TRAINING & CONSULTING

Physical Data (Re)Organisation

Oracle Benelux User Group Meeting, Maastricht, 2012

Kris Van Thillo, ABIS

ABIS Training & Consulting
www.abis.be
training@abis.be

2012

Document number: OraUserMaastricht2012Front.fm
25 April 2012

Address comments concerning the contents of this publication to:
ABIS Training & Consulting, P.O. Box 220, B-3000 Leuven, Belgium
Tel.: (+32)-16-245610, Fax: (+32)-16-245639

Physical Data Organization

- Discuss physical data organisation
- Discuss reorganisation techniques

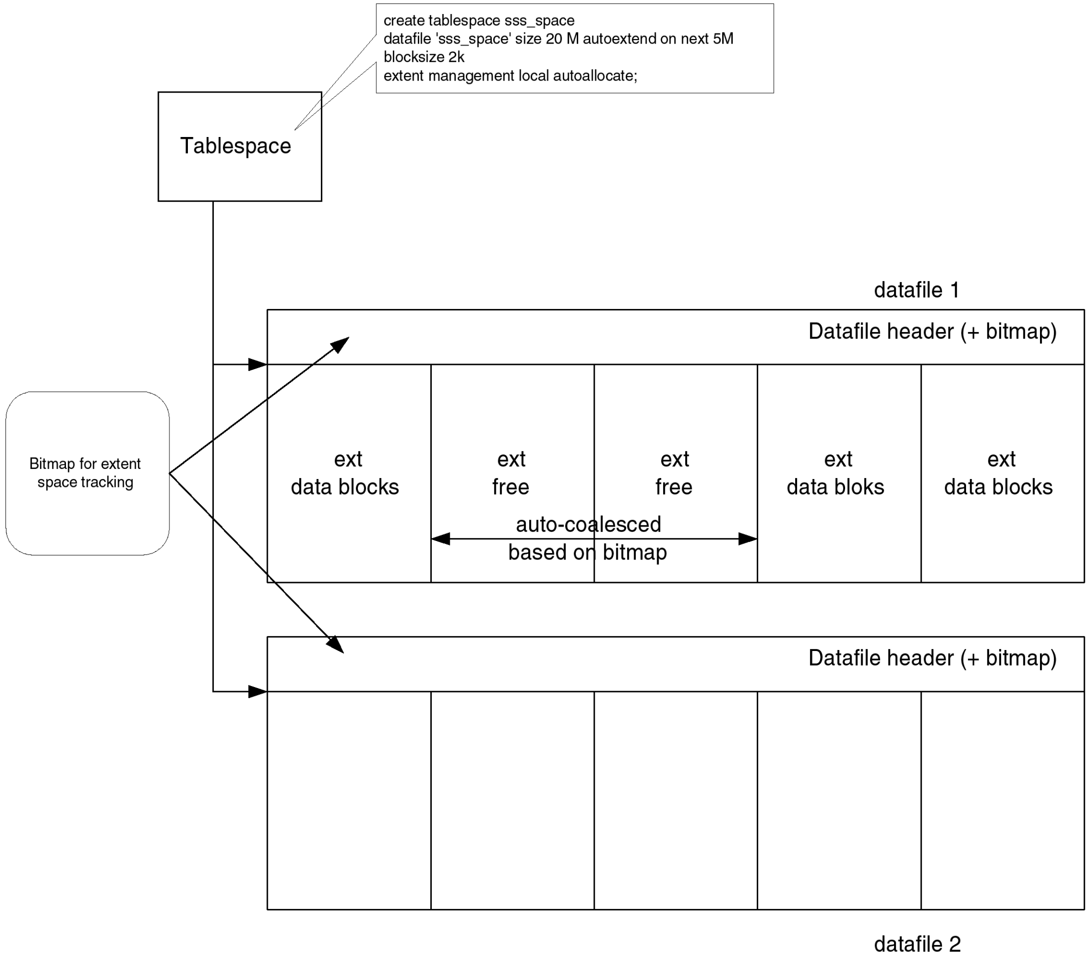
Locally managed tablespaces

Extent management allocation within tablespaces:

- **old - style** [deprecated] **-** dictionary managed **tablespaces**
- **today, use** locally managed **tablespaces:**

Locally managed
extent allocation tracking through a bitmap in the tablespace header
no recursive sql required - bitmap keeps track of free/used space
free space tracked using bitmaps - sequence of free space corresponds with sequence of free bits: no coalesce required (auto coalesce)
multi threaded
enforces extent size uniformity at the tablespace level - minimal diversity

Locally managed tablespaces (2)



Physical Data Organization

Locally managed tablespaces (3)

Automatic extent allocation	Unifom extent allocation
Oracle deals with extent sizing automatically at the object level	DBA specifies extent size on tablespace level - unconditionally applied by Oracle
4 different extent sizes used on object level - link between object size and extent size is as follows: object size — extent size [0-1 M] — 64 K [1 M - 64 M] — 1 M [64M - 1 GB] — 8 M [> 1 GB] — 64 M all extent sizes are multiples of each other!	one uniform extent size used for the entire tablescape can not be overruled for any object in the tablespace any isolated free extents can be reused - all extents in the tablespace have the same size!
Minimal DBA control	

dba_tablespaces.*extent_management* and *allocation_type*
dba_data_files.*blocks* and *user_blocks* -> bitmap overhead explains difference

