used. Reasons always fall apart under questioning; and the proof is in the pudding.

  *The SAN, and whatever configuration is implemented, is independent of ASE, and vice versa. ASE 'sees' the Logical Volume on the SAN as a **contiguous disk allocation**, and treats it that way in attempting to obtain performance out of it (eg. Asynch Pre-Fetch & Large I/O)*

Based on the naïve belief that *Evangelists Preach the Gospel*, while ignoring the fact that Evangelism is a marketing concept, and in substitution of genuine knowledge and technical examination:

- **Myth** that the DOL Placement Index (unfortunately addressed via the "clustered" syntax), is the same as the Clustered Index.

  The unconfused technical term for the two separate DataStructures is **Heap and Placement Index**

12 II Page Chain

SOFTWARE GEMS

This part of the document identifies Level II Fragmentation: Pages within Extents, and shows the effect for the different LockSchemes. There are four aspects to this level, presented in seven sections:

- PageChain Fragmentation
- Overflow Pages
- Unused Space (Pages) per Extent, and
- Unused Space per Page.

AllPage Locked

12 Page Chain Fragmentation

PageChains exist for:

- Heap (which exists only when there is no Clustered Index)
- Clustered Index (all Index levels & Leaf levels, meaning *index and data* Pages, since the Leaf is the data row), as per [**3.2**].
- Nonclustered Index (Leaf level only)

12.1 Fresh

Clustered Index

```
Extent  [      8 Pages          ]      ObjectAllocMap
E512    [1][2][3][4][5][6][7][8]       [C] ▶AU512 ▶A ▶Extent
E520    [9][10][11][12][13][14][15][16]              ▶Extent
                                                     ▶Extent
```

This illustrates an unfragmented Clustered Index Leaf level PageChain, containing *index Leaf plus data*. It is contiguous, fresh after loading via `bcp` or `DROP/CREATE CLUSTERED INDEX`.

- Asynch Pre-Fetch & Large I/O (multiple Extents, up to an entire AllocUnit, at Level II, and multiple Pages) are fully enabled.

12.2 Fragmented

```
                                       ObjectAllocMap
E512    [1][2][4][5][6][8][9][10]      [C] ▶AU512 ▶A ▶Extent
E520    [11][13][14][16][17][19][20][3]              ▶Extent
E528    [22][7][23][12][24][15][18][21]              ▶Extent
```

- This shows a disturbed PageChain, caused by Page Splits, when full pages need to be split due to interspersed `INSERTS`, *and* no space being available on the Page.
- This shows Pages out of sequence while remaining in the same AllocationUnit; the I/O penalty is more severe when the out-of-sequence Pages are located in other AllocationUnits, as per [**8.3**].

12.3 Effect/Range Query & Table Scan

```
[1][2]•[3]•[4][5][6]•[7]•[8][9][10][11]•[12]
[13][14]•[15]•[16][17]•[18]•[19][20]•[21]•[22]•[23]•[24]
```

- This shows the sequence in which the Pages must be fetched when traversing the PageChain, eg. for **Range Queries** and **Table Scans**, and highlights the interrupts involved in the traversal
- Asynch Pre-Fetch & Large I/O (multiple Extents, up to an entire AllocUnit, at Level II) are prevented. Multiple Pages are hindered.
- When traversing the PageChain, 15 reads are required instead of 3.
  - On a busy server, that could be up to 14 **interrupts**, or context switches, which are to be avoided
  - PageChains that are fragmented across AllocationUnits require more of those to be read, and even more I/O
  - If the Pages are aged out of the cache during this time, they must be read again, etc. (Not illustrated.)

DataPage/DataRow Locked

- There is no PageChain for the DOL Heap (it would defeat the purpose of the RowId based design)
- There is a PageChain for the Leaf level of Nonclustered Indices (including the Placement Index), as per [**3.3**]

Heap & Placement Index

```
Extent  [      8 Pages          ]      ObjectAllocMap
E768    [1][2][3][4][5][6][7][8]       [H] ▶AU768 ▶A ▶Extent
E60     [9][10][11][12][13][14][15][16]              ▶Extent
                                                     ▶Extent
E1280   [1][2][3][4]                   [P] ▶AU1280 ▶A ▶Extent
```

This shows an unfragmented DOL Heap, the *data*; it is contiguous ***because*** it has been freshly re-ordered via `DROP/CREATE CLUSTERED INDEX`. It also shows the unfragmented Placement Index.

