Database Systems II B: DBMS-Implementation

Chapter 7: Storage of Relations II: Sets of Bytestrings

Prof. Dr. Stefan Brass
Martin-Luther-Universität Halle-Wittenberg
Wintersemester 2019/20

http://www.informatik.uni-halle.de/~brass/dbi19/

Objectives

After completing this chapter, you should be able to:

- write a short paragraph explaining how blocks are allocated in Oracle (mention segments, extents).
- find storage information in the data dictionary.

  And use the `ANALYZE TABLE` command to populate the dictionary tables.
- explain how relations are stored in Oracle (row and block format, TIDs/ROWIDs, migrated rows).
- estimate the number of blocks needed for a table.
- set the basic storage parameters for relations in Oracle for good performance.

Inhalt

Row Manager/Heap Files (1)

- Table rows are stored physically in disk blocks. Normally, each block stores rows from one table only.

  Clusters in Oracle are an exception.

- The disk space manager assignes a sequence of disk blocks to every table (called a segment in Oracle).
- The row manager has to store a set of rows in this space.

  The lower level modules can treat a row as a bytestring without inner structure, i.e. they do not need to understand how columns are encoded in the rows. The row format is discussed in the next section.

Row Manager/Heap Files (2)

- The basic operations of the row manager are to
  - insert, update, and delete a row,
  - return all existing rows (in a loop),
  - manage pointers to rows.

    I.e. determine the address or some kind of ID of a row, and locate the row with a given address/ID.
- The simplest and most common file structure to store a table is the heap file. It stores rows in no particular order.

  Whereever space is available. After all, relations are sets.
  (Note that this heap has nothing to do with the heap of heapsort.)

Row Manager/Heap Files (3)

- In many database management systems, the heap file is the only way to store table data.

  Of course, they also have indexes, which are organized in a different way. But indexes contain only access information, not the primary copy of the table data.

- As an alternative to heap files, Oracle also has clusters and index-organized tables.

  In this case, the storage position depends on data within the rows. This improves the performance, but reduces the flexibility. Heap files remain the most common method.

ROWIDs/TIDs (1)

- ROWIDs (row identifiers) are physical pointers to rows. They are called also TID (tuple identifier).
- Indexes provide a fast way to look up the ROWIDs of those table rows that contain a given value in a certain column.

  An index over column `A` of a table `R` can be understood as an auxillary table `I(A,ROWID)`. The first column contains all data values that currently appear in `R.A`, the second column contains the ROWIDs of the matching rows in `R`. The index is not organized as a heap file, but e.g. as a B-tree, which gives fast access to the entry for a specific value (see below). One could organize the original table as a B-tree, but then only one attribute could be indexed (since B-trees basically store the entries sorted by `A`).

ROWIDs/TIDs (2)

- So the Row Manager must support two ways to access a row:
  - Read all rows of the table in a full table scan.
  - Get a particular row given its address (ROWID).
- The access via the ROWID should be especially fast, i.e. normally only a single block access.
- Therefore, ROWIDs usually contain the physical address of the block in which the row is stored.

  I.e. the file number and the block number within the file. Plus e.g. the number of the row within the block.

ROWIDs/TIDs (3)

- Most DBMS guarantee that ROWIDs/TIDs do not change for the entire lifetime of a tuple.

  Except when the tuple is exported and imported again. That would basically create a new row with the same values.

- The reason that ROWIDs should be kept stable is
  - there can be many indexes for the same table. If the ROWID of a tuple should change, all would have to be updated.
  - some DBMS (e.g. Oracle) make ROWIDs available on the user level.

ROWIDs/TIDs (4)

- Expert users can use ROWIDs in Oracle to improve performance.

  E.g. foreign keys could be supported by an additional column that contains the ROWID of the referenced tuple (the real foreign key is then needed only for export/import). One could also construct one's own tree structures (with restrictions). If the user can store ROWIDs, it might be difficult for the system to determine all pointers to a given row. Then stable ROWIDs are especially important.

- If ROWIDs must remain stable, and ROWIDs must contain a physical block address, tuples are basically locked to the block recorded in their ROWID.

  Design decision for DBMS vendor: Support stable ROWIDs?
  Support the one-block-access to rows by ROWID?

ROWIDs/TIDs (5)

- ROWIDs are similar to object identifiers (OIDs):
  - Even if two rows agree in all attributes, they can be distinguished by their ROWIDs.

    It is bad design to permit duplicate rows. At least, one must really know what one is doing.
  - The ROWID remains stable even if primary key attributes are updated.

