# Database Systems

## WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

SAARLAND UNIVERSITY COMPUTER SCIENCE Information Systems Group

## Topics (1/6)

- fundamental system concepts
- storage media
  - disk
  - flash
  - main memory
- storage management
  - principles
  - page/block mapping and replacement
- data layout - mapping data items to pages
  - vertical
  - horizontal
  - column grouping
  - hybrid mappings, PAX, fractured mirrors
  - compression
  - free memory management



SAARLAND UNIVERSITY COMPUTER SCIENCE Information Systems Group

# Data Layout - Mapping Data Items to Pages.

SAARLAND UNIVERSITY COMPUTER SCIENCE Information Systems Group

## Overview

- So far we considered how to organize pages/blocks in memory and on external storage
- however, data items may be much smaller than a page
  - tuples
  - objects
  - graph nodes
  - XML nodes
  - etc.
- Therefore: we need to think about the mapping from data items to blocks/pages
- **Note:** This is not trivial and may have huge impact on overall system performance.

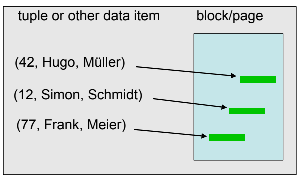


SAARLAND UNIVERSITY COMPUTER SCIENCE Information Systems Group

## Data Item Management

**Tasks:** Mapping from data items to pages/blocks

- Agenda
  - structure of a page
  - tuple addressing
  - tuple mapping
  - tuple layout
  - storage models
    - NSM
    - DSM
    - PAX
    - Fractured Mirrors
  - compression
  - long fields
  - memory management



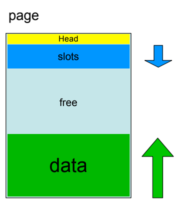



SAARLAND UNIVERSITY COMPUTER SCIENCE Information Systems Group

# Tuples-IDs.

## Structure of a Page

- A page consists of three parts:
  - head (meta data, e.g. page id, log sequence number: see recovery section)
  - slots (pointers to tuples)
  - data (tuple data)
- slot = (pointer, size of the tuple)
- space for slots is allocated top-down
- space for tuple data is allocated bottom-up

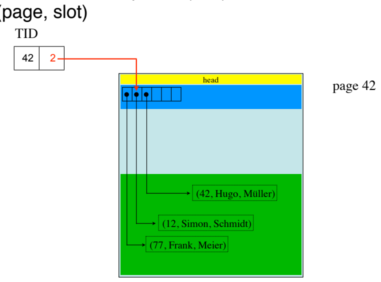



**Advantage: tuple may easily migrate inside a page**



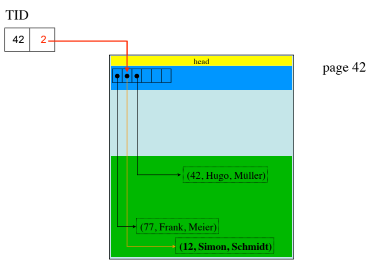
## Tuple IDs (TID)

- Indirection based on tuple-ID (TID)
  TID = (page, slot)



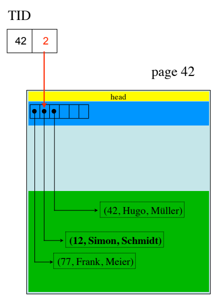



## Migration of Tuples Inside a Page



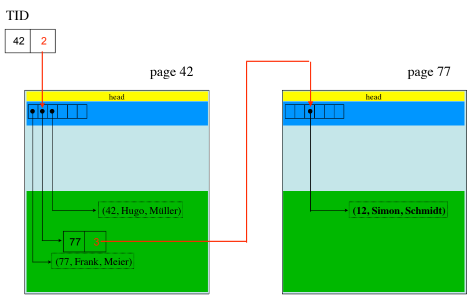



## Migration of a Tuple to Another Page






## Migration of a Tuple to Another Page






## Discussion

- access trivial if tuple does not migrate to a new page
- migration to other page using forward TIDs
- if migrated tuple migrates again: update first forwarding TID