- Although the syntax demands "clustered", it is false; the index is in fact a Placement Index, which is a Nonclustered Index with two additional criteria (the data is not clustered with the index); the illustration shows what *initial placement* does.

```
                                       ObjectAllocMap
E768    [1][2][4][5][6][8][9][10]      [H] ▶AU768 ▶A ▶Extent
E776    [11][13][14][16][17][19][20][3]              ▶Extent
E784    [22][7][23][12][24][15][18][21]              ▶Extent
E1280   [1][2][3][4]                   [P] ▶AU1280 ▶A ▶Extent
```

- The Heap is fragmented due to DML activity, and no space being available in the Page, standard fare for monotonically increasing indices.
- The sequence is not real, since Pages are not accessed in sequence; it merely provides a camparison to that on the left (the real sequence is much worse)
- To some extent that does not matter, because there is no PageChain and Range Queries are not supported. However, the overall access to the table is slowed, and scans must use the OAM method.

- **Range Queries** are based on a Clustered Index (index Leaf plus data), Relational or compound Keys, and require a PageChain; since DOL tables cannot have a Clustered Index, the feature is not possible for them.
- Traversing the Heap, eg. **Table Scans**, requires navigation via the ObjectAllocationMap; to the Allocation Page; to the Extent; to the Page. That is much slower than retrieval via a PageChain (or comparable to a heavily fragmented PageChain)

AllPage Locked | DataPage/DataRow Locked

12.4 Effect/Covered Query

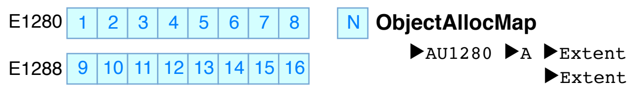
A **Covered Query** pertains to either Clustered Index or Nonclustered indices (including the Placement Index), where the query can be serviced by reading the Index Leaf level alone, and reading the data Pages is avoided. This is quite different to Range Queries, which applies to index plus data. It uses the PageChain available at the Leaf level of the Index.

- Refer to [**3.2**] for a definition of the CI, note the PageChain at every level of the B-Tree, and at the Leaf (data) level.
- Refer to [**3.3**] for a definition of NCI or PI and its relation to the data, note the PageChain at the Index Leaf level only.
- Refer to [**12.2**] and [**12.3**] for the effect of fragmentation on a Clustered Index on Table Scans and Range Queries.

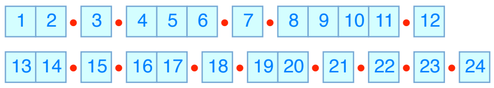
We will now contemplate the effect of fragmentation on Nonclustered Indices.

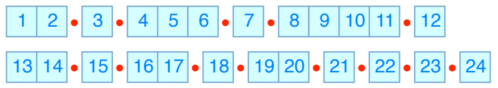
Nonclustered Index

Nonclustered or Placement Index



This illustrates an unfragmented Nonclustered Index Leaf level PageChain, containing index Leaf entries. It is contiguous, fresh after `DROP/CREATE NONCLUSTERED INDEX` (or "clustered" if it is a Placement Index)





This illustrates the effect of fragmentation on the PageChain of a Nonclustered index (including PI). It shows the sequence in which the Pages must be fetched when traversing the PageChain, and highlights the interrupts involved in the traversal

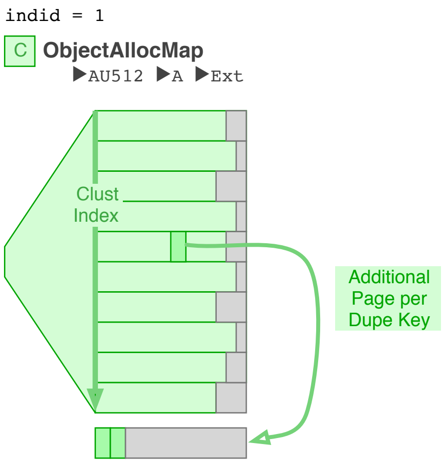
- Asynch Pre-Fetch & Large I/O (multiple Extents, up to an entire AllocUnit, at Level II) are prevented. Multiple Pages are hindered.
- When traversing the PageChain, 15 reads are required instead of 3.
  - On a busy server, that could be up to 14 **interrupts**, or context switches, which are to be avoided
  - PageChains that are fragmented across AllocationUnits require more of those to be read, and even more I/O
  - If the Pages are aged out of the cache, they must be read again, etc. (Not illustrated.)

> **Focus**
> In order to avoid confusion, and to maintain focus, other Levels of fragmentation are excluded from this Level II discussion. Page level issues such as the space usage consequences relating to DOL tables are discussed in **Level III Fragmentation**. Unused Space within Extents is discussed in [**14**], Unused Space within Pages is discussed in [**15**].