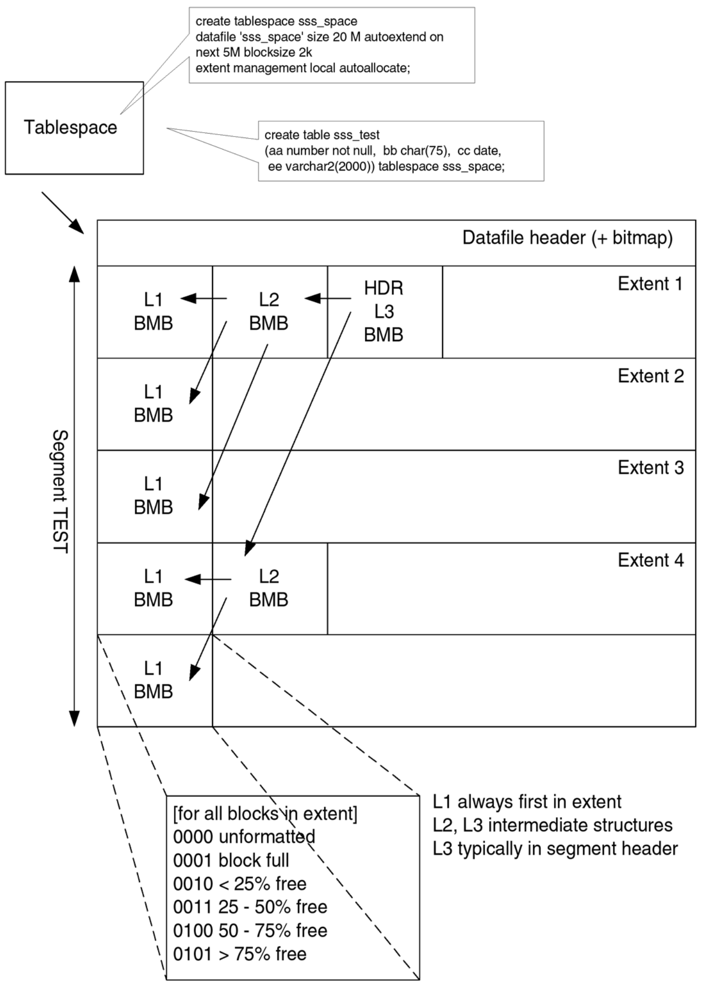
Automatic space management

Space management within segments/extents:

- **old - style** [deprecated] **-** segment space management using ‘free lists’
- **today, choose** automatic space management

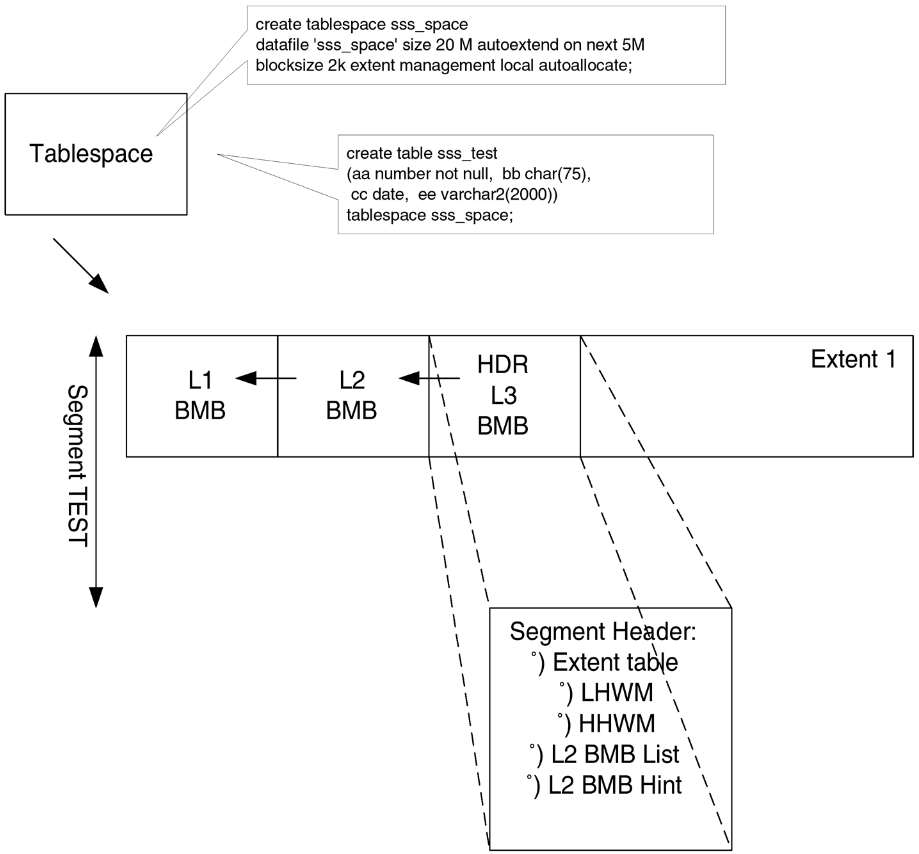
Automatic space management
space management within segments/extents using bitmaps representing block free space
granular and accurate overview of space usage within a block
no more system ‘freezing’ when space management occurs - new blocks formatted on a ‘as needed’ basis
bitmaps stored in bitmap blocks (BMBs) - number of BMBs decreases as data blocks increase