  ⇨ at most one indirection caused by TID

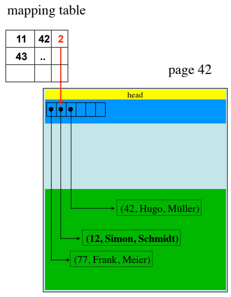
- performance
  - at least one page access required
  - at most 2 page accesses required
    (if forward TID has to be followed)

## Indirect Addressing: Mapping Table

- Idea:
  - 1. keep a separate mapping table
  - 2. hide physical addresses
    (outside world only knows logical addresses)
  - no forwarding
  - if tuple needs to be moved: change entry in mapping table

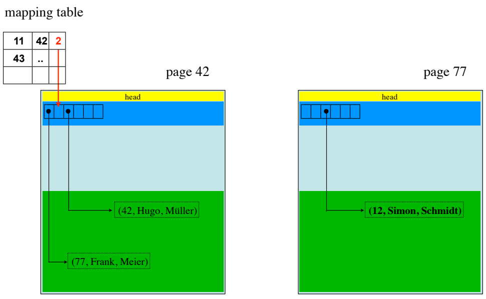


## Migration of Tuples: Mapping Table



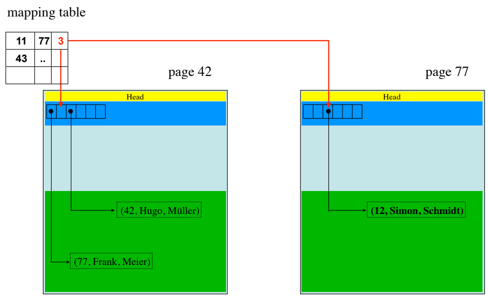



## Migration of Tuples: Mapping Table






## Migration of Tuples: Mapping Table





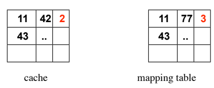

## Mapping Table

- Drawbacks of a mapping table:
  - 2 block accesses (1 mapping table block + 1 data block)
- Advantage:
  - no space wasted for forward TIDs



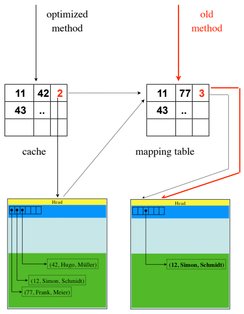
## Mapping Table (optimized, aka PPP)

- Drawbacks of a mapping table:
  - 2 block accesses (1 mapping table block + 1 data block)
- Optimization:
  - Access to mapping table can be avoided if frequently accessed entries are kept in a separate cache in main memory



## Mapping Table (optimized, aka PPP)

```
Algorithm to find a tuple
   IF tuple address contained in cache:
      no I/O access on mapping table
      address := cache-address
      if tuple was not found at address:
         I/O-access on mapping table
   ELSE
      I/O-access on mapping table
```








## Further Optimizations

- Observation: Mapping table corresponds to a big index page!
- So why not store the mapping

  logical tuple address ⟶ physical tuple address

  in an index structure in the first place, e.g., a B+-tree?
- Discussion
  - Advantages:
    - implicit ordering of entries guaranteed
    - no additional memory management required
  - Disadvantages:
    - expensive access using a multi-level tree structure (for each tuple access) --> Thus we should not do this!
    - memory usage





# Tuple Layout.



## How to Store Values: Data and Metadata

- Separation of data and metadata
  - **Metadata**: data in DB catalogue
    - attribute name
    - type
  - **Data**: data on each page/block
    - actual values
- Note:
  - In XML data and metadata are stored together:

    <tuple>
    <firstname> hugo </firstname>
    <lastname> müller </lastname>
    </tuple >





## Tuple Layout

- fixed-sized part:
  - stores all values having a type of fixed size
  - e.g. numeric(10,2), date, char[42]
  - Advantage: direct addressing
    address = sizeof(type) * pos
- variable-sized part:
  - e.g. varchars
  - store size and pointer in fixed-size part
  - store actual values in variable-sized part
  - Disadvantage: indirect addressing
    address = pointer
  - Important: if variable-sized types are used for any attribute of a tuple the direct addressing of the fixed-sized part is still possible!