AllPage Locked

13 Overflow Page

Clustered Index/Duplicate Row



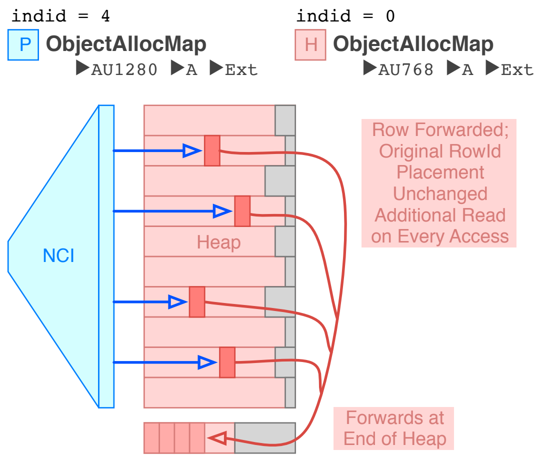
Overflow pages occur only for a Clustered Index that is non-unique. For each CI key that is duplicated, an Overflow Page is required, which contains a chain of duplicate rows, the single original row remaining in the contiguous CI DataStructure.

The **Clustered Index** DataStructure is not designed to allow duplicate keys.

- By definition, in a Relational Database, every row must be unique; APL tables are highly suited to that purpose; and thus it is not an issue in Relational tables
- Record filing systems with `IDENTITY` or surrogate keys should use DOL tables, and thus it is again not an issue.
- In any case, every CI should be unique; a non-unique CI should be viewed as a serious error, not merely as additional I/O.
- For 'queue' or 'pipe' or log tables, a Heap without a CI is best. Where a CI has been chosen (eliminating a Heap), ensure that the CI is unique.

DataPage/DataRow Locked

Heap & Placement Index/Forward



DOL DataStructures do not have Overflow Pages in the sense that *Sybase* has not given it a name. However the concept of Forwarded Rows is identical, and far more frequent (row expansion vs row duplication), although the overhead is greater. A technically accurate name, in the context of existing, established names, is Overflow Pages, albeit for Forwards rather than for Duplicates.

A further difference is that the Forwarded row consumes the space of two rows, since the original location cannot be used; whereas the APL duplicate consumes one row.

Since the **Nonclustered Index** (including Placement Index) and the Heap are physically separate DataStructures, and row order is not maintained, duplicate rows are not an issue: the management of duplicate keys can be handled within the index B-tree structure. For such indices, there is one Leaf entry (RowId) for each key, whether duplicated or not; the duplicate rows are merely two Index Leaf entries; two different RowIds.

14 II Unused Space Extent

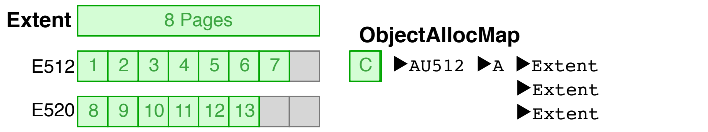
For all DataStructures, a few empty slots in each Page (via `FILLFACTOR`) and a few empty Pages in each Extent (via `RESERVEPAGEGAP`) is desirable, to allow for interspersed `INSERTS`. However, where there are more interspersed `DELETES` than interspersed `INSERTS`, this may be more than is desired. Where there are no interspersed `INSERTS`, unused space is not required.

The issue relevant to unused space is, whether it was planned or not; and only the latter is a problem. Let us consider unused space that is unplanned. Here the DataStructure that contains the data rows (Clustered Index for APL or Heap for DOL) is most relevant, and detailed below. Nonclustered Indices do get fragmented (in the category of unused space), when there are bulk `DELETES` that are interspersed. However, this is easy and fast to correct (drop and create the index). In any case, Nonclustered Indices are affected more by disturbed PageChains, than by unused Extents or Pages.

AllPage Locked

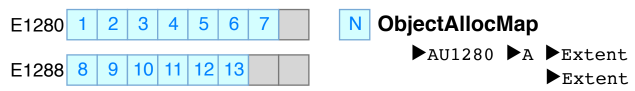
14 Unused Space/Extent

Clustered Index



Both CI and NCI are shown here, obviously the effect on data Pages, and the correction thereof, is much more serious. The NCI is easy and fast to correct.

Nonclustered Index

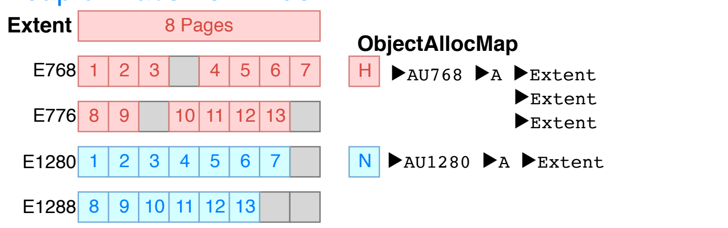


14.1 Effect

- Asynch Pre-Fetch & Large I/O (multiple Pages, up to an entire Extent, at Level II), where Extents are requested, is not hindered. The self-modulating Look-Ahead Set is simply scaled down a little, unless the ratio of empty Pages is large.
- This applies when traversing the Clustered Index, eg. for **Range Queries, Covered Queries** and **Table Scans**, and traversing the Nonclustered Index for **Covered Queries**.