Automatic space management (2)



dba_tablespaces.segment_space_management

Automatic space management (3)



Physical Data Organization

Analysing segment and extent information

- **catalog tables:**
  - **dba_tablespaces, dba_data_files, dba_free_space**
    [allocation data ts/df level]
  - **dba_segments, dba_extents**
    [allocation data seg/ext level]
- **procedures:**
  - **dbms_space.space_usage**
    [space usage before the HWM]
  - **dbms_space.unused_space**
    [space usage in general, including info about HHWM]

[On the next set of pages, reports are shown based on a wrapper procedure invoking dbms_space.space_usage and dbms_space.unused_space. The first part of the output refers to space_usage, the second part of the output to unused_space. Parts of the report presented in *italic* have been added through manual catalog investigation.]

Sample case (a)(b)

```
(a)
create tablespace sss_space
datafile 'sss_space' size 20 M autoextend on next 5M
blocksize 2k extent management local autoallocate;


(b)
create table sss_test
(aa number not null,  bb char(75), cc date, ee varchar2(2000))
tablespace sss_space;
--
create index ix1 on sss_test(bb) tablespace sss_space;

(b**)
synonym, view and stored procedure created on sss_test as well
```

Sample case (a)(b)

```
                                      (b)
Space use in object
Unformatted Blocks ....................0
FS1 Blocks (0-25) .....................0
FS2 Blocks (25-50) ....................0
FS3 Blocks (50-75) ....................0
FS4 Blocks (75-100)....................0
Full Blocks    ........................0
Free Space in object
Total Blocks..........................32
Total Bytes........................65536
Total MBytes...........................0
Unused Blocks.........................28
Unused Bytes.......................57344
Last Used Ext FileId...................6
Last Used Ext BlockId.................33
Last Used Block........................4
```

<u>(a) initial tablespace creation</u>
dba_tablespaces: extent_management local, allocation_type system
dba_data_files: blocks 10240 [20M/2k], user_blocks 10208 [10240 - 10208 = 32 overhd]
dba_free_space: block_id 33 [free for storing objects as of block 33]

<u>(b) initial object creation (one for table, one for index)</u>
Extent size 2k bl * **32** bl = 64k [expected for local managed TS autoallocate]; **4** bl overhead [last used block]; not accounted for in Space use part of report [dba_free_space: block_id 97 - ie. 32 bl (bitmapadmin) + 32 bl (table) + 32 bl (index) are in use]

High water mark

When reading data, Oracle will typically:

- **use** index - based **access paths to the data**
- **use** table - based **access paths -** table scan

Table scans:

- **are typically performed using** ‘big block reads’ **-** [db_file_multiblock_read_count]
- **read all data upto a so-called** ‘high water mark (HWM)’

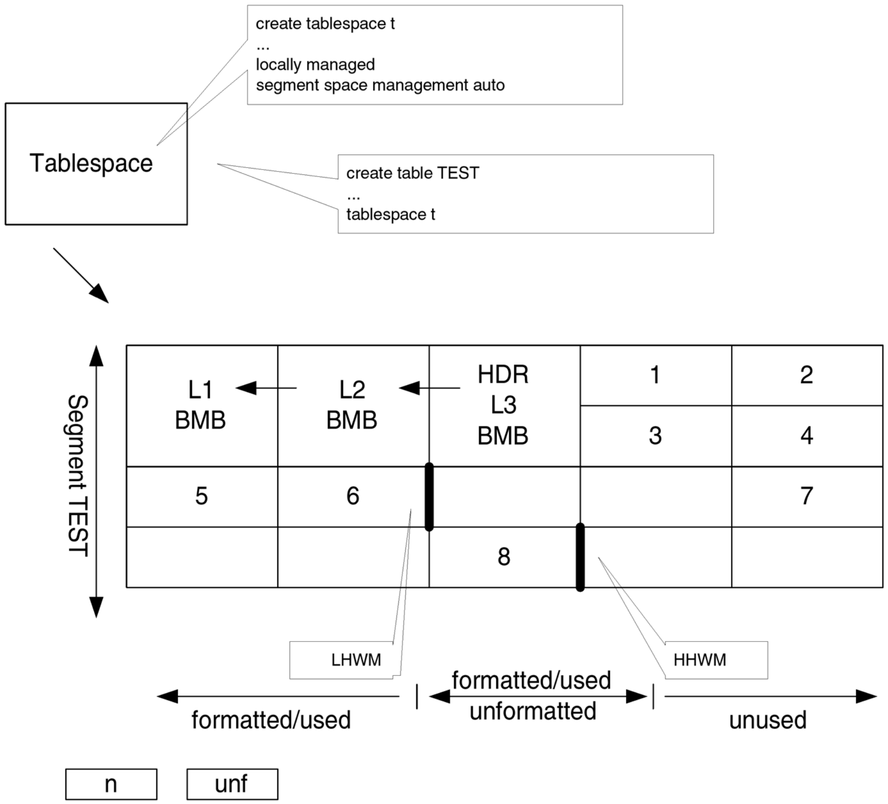
When data is inserted into the table, the HWM gets moved upwards; when data is removed, the HWM does not get reset!