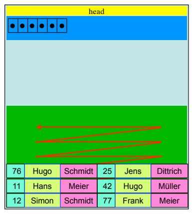
## Tuple Layout

- NULL-values
  - small bitmap of fixed size at the beginning of each tuple
  - “1” if attribute is set to NULL, else “0”
  - Advantage: simple and efficient

# Column, Row, and Hybrid Mappings.

## Data Item Mapping

| Key | fname | lname |
|---|---|---|
| 77 | Frank | Meier |
| 12 | Simon | Schmidt |
| 42 | Hugo | Müller |
| 11 | Hans | Meier |
| 25 | Jens | Dittrich |
| 76 | Hugo | Schmidt |

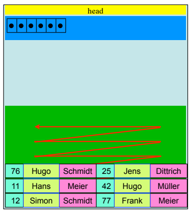



row-wise assignment of tuple values to page



## n-ary Storage Model (NSM)

| Key | fname | lname |
|---|---|---|
| 77 | Frank | Meier |
| 12 | Simon | Schmidt |
| 42 | Hugo | Müller |
| 11 | Hans | Meier |
| 25 | Jens | Dittrich |
| 76 | Hugo | Schmidt |




- tuple values are assigned row-wise to page
- all attribute values of a tuple are adjacent on the page



## Decomposition Storage Model (DSM)

| RID | Key | fname | lname |
|---|---|---|---|
| 1 | 77 | Frank | Meier |
| 2 | 12 | Simon | Schmidt |
| 3 | 42 | Hugo | Müller |
| 4 | 11 | Hans | Meier |
| 5 | 25 | Jens | Dittrich |
| 6 | 76 | Hugo | Schmidt |

| RID | Key |
|---|---|
| 1 | 77 |
| 2 | 12 |
| 3 | 42 |
| 4 | 11 |
| 5 | 25 |
| 6 | 76 |

| RID | fname |
|---|---|
| 1 | Frank |
| 2 | Simon |
| 3 | Hugo |
| 4 | Hans |
| 5 | Jens |
| 6 | Hugo |

| RID | lname |
|---|---|
| 1 | Meier |
| 2 | Schmidt |
| 3 | Müller |
| 4 | Meier |
| 5 | Dittrich |
| 6 | Schmidt |

- split table into set of two-column tables
- alternatively: split table into one-column table storing the RID implicitly (array-like representation)

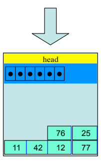


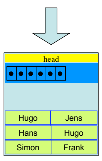
## Decomposition Storage Model (DSM)

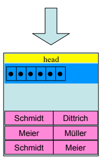
| RID | Key |
|---|---|
| 1 | 77 |
| 2 | 12 |
| 3 | 42 |
| 4 | 11 |
| 5 | 25 |
| 6 | 76 |

| RID | fname |
|---|---|
| 1 | Frank |
| 2 | Simon |
| 3 | Hugo |
| 4 | Hans |
| 5 | Jens |
| 6 | Hugo |

| RID | lname |
|---|---|
| 1 | Meier |
| 2 | Schmidt |
| 3 | Müller |
| 4 | Meier |
| 5 | Dittrich |
| 6 | Schmidt |