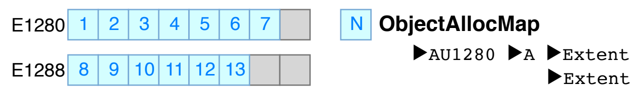
DataPage/DataRow Locked

Heap & Placement Index



Both the Heap and Placement Index are shown here, obviously the effect on data Pages, and the correction thereof, is much more serious. Correcting the Heap constitutes a demand to drop and create the Placement Index (unfortunately addressed via the "clustered" syntax), since the PI defines *initial* placement of rows in the Heap.

Nonclustered or Placement Index

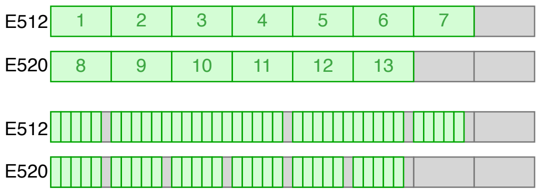


- Asynch Pre-Fetch & Large I/O (multiple Pages, up to an entire Extent, at Level II), where Extents are requested, is somewhat hindered. The self-modulating Look-Ahead Set is scaled down a little more than in APL.
- This applies when traversing the relevant Nonclustered Index, eg. for **Covered Queries**.
- **Range Queries** are not supported for DOL tables.
- **Table Scans** use the OAMPage access method.

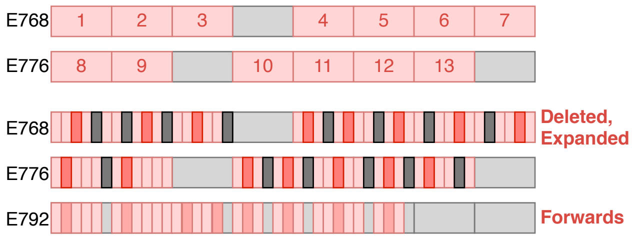
AllPage Locked | DataPage/DataRow Locked

15 Unused Space/Page

This illustrates the result of heavy interspersed `INSERT/DELETES` at the Page level for the Lock Schemes, the rows in the Pages in the same pair of Extents in [**14**] above are shown.



The Page is kept trim: rows are shifted upon deletion and row expansion/contraction.



Note that even at this level, the forwarded rows (red); forwards (dark pink); and deleted rows (dark grey) are visible, separate from unused space (light grey). The additional space requirement is obvious. (*In order to avoid confusion, Level III Fragmentation is excluded from this Level II discussion; it is discussed separately, overleaf*.)

15.1 Effect

AllPage Locked

- Asynch Pre-Fetch & Large I/O (multiple Pages, up to an entire Extent, at Level II), where Extents are requested, is not hindered, since the Pages are trimmed. The self-modulating Look-Ahead Set is simply scaled down a little, unless the ratio of Unused Space per page is large.
- This applies when traversing the Clustered Index, eg. for **Range Queries**, **Table Scans,** and traversing the Nonclustered Index for **Covered Queries**.

DataPage/DataRow Locked

- Asynch Pre-Fetch & Large I/O (multiple Pages, up to an entire Extent, at Level II), where Extents are requested, is somewhat hindered, since the Pages are not trimmed; `DELETED` rows are not deleted; and rows are Forwarded. The self-modulating Look-Ahead Set is scaled down a lot more than in APL.
- This applies when traversing the relevant Nonclustered Index, eg. for **Covered Queries**.

16 Level II Summary

To summarise the types of fragmentation covered in Level II:

- PageChains are fragmented across Extents, or worse, across AllocationUnits.
  - This prevents Asynch Pre-Fetch & Large I/O (multiple Extents and Pages at Level II).
  - Such fragmentation can be greatly reduced at the highest level by implementing Segments, since it limits the physical range of DataStructures.
  - It can be reduced at the DataStructure level by reserving space for expected interspersed `INSERTS` and row expansion. Disk space is cheap.
- Unplanned Unused Space within Extents and within Pages scale down Asynch Pre-Fetch & Large I/O.
  - Planned reserved space maintains the speed of the DataStructure. Yes sir, everything in a computer system is a trade-off.
- Level II fragmentation is corrected via `DROP/CREATE CLUSTERED INDEX` with the appropriate `FILLFACTOR`.

Elimination of Row Movement

- Variable length rows is the main causes of **Deferred Writes**, which are much slower than **Direct Writes**.
- Row movement within the page, and the consequential PageSplits (in APL), and Row Forwarding (in DOL) is caused by row size changes. This can be *eliminated* by implementing fixed rows. That means elimination of variable length and Nullable columns.