- **LHWM -** all blocks below LHWM contain object relevant data
- **HHWM -** all blocks above HHWM are not relevant for the current object, have never been used, not formatted, ...
- **in between the LHWM - HHWM -** some blocks might be used, some might not be used...

High water mark (2)

- **when *reading* data:**
  - all data upto the LHWM needs to be read
  - all data in between the LHWM - HHWM should be read IF APPROPRIATE, ie. in use, assigned to the concerned object
    bitmap structures are used to track blocks in use for that object
- **when *inserting* data:**
  - blocks below the LHWM are filled-up with object data as long as space permits
  - if no more blocks are available below the LHWM, blocks between the LHWM and HHWM are used:
    -> favour blocks already in use
    -> switch to unused blocks below the HHWM *(format first?)*
  - if no more blocks are available below the HHWM, the LHWM and the HHWM are adjusted, and new 'unused blocks' become available below the HHWM yet above the LHWM
  - extra blocks are formatted when *first used* for storing data
- **when *updating* data:** pctfree
- **when *deleting* data:** HHWM and LHWM does not move

High water mark (3)



Physical Data Organization

Sample case (c)(d)(e)(f)(g)

```
(c)
begin
for i in 1 .. 125000
loop
 insert into sss_test values (i, to_char(i), sysdate + i, to_char(i));
 if mod(i, 250) = 0
    then commit;
 end if;
end loop;
end;
```

```
(d)
alter table sss_test allocate extent;
alter table sss_test deallocate unused keep 500k;
```

```
(e)
add 3000 rows already present using insert into ... select *
```

```
(f)
add 250 rows already present using insert into ... select *
```

```
(g)
insert /*+append */ into sss_test
select * from sss_test where aa < 2500;
```

Sample case (c)(d)(e)(f)(g)

	(c)	(d)	(e)	(f)	(g)
Space use in object					
Unformatted Blocks ...	**144**	144	**0**	**24**	**0**
FS1 Blocks (0-25)	0	0	0	0	0
FS2 Blocks (25-50) ...	1	1	1	1	1
FS3 Blocks (50-75) ...	0	0	0	0	0
FS4 Blocks (75-100)...	38	38	6	27	51
Full Blocks	7352	7352	7528	7539	7845
Free Space in object					
Total Blocks.....	**7680**	**7936**	7936	7936	8448
Total Bytes.....	15728640	16252928	16252928	16252928	17301504
Total MBytes.......	15	15	15	15	16
Unused Blocks.......	0	**256**	256	192	**390**
Unused Bytes.......	0	524288	524288	393216	798720
Last Used Ext FileId...	6	6	6	6	6
Last Used Ext BlockId...	18465	**18465**	18465	**19489**	20001
Last Used Block....	512	**512**	512	**64**	**122**
#extents	*30*	*32*	*32*	*32*	*33*
LastExtentStartBlock	*18465*	*18465*	*18465*	*19489*	*20001*
SizeLastExtent	*512*	*512*	*512*	*256*	*512*

<u>(c) adding 125000 rows to table using simple insert</u>
Blocks allocated: **7680** (verif dba_extents: 16 ext @ 32bl + 14 ext @ 512bl = 7680 bl)
144 blocks above LHWM and below HHWM not formatted; no blocks above the HHWM
The start block*id* of the *last extent* with data is 18465 (verif dba_extents.block_id), with last used block in that extent block# 512, the last one.

Sample case (c)(d)(e)(f)(g)

(d) allocate a next extent and deallocate keep unused 500k
Blocks in use: from 7680 to **7936** -> 256 bl @ 2k = 512 k, the requested 'keep unused' size. The HHWM did not change (block # 512 in extent start 18465).

(e) add 3000 rows already existing in the table, to that same table
Blocks between the LHWM and the HHWM are being used (unformatted decreases upto **0**); no additional blocks are allocated!

(f) add 200 rows already existing in the table, to that same table
A new extent is being used (with starting block id **19489** and a size of **256** - remember (c)); part of it (**64** bl) is being used (moved below the HHWM), the remainder is not used (above the HHWM, 192 bl). Between the LHWM and the HHWM, 24 bl are not being used.

(g) direct-load 5200 rows into the table
[direct load always resync's LHWM, HHWM]

High water mark (4) - it's all about cheese...