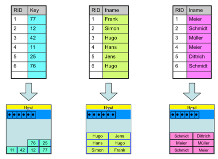





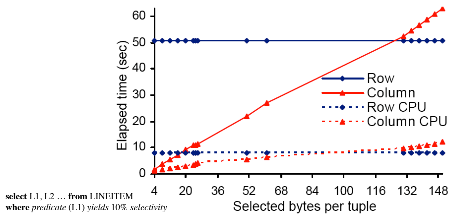

## Decomposition Storage Model (DSM)

- optimized for accessing few attributes
- Advantage: very efficient when only few attributes need to be accessed
- Disadvantage: inefficient when many attributes are accessed
- Disadvantage: tuple information distributed to several pages => seeks



- Literature
  - Don S. Batory: On Searching Transposed Files. ACM Trans. Database Syst. 1979.
  - George P. Copeland, Setrag Khoshafian: A Decomposition Storage Model. SIGMOD 1985

## Some Recent Study on Column Stores




- Tuple width: 150 bytes, 16 attributes, 9.5GB table
- Literature: Performance Tradeoffs in Read-Optimized Databases, Harizopoulos et.al, VLDB 2006





## Column Stores

- Several Products
  - Sybase IQ (since early 90ies)
  - Applix
  - Monet DB (main memory)
  - SAP BI Accelerator (main memory)
  - Vertica (main memory)
  - ...
- Will become more and more important given current hardware trends
- Student in 1995: What is tape?
- Student in 2010: What is a hard disk?





## Fractured Mirrors

- Idea:
  - keep both representations row-store and column store
  - depending on type of query pick appropriate store
- This idea is related to indexing.
- We will come back to this in the context of projection and bit-sliced indexes.
- Literature:
  - Ravishankar Ramamurthy, David J. DeWitt, Qi Su: A case for fractured mirrors. VLDB J. 12(2): 89-101 (2003)





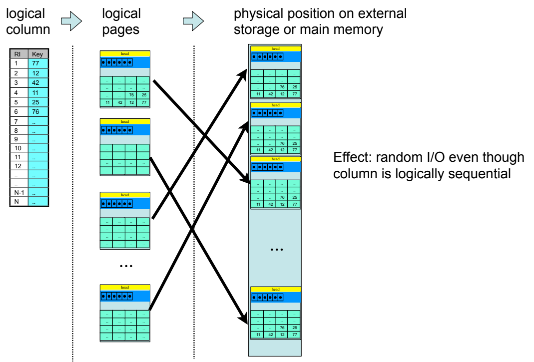
## Note: Intra- versus Inter-page Sequential Layout

- DSM groups data by attribute (intra-page mapping)
- effect for query processing:
  - only pages containing those attributes need to be loaded
  - total number of pages loaded is reduced
  - total amount of data loaded into the caches is reduced
- however keep in mind:
  - whether all pages pertaining to the same attribute are sequentially stored on external memory (e.g., disk) is determined at a different level: the storage manager (inter-page mapping)





## Intra-page Non-Sequential Layout



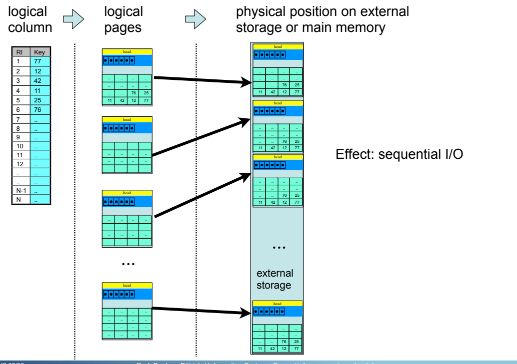





## Note: Intra- versus Inter-page Sequential

- Example:
  - consider you need to read all values for a given attribute a42
  - => need to read all blocks containing values for attribute a42
  - if blocks pertaining to attribute a42 are not sequentially stored on disk, this may in the worst case trigger one random I/O per block!
  - sequential layout on external storage would be great in this case...