    Normally, there should be no updates on primary key attributes.
- However, if a tuple is deleted, a newly created tuple might get its ROWID (this differs from real OIDs).

Oracle ROWIDs (1)

- In Oracle, every table has a "pseudocolumn" ROWID, which can be queried like a real column:

```
SELECT ROWID, FIRST, LAST
FROM   STUDENTS
```

- The column is not listed with `describe` or `SELECT *`.
- It is not possible to update the column ROWID.

  It is not stored, but computed from the storage position of the row.

- The pseudocolumn can also be used in conditions:

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  ROWID = 'AAACiMAACAAAAYnAAA';
```

Oracle ROWIDs (2)

- An Oracle8 ROWID consists of:
  - `SUBSTR(ROWID,1,6)`: Data object number.

    This identifies the segment. I do not see why it is necessary. Old Oracle 7 ROWIDs did not contain this part. The data object number is e.g. shown in `USER_OBJECTS`.
  - `SUBSTR(ROWID,7,3)`: Relative file number.
  - `SUBSTR(ROWID,10,6)`: Block number in the file.
  - `SUBSTR(ROWID,16,3)`: Row number in the block.
- A base 64 encoding is used for the numbers.

  Six bits per character (0–63) are coded using the characters `A-Z`, `a-z`, `0-9`, `+` and `/`. E.g. `AAC` is the number 2.

Oracle ROWIDs (3)

- There is a package of stored functions for decoding the components of a ROWID:

```
SELECT DBMS_ROWID.ROWID_OBJECT(ROWID),
       DBMS_ROWID.ROWID_RELATIVE_FNO(ROWID),
       DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID),
       DBMS_ROWID.ROWID_ROW_NUMBER(ROWID),
       FIRST, LAST
FROM   STUDENT
```

- Rows in a block are numbered 0, 1, 2, ...

  Holes in the sequence are numbers of deleted rows.

- By querying and decoding the `ROWID`, it is possible to find out where a particular row is stored.

Fixed-Length Rows (1)

- In old, simple DBMS, rows had to be of fixed length (i.e. all rows in a table had the same storage size).

  Like e.g. a record in C. In newer systems, this might be an option for certain tables (not in Oracle).

- This simplifies the task of the row manager: It stores as many rows in one block as the space permits.

  So if the row size is 100 bytes, the first row would begin e.g. at offset 0 from the beginning of the block, the second at offset 100, the third at offset 200, etc. (like an array in C).

Fixed-Length Rows (2)

- Normally, one would not split a row between two blocks, but rather leave some space unused.

  Unless the row is very long, it should be possible to retrieve it with one block access. E.g. block size 2048: 48 Byte wasted.

- In order to manage the space within a block, a flag "deleted" (or "free") is needed for every slot that can contain a row.

- In addition, e.g. a linked list of blocks with empty space is needed to find a free slot when a new row is inserted.

Fixed-Length Rows (3)

- There must also be a mechanism to find all blocks that might contain rows in them (for a full table scan).
- With fixed-length rows, stable addresses mean that we cannot move a row after it has been created.

  E.g. even if after some deletions only one row remains in a block, we are not allowed to move it to another block with free space, since this would change its ROWID (and a full table scan runs the faster the less blocks are needed).

Variable-Length Rows (1)

- Often rows in a table have a variable size.

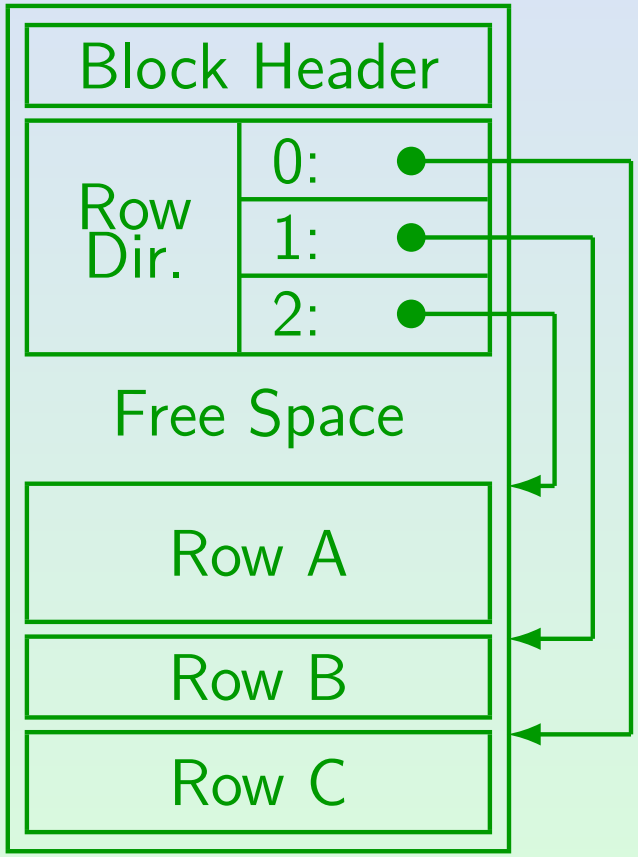
  E.g. because of `VARCHAR` columns.

- Then rows can also grow or shrink via updates.
- Oracle treats all rows as variable-length.

  Since columns can be added to a table with `ALTER TABLE`, one must either copy the entire table at this point, or abandon the idea of fixed-length rows. Also when null values should be stored with less space than the normal column value, the row length becomes variable.

- Variable-length rows are usually managed in a block with a row directory, i.e. a small table giving the offsets (start addresses) of the rows in the block.

Variable-Length Rows (2)



Variable-Length Rows (3)

- The ROWID consists of file number, block number, and the index in the row directory.
- The indirect addressing via the row directory makes it possible that rows are moved within the block:
  - E.g. Row B is updated and grows slightly.

    Then Row A has to be moved towards the beginning of the block (where there is still free space) to make room.
  - Or suppose that Row B is deleted.

    Then Row A would be moved towards the end of the block, such that the free space is not fragmented. However, most systems including Oracle merge free space only if necessary to insert a new row.

Variable-Length Rows (4)

- The block header may e.g. contain
  - Block address, type of segment, table name.
  - The size of the row directory, size of free space.
  - Next block in the list of blocks with free space.
  - A serial version number for this block which is incremented for every update.

    This is needed for crash recovery.
  - A bit pattern to detect partially written blocks.

    The pattern at the begin and end of the block must agree, they are both inversed on every write.

Variable-Length Rows (5)

- Block overhead in Oracle: ca. 84–107 Byte.
  (Gurry/Corrigan use 90 Byte in computations.)
- In Oracle, the row directory needs two bytes per entry.

  Oracle never releases elements of the row directory. If at some point in time, 50 rows were stored in the block, the row directory will always need 100 bytes, even if it contains only a single row. Of course, if the row is stored in location 50, there is would be in any case no way to shorten the row directory, because the ROWID must be kept stable.

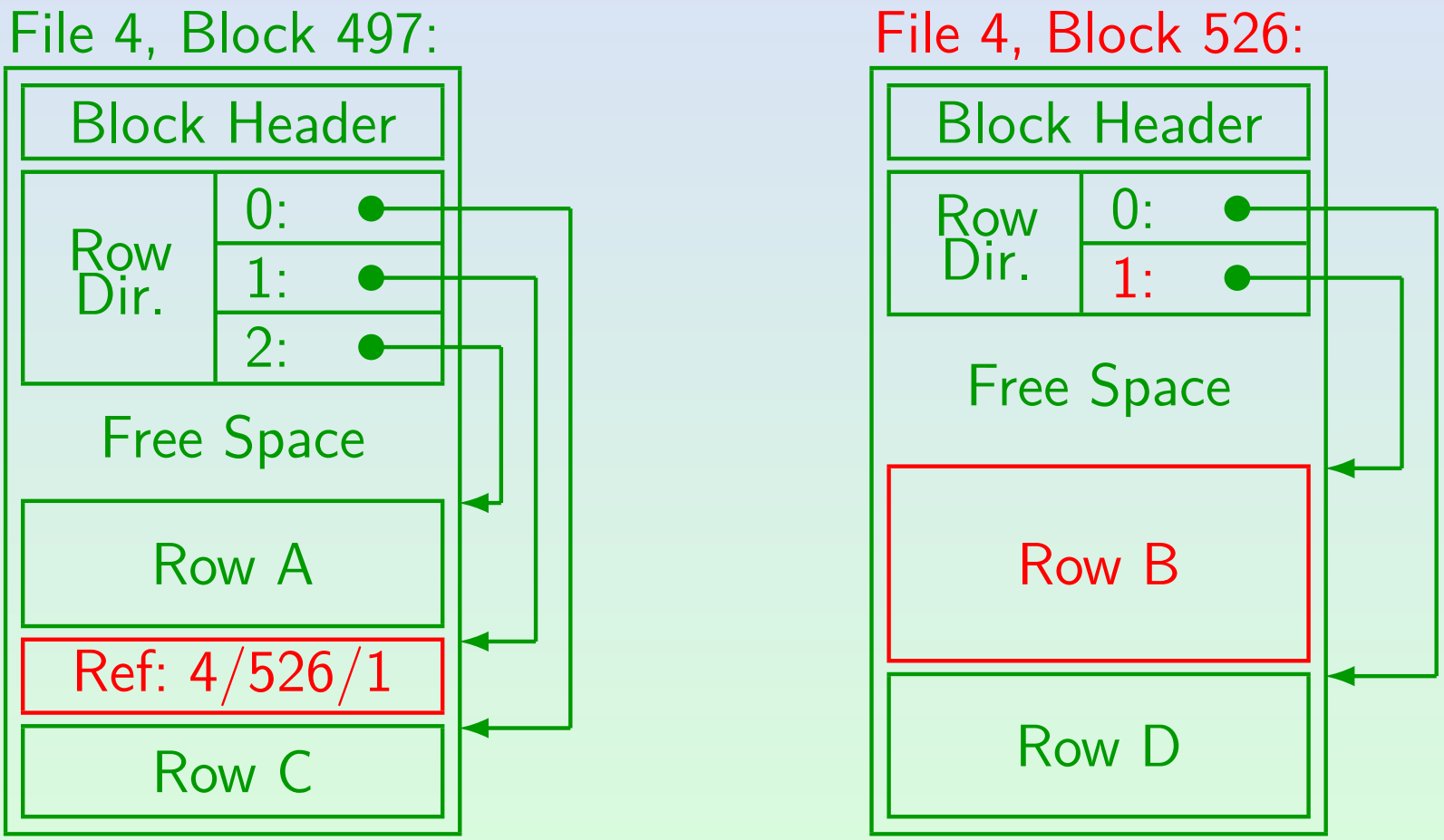
Variable-Length Rows (6)

- If a row grows and there is not enough free space left in the block, it must be moved ("migrated") to another block.
- A pointer must be left behind in this block so that the row can still be found via its ROWID.

  Thus, e.g. its entry in the row directory is still used.
- So now two block accesses are needed in order to retrieve this row, given its ROWID.

  This decreases performance, especially since the row might be stored far away on the disk.

Variable-Length Rows (7)



Variable-Length Rows (8)

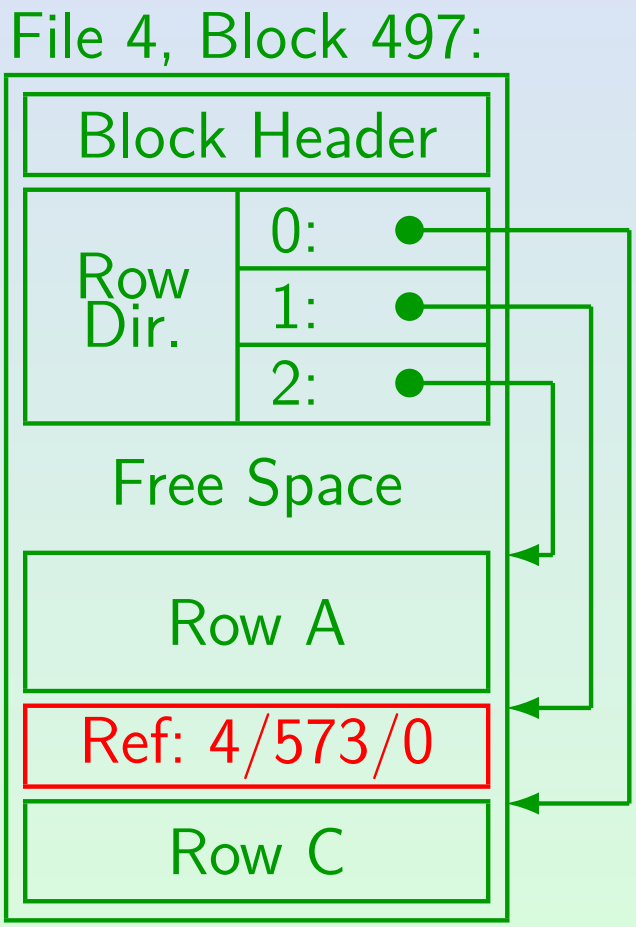
- If Row B should have to move again, the original reference in block 497 is updated. In this way, two block accesses remains the maximum to retrieve a row with given ROWID.

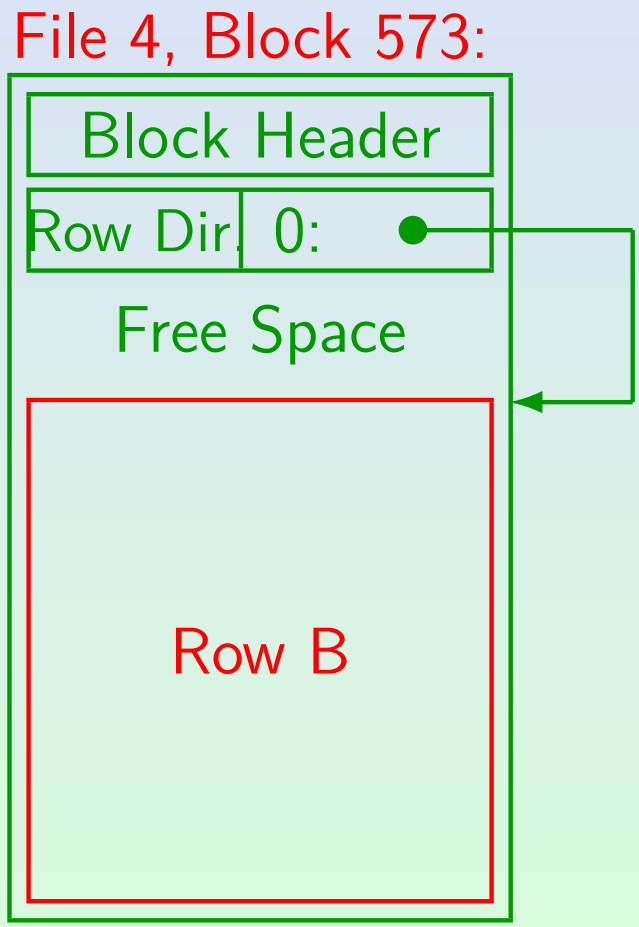
  The new address is stored only in the reference under the old (and only) ROWID.

- When a new row is stored, storage must be reserved that is at least large enough to contain a reference to a new place.

  Each row will at least need e.g. 14 Bytes.

Variable-Length Rows (9)





Variable-Length Rows (10)

- Oracle can also store rows in multiple pieces in different blocks (“chained rows”).
- This is only done for rows longer than a block. If the row fits in a block, it is completely moved to another block.