- **when performing a scan, all data has to be read upto the HWM**
- **the more 'free space' available below the HWM** [swiss cheese]**, the more free, useless space needs to be read when scanning the data**

  ***inefficient* scan**

- **the less 'free space' available below the HWM** [dutch cheese]**, ...**

  ***efficient* scan**

Sample case (h)

```
(h)
update sss_test
set ee = rpad('a', 1000, 'x')           -- ee => varchar2(2000)
where aa between 2500 and 7500;
commit;
update sss_test
set ee = rpad('a', 1900, 'x')
where aa between 25000 and 30000;
commit;
delete from sss_test
where mod(aa, 150) = 0;
commit;
delete from sss_test
where mod(aa, 242) = 0;
commit;
update sss_test
set bb = rpad ('a', 50, 'x')            -- bb => char(75)
where mod (aa, 250)=0;
commit;
delete from sss_test
where cc < sysdate + 1500
or cc > sysdate + 85000;
commit;
```

Sample case (h)

```
                               (g)          (h)
Space use in object
Unformatted Blocks ..............0........360
FS1 Blocks (0-25) ...............0.........32
FS2 Blocks (25-50) ..............1.......5085
FS3 Blocks (50-75) ..............0.........19
FS4 Blocks (75-100).............51.......2905
Full Blocks     ..............7845......11980
Free Space in object
Total Blocks..................8448......20736
Total Bytes...............7301504...42467328
Total MBytes....................16.........40
Unused Blocks..................390..........0
Unused Bytes................798720..........0
Last Used Ext FileId.............6..........6
Last Used Ext BlockId........20001......32289
Last Used Block................122........512
```

(g) after loading the table

(h) number of update/delete operations performed; observe space usage increases dramatically!

High water mark (4) - it's all about cheese... (cont.)

How to control/influence space usage below the HWM?

[because we need to *control space usage*]
[because we need to *increase scan efficiency* for SQLs inducing scanning]

- **lower the high water mark if possible!**
- **squeeze more rows into each block by modifying PCTFREE (and/or PCTUSED when relevant)**
- **reduce the row length, possibly by moving large, infrequently accessed columns to a separate table**
- **compress the data in the table**

High water mark (5) - lowering the...

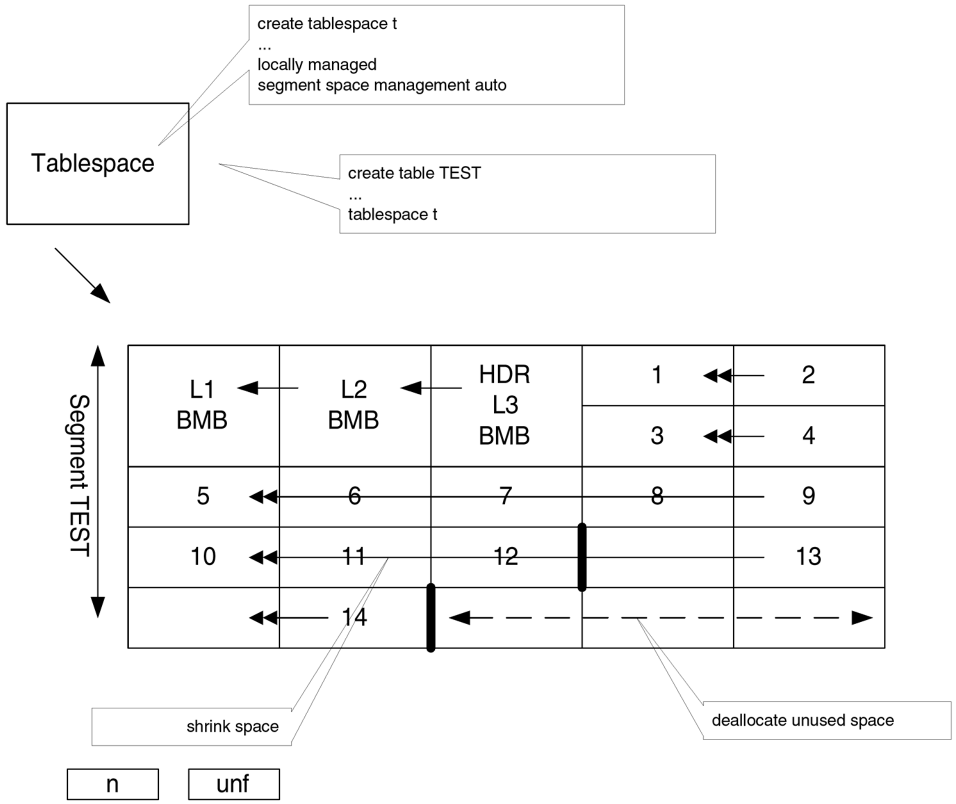
- WRONG **- deallocate space** [beyond the HWM!]
- **shrinking space - shrink**
  two-phased process:
  - compact space usage, ie.
    => move data if possible to the front of the extents
    => does not change the HWM as such
    [a mere data reorganisation moving rows, changing rowids]
  - shrink space usage, ie.
    => reset the HWM

    alter table enable row movement

    alter table shrink space compact -- if compact phase needed

    alter table shrink space
  - impact on indexes? => none
  - impact on dependant objects? => none

High water mark (5) - lowering the... (cont.)



High water mark (5) - lowering the... (cont.)