Reserved Space

- Space can always be reserved, for any DataStructure (Heap, CI, NCI) via:
  `RESERVEPAGEGAP`: reserve Page(s) per AllocationUnit or Extent
  `FILLFACTOR`: reserve space per Page
- use `sp_chgattr` in order to make the settings permanent

Reserved Space/DOL

- Fixed length rows are best, because it eliminates Row Forwarding entirely. However, if that cannot be achieved, the `EXP_ROW_SIZE` should always be set correctly.

17 III Page

Level III is a new form of fragmentation (Pages and Rows) that applies to **DOL tables only**. These pages illustrate the fragmentation in their DataStructures, as a consequence of normal DML activity, step by step, and compares them with APL. Understanding the different **DataStructures** and their relations, is a pre-requisite.

AllPage Locked

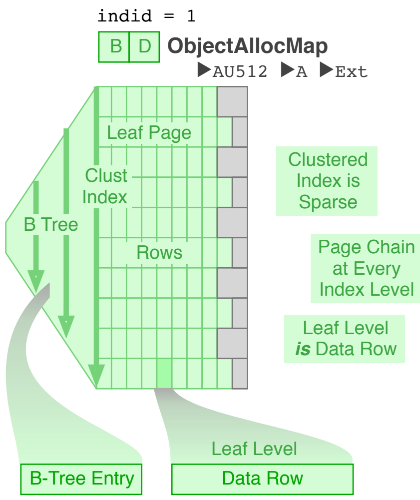
DataPage/DataRow Locked

Before launching into the detail, the key issue that must be understood, the essential difference between APL vs DOL is:

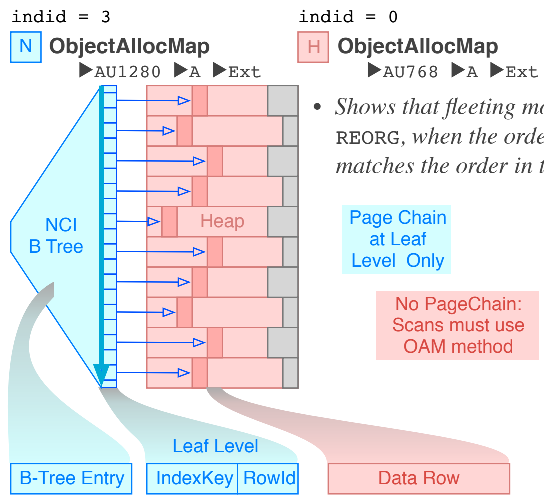
- APL DataStructures are **Clustered Index based**, and
- The Clustered Index is kept ordered and trim

- DOL DataStructures are **Static Heap based**, and
- The Heap is not ordered, it is not kept trim

17.1 Clustered Index Fresh

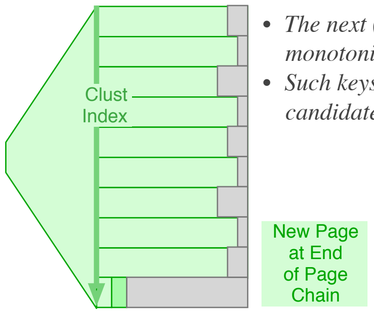


Heap & Placement Index Fresh



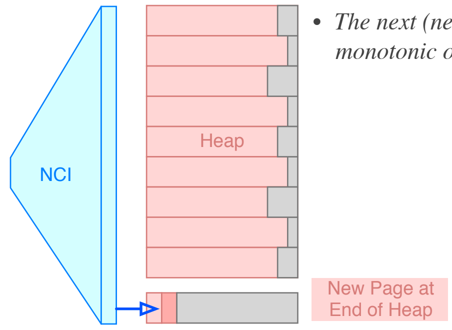
- *Shows that fleeting moment after* REORG, *when the order in the Heap matches the order in the PI*

17.2 Clustered Index Next Sequential Insert



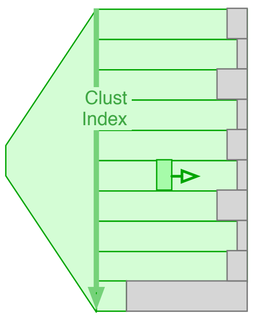
- *The next (new max) value of a monotonic or surrogate Key.*
- *Such keys are the worst candidate for a Clustered Index*

Heap & Placement Index Next Sequential Insert



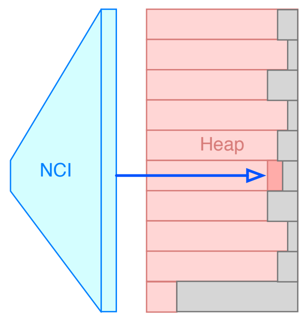
- *The next (new max) value of a monotonic or surrogate Key.*