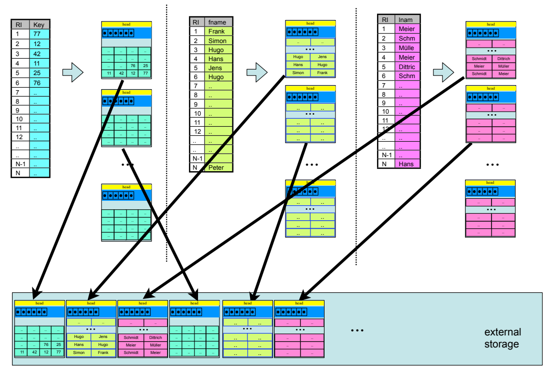
## Intra-page Sequential Layout








## Column Grouping








## Column Grouping

- Main effect: inter-page locality among tuples (like NSM...)
- In contrast to NSM no gain in terms of I/O
- Inside a page sequential layout: good for cache hierarchy if you do main memory processing page-wise (less cache misses)
- However:
  - Consider some of the attributes are frequently accessed together
  - then you may adapt the grouping to this
  - keep frequently accessed columns grouped
  - keep infrequently accessed columns as separate columns
- In addition, if you have frequent access to attributes (a,b,c) and (e,b,f), you may even replicate b and store it in both groups.

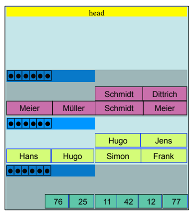




## Partition Attributes Across (PAX)

- Another Idea: co-locate values of the same attribute **inside** a page

| Key | fname | lname |
|---|---|---|
| 77 | Frank | Meier |
| 12 | Simon | Schmidt |
| 42 | Hugo | Müller |
| 11 | Hans | Meier |
| 25 | Jens | Dittrich |
| 76 | Hugo | Schmidt |




subpage 1

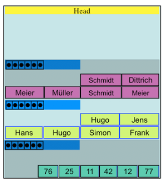
subpage 2

subpage 3





## Partition Attributes Across (PAX)




- Advantages
  - improves locality for single attributes
  - data values are reorganized inside a page only
    ⇨ no change to the outside system (if appropriate information hiding was used.)
  - tuple reconstruction cheap
  - 15%-2x performance improvements when compard with NSM
- Disadvantages
  - not the best possible solution for decision support (DSS, OLAP)
  - DSM wins...
- Literature: Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, Marios Skounakis: Weaving Relations for Cache Performance. VLDB 2001.





## How to Map Long Tuples

- Problem:
  - What if a tuple is larger than a page?
  - Example: Blobs (Binary Large Objects)
- 1. Solution
  - split tuple into pieces of size page_size
  - create index (byteoffset ⟶ block) for pieces using a hash-table or a b+-tree
- 2. Solution
  - store tuple in separate storage space (e.g. file system of the OS)
  - store only a link to the separate storage on the DB-page

# Compression.




## Compressing Data to External Storage

- in addition to the methods shown so far, we may compress data before mapping it to external storage
- this means
  - when storing data on external storage we compress the data
  - when loading data from external storage we decompress the data
- disadvantages
  - CPU time to compress data (possibly at update time)
  - CPU time to decompress data (possibly at query time)
- advantages
  - less space used on external storage
  - less money spend for external storage
  - **better overall performance** if (de-)compress faster than gain in saved I/O



## Lossless Versus Lossy Compression

- most type of information systems will require lossless compression:
  - RLE
  - type compression
  - dictionary-based compression
  - LZW
- otherwise correcteness cannot be guaranteed
- lossy algorithms may however work in cases when correctness is not required:
  - multimedia data, e.g., image search (VA-file)
  - data mining: clustering, outlier detection, sampling



## Lossless Versus Lossy Compression

- also lossy indexing:
  - lossy compression considered a prefiltering step
    - retrieve candidate set with lossy index
    - compute precise result from candidates
  - has to make sure that lossy index retrieves superset of the actual result
  - otherwise: result will not be correct
- in general strong relationship between compression and indexing...
- most obvious example: huffman codes
- other examples: z-codes (see indexing slides later on)