- **Is an object a candidate for shrinking?**
  - dbms_space.verify_shrink_candidate -- stored proc
  - dbms_space.verify_shrink_candidate_tbf -- pipelined function

```
SELECT *
FROM TABLE
    (dbms_space.verify_shrink_candidate_tbf
        ('owner',
         'object',
         'object type',
        verification bytes size));
```

0 - shrink size not possible
1 - shrink size possible

Allows for scripting/automation?!

High water mark (5) - lowering the... (cont.)

- **moving objects ...**

  Using 'alter table move' to move a table to a different tablespace, typically with the same properties!

  - logical dependencies are maintained (synonyms, grants, constraints, ...)
  - physical dependencies are invalidated (yet kept in the catalog with status *invalid* for ease of reference and recreating/revalidation)
  - characteristics of the object moved CAN NOT be changed!

  Moving to the same tablespace is possible;

High water mark (5) - lowering the... (cont.)

- **CTAS - create table as select + rename**
  - **advantage:**
    - flexibility of the select statement, including the often powerful order by clause!
    - all characteristics (logical/physical) of the object can be manipulated!
  - **disadvantage: logical and physical objects need to be recreated on the newly created object!**
- **Other techniques might include...**
  - **online reorg using the** dbms_redefinition **package**
  - **data pump (**expdp, impdp**)**

Sample case (i)(j)(k)

	(h)	(i)	(j)	(k)
Space use in object				
Unformatted Blocks	360	0	0	0
FS1 Blocks (0-25)	32	0	0	0
FS2 Blocks (25-50)	5085	4656	0	0
FS3 Blocks (50-75)	19	709	0	0
FS4 Blocks (75-100)...........	2905	3978	0	0
Full Blocks	11980	11038	15716	15389
Free Space in object				
Total Blocks..................	20736	20736	16384	15872
Total Bytes...............	42467328	42467328	3554432	32505856
Total MBytes.....................	40	40	32	31
Unused Blocks.....................	0	0	386	209
Unused Bytes......................	0	0	790528	428032
Last Used Ext FileId..............	6	6	9	9
Last Used Ext BlockId.........	32289	32289	15905	15393
Last Used Block.................	512	512	126	303
Indexes Valid?	*n/a*	*Y*	*N*	*N (removed)*
Dependencies Valid?	*n/a*	*Y*	*Y*	*N*

(i) <-> (h) - shrink

```
alter table enable row movement
alter table shrink space compact
alter table shrink space
```

(j) <-> (h) - move

```
alter table move
```

(k) <-> (h) - CTAS

```
create table as select order by on index column
```

Physical Data Organization

1. Locally managed tablespaces
2. Automatic space management
3. High water mark
4. Row Chaining - Row Migration
5. Indexes
6. About compression
7. About row-length
8. Conclusion