17.3 Clustered Index Interspersed Insert/Space



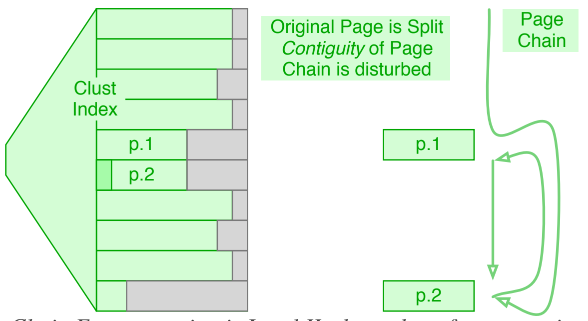
- *A random value of a Relational (composite) Key, where there is space on the page. The rows remain ordered and distributed.*
- *Such keys are the best candidates for a Clustered Index*

Heap & Placement Index Interspersed Insert/Space



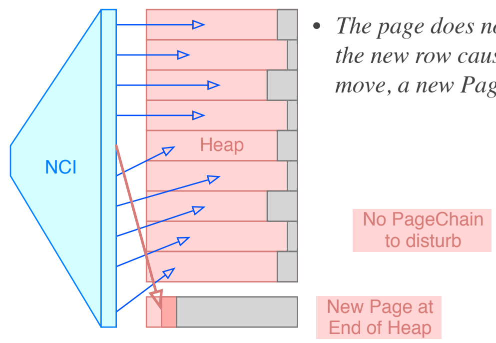
- *A random value of a Relational (composite) Key, where there is space on the page. The rows are not ordered; it is located "near by"*

17.4 Clustered Index Interspersed Insert/No Space



PageChain Fragmentation is Level II, shown here for comparison. In terms of the CI, or logically, the split pages appear next to each other. Physically, the new page is at the end of the structure.

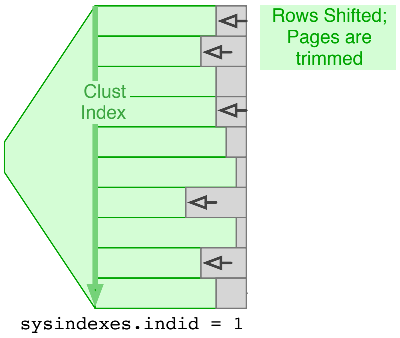
Heap & Placement Index Interspersed Insert/No Space



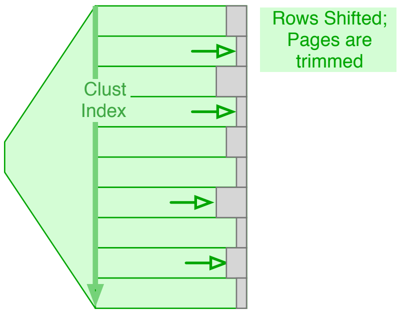
- *The page does not need to be full; if the new row causes existing RowIds to move, a new Page or Extent is used*

AllPage Locked

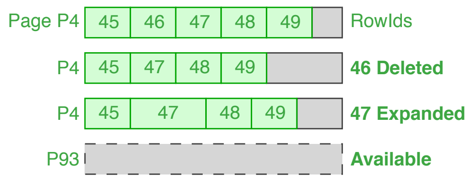
17.5 Clustered Index Interspersed Delete



17.6 Clustered Index Interspersed Update (Expand)

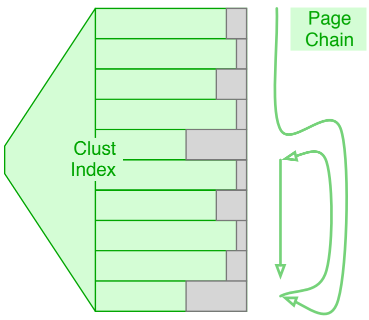


17.7 Clustered Index No Page Fragmentation



Shown here for comparison only. In APL tables there is no Level III Fragmentation, and *the Pages are kept trim*.

18 Level III Summary

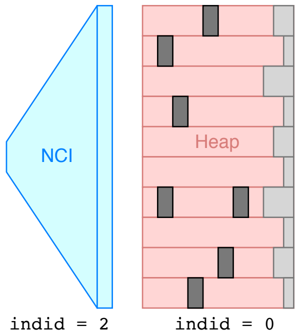


Shown here for comparison only, there is no Level III Fragmentation in APL tables. PageChain fragmentation is Level II.