## Performance Gain Example 1

- assume
  - large sequential scan
  - 100 MB/s bandwidth
  - 3 GB data
  - => 30 seconds to scan
- compression ratio 1:3
  - just 10 seconds I/O-time to scan (1 GB to read)
  - however plus CPU time to decompress
  - can you decompress all that data within 30 sec CPU time? (note: overlap of CPU and I/O-time)
  - let's say we have a single 3 GHz CPU
  - thus we may spent up to **90 clocks per byte** to decompress and would still be better than uncompressed...



## General Trade-Off

- De-/Compression Time versus Compression Ratio
- In general: the higher the compression ratio the more expensive it becomes to compress and decompress the data
- methods like ZIPF/LZW usually too slow
- requires a method with a "good" trade-off
- lightweight-compression
- not the perfect space gain, yet will not ruin query performance
- choice of methods depends on storage scenario, type of information system...

## Performance Gain Example 2

- assume
  - query processing algorithm operating on large amounts of data
  - algorithm needs to store temporary data on disk
  - for instance:
    - external sorting needs to store and read initial and/or intermediate "runs"
    - external partitioning needs to store and read initial and intermediate "partitions"
  - in the end: manage **external queues of data on disk**
  - again: external queues basically lead to sequential I/O
- compression ratio 1:3
  - same considerations as in Example 1
  - however: we also have to compress the data
  - therefore
  - I/O of compressed data plus CPU time for compress and decompress
  - sum of this should be faster than I/O of uncompressed data



## Additional Advantage: Main Memory DB

- for some systems compression may make the difference to letting the system become a main memory system!
- For example: consider you have 300 GB uncompressed data but 100 GB of main memory only
- instead of thinking about external memory structures, you should first think of compressing the data in a way that everything fits into main memory
- the gain in terms of query performance will be so huge that almost any compression technique will work



## Compression without Decompression

- so far: data needs to be uncompressed at query time
- other idea: do not decompress data but rather adapt query processing methods to directly operate on compressed data
- advantage:
  - no time lost for decompressing
- disadvantage:
  - adaption of query processing methods required
- Literature: Till Westmann, Donald Kossmann, Sven Helmer, Guido Moerkotte: The Implementation and Performance of Compressed Databases. SIGMOD Record 29(3) (2000)



## Additional Advantage: Less Cache Misses

- First effect we saw: less data => less I/O on external storage (sometimes no I/O anymore) => better I/O performance
- Second effect: less data => more data fits into caches => less cache misses => better CPU performance
- Example:
  - consider you have a compression ratio of 1:3
  - L1 and L2 caches may now contain three times more data
  - => probability to get a cache misses decreases
- Note
  - you may use different compression schemes for the two levels
  - (1) main memory to caches: optimized for less cache misses
  - (2) external storage to main memory: optimized to get less I/O
- Literature: Marcin Zukowski, Sándor Héman, Niels Nes, Peter A. Boncz: Super-Scalar RAM-CPU Cache Compression. ICDE 2006



## Compression Granularity

- individual attribute values
- individual tuples
- entire pages
- entire extents
- DSM may be compressed easily
- Literature (selection):
  - Meikel Pöss, Dmitry Potapov: Data Compression in Oracle. VLDB 2003.
  - Balakrishna R. Iyer, David Wilhite: Data Compression Support in Databases. VLDB 1994.
  - **Managing Gigabytes**: Compressing and Indexing Documents and Images by Ian H. Witten, Alistair Moffat, and Timothy C. Bell. 2nd edition. Morgan Kaufmann Publishing. 1999.



# Free Memory Management.

## Free Memory Management: Append Only

- Task: Find a free slot for a newly inserted tuple
- Naive implementation (append only):
  - only consider the last created page
  - if tuple fits into this page: OK
  - Else: create a new page
- Discussion
  - very efficient insert
  - poor memory usage (deletes?)