Row Chaining - Row Migration

***Row chaining*: a row is split up in row pieces, distributed over multiple blocks**

***Row migration*: a row is moved from one block to the other**

- **insert size > block size --> row chaining**
- **update size**

  > block free space

  < block size

  ==> row migration

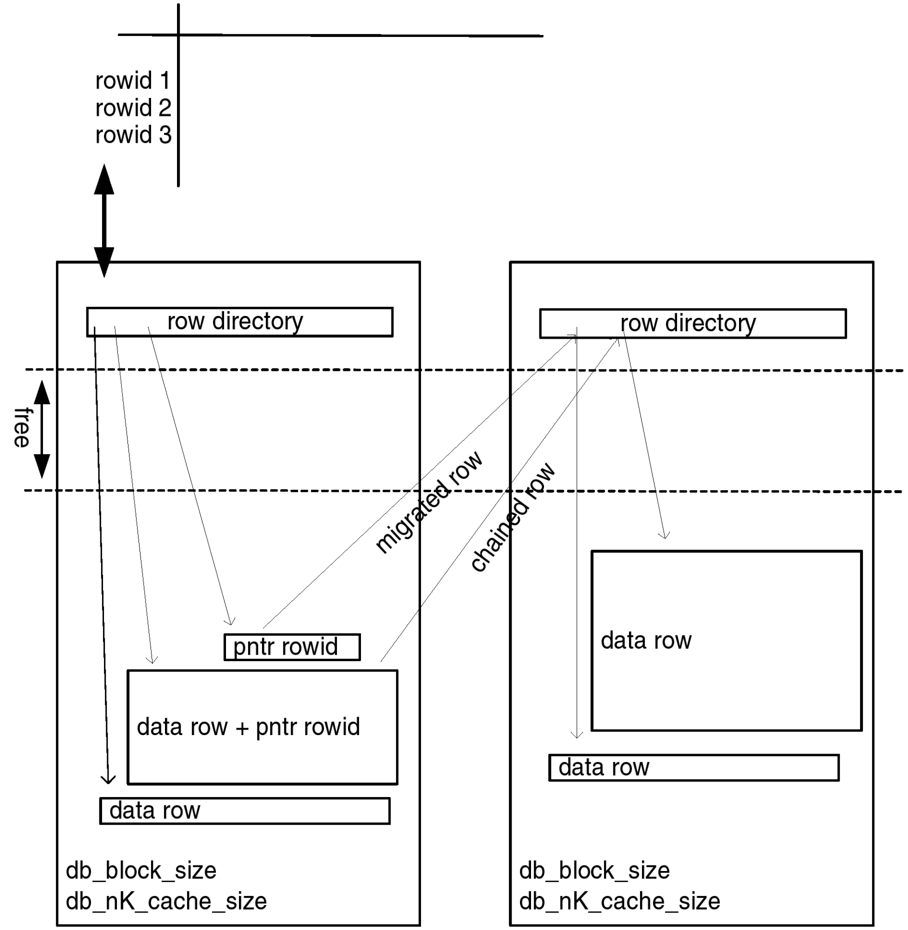
  ==> pointer
- **update size**

  > block free space

  > block size

  ==> row chaining

Row Chaining - Row Migration (2)



Physical Data Organization

Row Chaining - Row Migration (3)

How to evaluate?

- **SELECT ***
  FROM TABLE(
  dbms_space.object_space_usage_tbf
  ('*owner*',
  '*table*',
  '*object type*',
  NULL));
- **alternative (older) scenario:**
  - **create a table** chained_rows (utlchain.sql)
  - **run** analyze table list chained rows into chained_rows
  - **rows (with rowid) present in the table are ‘chained’ - count to get the number off...**

Sample case (g)(h)(i)(j)(k)

	(g)	(h)	(i)	(j)	(k)
# rows in table	133393	86065	86065	86065	86065
chained rows - fuction	0%	12%	11%	3%	3%
# chained rows - analyze	0	10068	9603	4947	4947

(g) initial situation

(h) situation after update/delete operation

(i) situation following a shrink operation, starting from (h)

(j) situation following a move operation, starting from (h)

(k) situation following a CTAS operation, starting from (h)

Row Chaining - Row Migration (4) - Think about ...

***Things* to** think **about...** [with an impact on chaining and migration]

- **numeric data types:**
  - **set a precision for NUMBER cols when numeric operations result in an unnecessarily high fractional precision - fractionals!**
  - **remember: binary_float, binary_double**
- **date data type: use timestamp to make** your **life easier**
- **character data type:**
  - **when char-size > 4000, use (C)LOBS**
  - **when char-size < 4000, use varchar2** *or char if possible*, **depending on:**
    - expected variation is column length size
    - expected degree of updatability
  - **never use LONGs, varchar**

Row Chaining - Row Migration (4) - Think about ... (cont.)

- **Nulls:**
  - **storage**
    - ‘not’ stored, implied if possible
    - if not implied -> ‘placeholder’
    - row growth?
  - **indexes**
    - never stored in traditional B*-tree indexes
      (exceptions exist - eg. nulls in concatenated indexes)
    - stored in bitmap indexes
    - dense indexes
    - what about your where-conditions?
      => not null with default?

Row Chaining - Row Migration (4) - Think about ... (cont.)

- **Column order:**
  - **generally of <u>minor</u> importance...**
  - **perhaps use following order:**
    - most accessed first
    - fixed first, variable last
    - null columns last
- **Blocksize!**
- **pctfree [and pctused]**
  - **if frequent table scans => use a low PCTFREE if row-lengthening updates are rare**
  - **if table is subject to row-lengthening updates, increase PCTFREE to avoid row migration**
  - **using a low PCTFREE + high concurrent transactional activity: increase INITRANS**

Consultation

PCTFREE --> LOW
PCTUSED --> HIGH

Space conservation
Little block fragmentation

Modification

Avg row size increases — Increase PCTFREE

- waste space?
- block fragmentation?
- decreased row concentration

- explicit nulls
- implicit nulls
- variable length columns

Avg row size decreases — Decrease PCTFREE

- conserve space ?
- block defragmentation ?
- increased row concentration

Additions

Few deletes: Set PCTUSED high to avoid fragmentation, and to increase row concentration.

Frequent deletes: PCTUSED can be set lower: frequent inserts will prevent fragmentation.

Deletions

higher PCTUSED causes:

- consider block gragmentation
- consider row concentration

Physical Data Organization

Validate/evaluate the structure of existing indexes, using the analyze index validate structure **command!**
[do not perform online]

=> index health statistics stored in two tables:

- index_stats [1]
- index_histogram

(1) columns include:
height - blocks
name - lf_rows
br_rows - br_blks
del_lf_rows - del_lf_rows_len
distinct_keys - most_repeated_key
btree_space - used_space
pct_used

Indexes (2)

Crucial statistics:

- **pct_used, the percentage of index space used**
- **% of used space that no longer exists in data tables:**

(del_lf_rows_len / used_space) * 100

< 20 % - 25 %

Indexes (3) - recreating...