- No Level III Fragmentation:
  - Deletes are immediate, there is no dead space
  - Row expansion is in place, there is no Row Forwarding
  - No `REORG` required
- There are therefore *two* levels of difference between APL and DOL DataStructures, regarding maintenance or performance

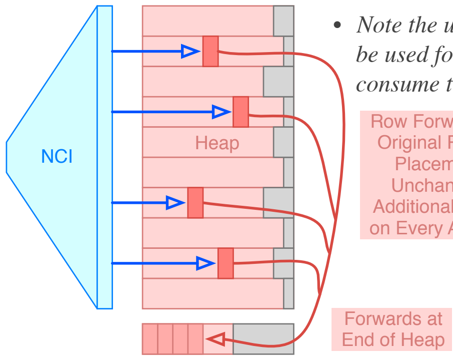
DataPage/DataRow Locked

Heap & Placement Index Interspersed Delete



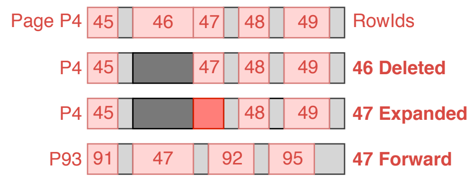
- *Note the unused space; it cannot be used for new rows.*

Heap & Placement Index Interspersed Update (Expand)

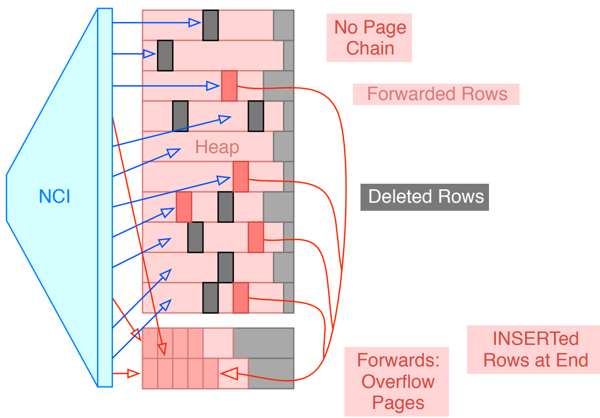


- *Note the unused space; it cannot be used for new rows. Forwards consume twice the space.*

Heap & Placement Index Page Fragmentation



Level III Summary



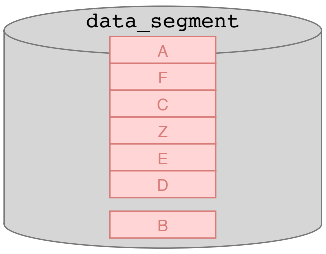
- Level III Fragmentation:
  - Deleted row positions are not reused: dead space
  - Row expansion causes Row Forwarding (twice the space usage)
  - Regular `REORG REBUILD` is demanded
  - Substantial additional space requirement

19 Index Type

SOFTWARE GEMS

AllPage Locked

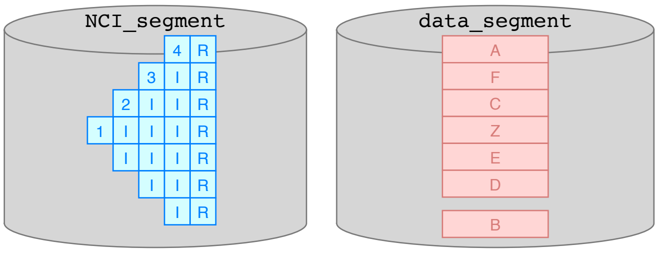
19.1 Heap (When No Clustered Index)



sysindexes.indid = 0

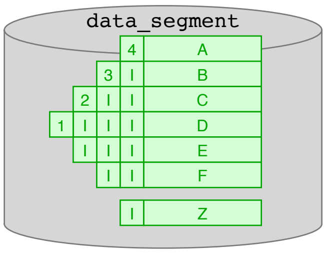
- Chronological (INSERT) order

19.2 Heap plus NCI (When No Clustered Index)



sysindexes.indid = 2 sysindexes.indid = 0

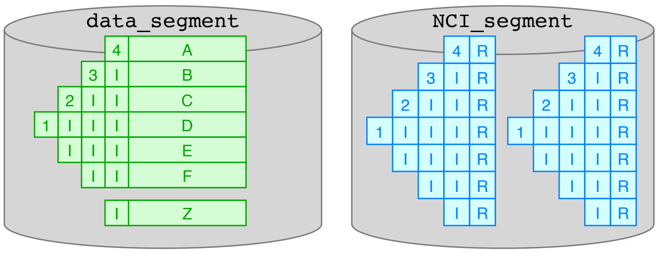
19.3 Clustered Index



sysindexes.indid = 1

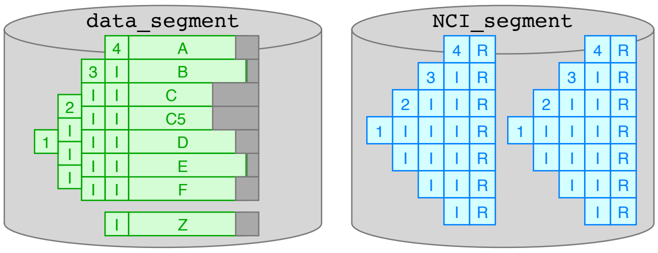
- Rows are maintained in Clustered Index order
- The Heap is eliminated
- Pages & Extents are trimmed
- One less I/O on every access.
- RowIds may change on interspersed INSERT/DELETE/UPDATE (Expand/Shrink)