## Append Only(n)

- Generalization of append only:
  - only consider the n last created pages
  - if tuple fits into one of this pages: OK
  - Else: create a new page
- Discussion
  - very fast insert
  - still: poor memory usage (deletes?)



## Best Fit, First Fit, Next Fit

- Best Fit:
  - start search from the beginning of the page list
  - find optimal page
  - Cost = linear search in list
- First Fit:
  - start search from the beginning of the page list
  - take the first page that fits
  - disadvantage: pages at the beginning of the list will soon be full (waste of time to search through these pages)
- Next Fit:
  - like first fit, but: start search from the last position



## Hybrid Approaches HY(n,u)

- Algorithm:

  If memory usage better than u:

  use append only(n)

  Else

  use Next Fit
- Discussion: acceptable compromise?



## Free Memory Table

- Idea: for each page: store available memory
- Implementation: extra table in segment either

1. providing accurate memory information

| | | | | | | |
|---|---|---|---|---|---|---|
| **page** | 1 | 2 | 3 | 4 | 5 | |
| **bytes avail.** | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | |

2 byte/entry

2 byte/entry

or 2. providing approximate memory information

| | | | | | | |
|---|---|---|---|---|---|---|
| **page** | 1 | 2 | 3 | 4 | 5 | |
| **bytes avail.** | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | |

2 byte/entry

**k bits**/entry

memory available >= $(f_i / 2^k)$ * page_size



## Space Map

- Drawback of free memory table: linear search to find suitable page
- Solution “space map”:
  - inversion of the free memory table

| | | | | | | |
|---|---|---|---|---|---|---|
| **page** | 1 | 2 | 3 | 4 | 5 | |
| **bytes avail.** | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | |

⇒

| | | | | | |
|---|---|---|---|---|---|
| **bytes avail.** | $f_4$ | $f_2$ | $f_3$ | $f_5$ | |
| **page** | 4 | 1,2 | 3 | 5 | |

  - index on “ bytes avail.” attribute
  - binary search
  - works for accurate and approximate variant

## Space Map

- Trade-off: granularity vs. memory needed
- Trade-off: granularity vs. performance
- Advantage: efficient search for best fit: O(log n)
- Disadvantage: maintenance cost for space map





# Defragmentation.



## Fragmented Data Layout

- in the long run update and insert operations will lead to fragmentation
- similar problems as for file systems in operating systems:
  - fragmented data on disk may be a major bottleneck in overall system performance
  - my current Macbook is a negative example:
    (Note I really love Macs but the current version of the storage system is just crap.)
  - massive slow-down due to tons of random I/O operations
  - rule of thumb: if we decrease numer of random IOs, we increase overall system performance





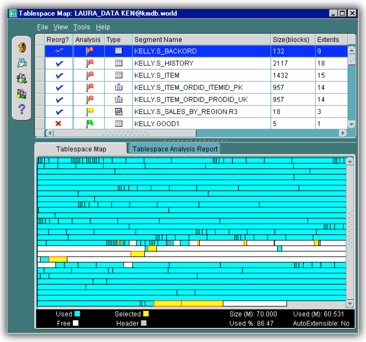
## How to Defrag

- physical layout on external memory should be optimized for access patterns
- we have already seen an example with columns
- in general: sequential layout is a good idea
- however, you may use any layout which gives the best performance
- Example
  - booting an operating system in single threaded mode
  - this will always lead to the **same** sequence of page accesses
  - so why not record the block sequence once
  - then layout blocks sequentially on disk
  - effect for next boot: no random I/O at boot time
  - I am not aware of any OS doing this. It would be so simple.





## Example: Oracle Tablespace Map



- visualizes data layout
- helps you to understand performance problems
- allows you to defrag data

- source: http://www.oracle.com/technology/products/oracle9i/datasheets/oem_tuning/tuning.html





# Next Topic: Indexing.