Rebuild ...	**Coalesce ...**
quick move to another tablespace	cannot move index to another tablespace
higher costs - more disk space	lower costs - no additional index space required
new tree created 'from scratch'	leaf blocks are coalesced within same branch
storage parameters can be changed	frees up index leaf blocks
index status is not important	index has to be available (status = valid) for operation to be executed

Sample case (g)(h)(i)(j)(k)

	(g)	(h)	(i)	(i1)	(i2)
lf_rows	133394	98205	99058	86065	86065
lf_blocks	11340	11361	11361	4530	4616
br_rows	11339	11360	11360	4529	4615
br_blocks	126	128	128	39	127
del_lf_rows	0	12140	12993	0	0
del_lf_rows_len	0	1056180	1130391	0	0
used_space	11742593	8682893	8757104	7559293	8807072
pct_used	56	41	42	90	86
height	3	3	3	3	3

(g) initial situation

(h) situation after update/delete operation

(i) situation following a shrink operation, starting from (h)

(i1) situation after index rebuild, based on (i)
(i1) situation after index coalesce, based on (i)

Sample case (g)(h)(i)(j)(k) - cont

	(g)	(h)	(i)	(j) +rebuild	(k)
lf_rows	133394	98205	99058	86065	86065
lf_blocks	11340	11361	11361	4530	4530
br_rows	11339	11360	11360	4530	4529
br_blocks	126	128	128	39	39
del_lf_rows	0	12140	12993	0	0
del_lf_rows_len	0	1056180	1130391	0	0
used_space	11742593	8682893	8757104	7559352	7559734
pct_used	56	41	42	90	90
height	3	3	3	3	3

(g) initial situation

(h) situation after update/delete operation

(i) situation following a shrink operation, starting from (h)

(j)
°) access to index not allowed before rebuilding the index
°) rebuild using coalesce not possible if unusable

(k)
°) index needs to be recreated

Indexes - clustering factor

A clustering factor is a measurement of how sorted a table is with respect to an index key.

- **If clustering_factor approaches the number of blocks in the table, the rows are ordered.**
- **If clustering_factor approaches the number of rows in the table, the rows are randomly ordered.**

Important for (large) range scans:

impact on query optimization!

Sample case (g)(h)(i)(j)(k)

	Clustering factor
(g)	36813
(h)	24619
(i)	23723
(j)	30670
(k)	18318

(g) initial situation

(h) after modifying data

(i) after shrink

(j) after move

(k) after ctas WITH order by

Indexes - clustering factor - optimization

- **access-path chosen by Oracle depends <u>also</u> on the number of expected data reads**
- **for range queries, ‘logically ordered’ data makes use of indexes less expensive**
- **example:**
  select * from sss_test where bb between x and y

X	Y		case (g)	case (k)
2000	- 4000	(27% data)	IX	IX
2000	- 5000	(40% data)	IX	IX
2000	- 7000	(65% data)	IX	IX
2000	- 8000	(77% data)	scan	IX
2000	- 8500	(84% data)	scan	IX
2000	- 9000	(85% data)	scan	scan

About compression

- **prior to Oracle 11g, only available when a table was created, rebuilt, or using direct load operations**
- **as of Oracle 11g, the Advanced Compression option enables data to be compressed when manipulated by DML**
  - **11.1 - compression on an row-by-row basis - repeated data** within a row
  - **11.2 - compression on a column-by-column basis - repeated data** within a column

    'aggressiveness' of compression can be tailored to specific needs
  - **thinks about compression overhead, specifically during DML operations**
- **remember:**
  - **character data, 'no' numerical data**
  - **table scans (I/O driven table scans)**

Sample case (l)

```
                                  (h)          (l)
Space use in object
Unformatted Blocks ...............360..........48
FS1 Blocks (0-25) .................32..........31
FS2 Blocks (25-50) ..............5085..........23
FS3 Blocks (50-75) ................19..........20
FS4 Blocks (75-100)..............2905........8060
Full Blocks     .................11980.......10183
Free Space in object
Total Blocks.....................20736.......18688
Total Bytes...................42467328....38273024
Total MBytes.......................40..........36
Unused Blocks.......................0...........0
Unused Bytes........................0...........0
Last Used Ext FileId................6...........6
Last Used Ext BlockId...........32289.......30241
Last Used Block...................512.........512
```

(h) after loading the table and performing the updates/deletes

(l) after loading the table and performaing the updates/deletes on a table created with the compression option.

About row-length

When frequently scanning objects, ‘reducing’ the row length might offer considerable advantages:

- **moving rarely read columns to another tablespace**

  might require complex (trigger based?) maintenance/joining of data

- **making sure LOB data is stored separately from ‘structured’ data**

  - **LOB size < 4000 - storage in row OR out of row - your call**
  - **LOB size > 4000 - storage out of row**

  [freq of retrieval, (no)caching, chunks, use separate tablespaces, ...]

- **using IOTs, thinking carefully about which columns to move to the overflow area, and which columns should not**
  [pcttreshold, including]

Conclusion

- **Physical data organisation is important**
- **Physical data organisation has an impact on:**
  - **space usage**
  - **query optimisation**
- **Data can be reorganised - know the consequences (pro's and con's) of the different techniques available.**

Thank you!

ABIS Training & Consulting
Kris Van Thillo
kvanthillo@abis.be

Physical Data Organization

1. Locally managed tablespaces
2. Automatic space management
3. High water mark
4. Row Chaining - Row Migration
5. Indexes
6. About compression
7. About row-length
8. Conclusion