19.4 Clustered Index plus NCI



sysindexes.indid = 1 sysindexes.indid = 2 & 3

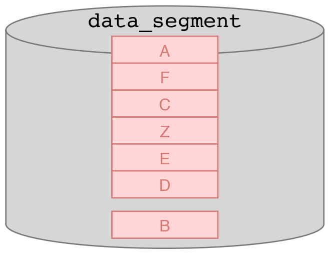
19.5 No Level III Fragmentation



sysindexes.indid = 1 sysindexes.indid = 2 & 3

DataPage/DataRow Locked

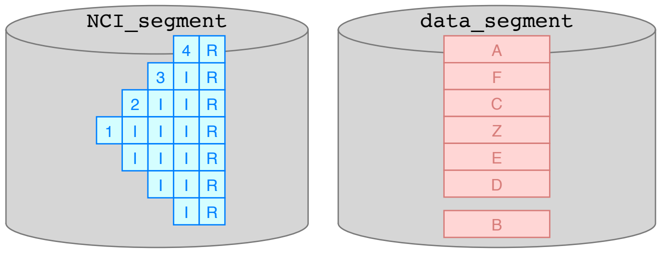
Heap (Always)



sysindexes.indid = 0

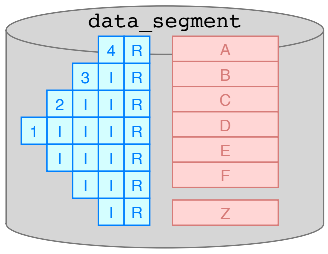
- Chronological (INSERT) order

Heap plus NCI (No Placement Index)



sysindexes.indid = 2 sysindexes.indid = 0

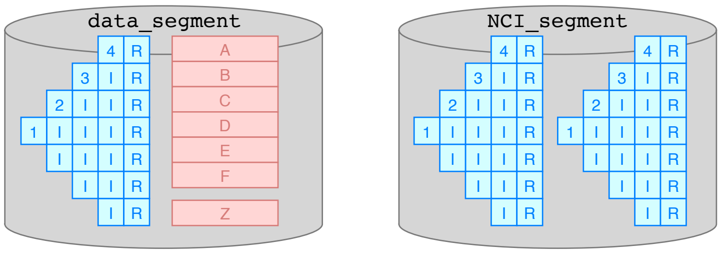
Heap & Placement Index



sysindexes.indid = 0 & 2

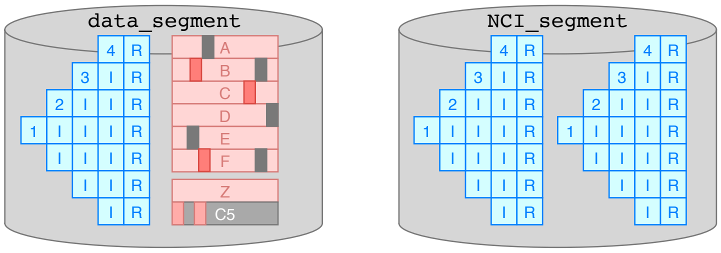
- RowId based, RowIds do not move
- The Placement Index and the Heap remain separate storage (sysindexes) structures, but on the same Segment.
- Rows are **placed** in order initially (but that cannot be maintained under DML activity)
- Rows are ***not*** shifted on INSERT/DELETE/UPDATE (Expand/Shrink)

Heap & Placement Index plus NCI



sysindexes.indid = 0 & 2 sysindexes.indid = 3 & 4

Level III Fragmentation



sysindexes.indid = 0 & 2 sysindexes.indid = 3 & 4

APL Dis/Advantage

- Extents and Pages are kept trim, to maintain contiguity
- RowIds change if Page is split or row is expanded:
  - NCI entries need to be updated *if the RowId in the CI changes*
- Clustered Index & Page Chain allows Range Queries
- No Level III fragmentation; REORG (offline maintenance) is not required

DPL/DRLDis/Advantage

- Row Ids do not change: • Rows do not move
- No Page Chain
- No Range Queries
- Becomes heavily fragmented (Level III) over time
  - Expanded rows are forwarded
  - Inserted rows placed at end of Heap
  - Deleted rows are not deleted (only marked for deletion)
  - Regular de-fragmentation via REORG REBUILD (offline) is required
  - REORG RECLAIM_SPACE & FORWARDED_ROWS are ineffective in correcting Level III Fragmentation

Indices are B-Trees:

- 4 4 Level, Index Height
- I I Intermediate Level
- Z CI Leaf: Data row
- R NCI Leaf: RowId

Only DOL tables are afflicted by Level III Fragmentation, which is shown here in summary form:

- F Deleted Rows
- G Forwarded Rows
- C5 Forward

Full Detail

Intro Unit DataStruct Defn II Determ III Determ I AllocUnit I Segment II PageChain II Unused III Page