- If there are many chained rows, consider increasing the `DB_BLOCK_SIZE` (requires recreation of the DB).

  Depending on the version, the default size might be 2KB. The block size should be a multiple of the OS block size (often 4KB or 8KB). The parameter can only be set when the database is created. A block size which is too large can decrease the performance for accesses to single rows (e.g. via an index) and also the caching performance.

Summary: Row Manager (1)

Main Operations of the Row Manager for Heap Files:

- Operations for full table scans:
  - Open a scan ("cursor") over a given table.
  - Are there further rows? ("end of scan")
  - Get next row for a given scan.

    Implementation detail: Row is not copied. Instead the containing block is pinned in the buffer, pointer is returned.
  - Determine the ROWID of the current row.
  - Close a scan.
- Get a row given its ROWID.
- Insert/Update/Delete a row.

Summary: Row Manager (2)

The main tasks of the row manager are:

- Free space management

  In which block should a new row be stored?
  What happens if a row grows or shrinks?

- Used space management

  Which blocks actually contain rows and must be read during a full table scan?

- Management of stable addresses for rows.

  Of course, the access via ROWIDs should be efficient (usually one block access, sometimes two).

Summary: Row Manager (3)

- Problem: Rows have in general variable length and can grow and shrink.
- The row manager determines the block format.

  A small part of the block might already be used by the disk manager to implement segments.
- The heap file is very common, but there are alternatives. These might support associative access to the rows.

  I.e. return the row with a given attribute value. E.g., in the Transbase DBMS, all relations are stored as B-trees.

Inhalt

PCTFREE (1)

- To avoid migrated rows, some free space in each block should be reserved for growth of the rows.

  Then `INSERT` commands will not use up all space, only subsequent `UPDATE` commands can fill a block entirely.

- Oracle has a parameter `PCTFREE` in the `CREATE TABLE` which determines this space reserve (in percent of the block size).

  E.g. if `PCTFREE` is `20`, and the block size is 2KB (2048 Byte), the space reserve is $(20/100) * 2048 = 410$ bytes. This space must remain free after the `INSERT`. If the row to be inserted is 50 bytes long, it will be inserted only in a block with at least 460 bytes of free space (two additional bytes might be needed for the row directory entry).

PCTFREE (2)

- One must estimate how much the row length will grow over the row's lifetime

  This is part of physical DB design. Typical case for growing rows: Some attributes are null when the row is inserted, and later filled out.

- If `PCTFREE` is too small, there will be migrated rows.
- If `PCTFREE` too large, space is wasted and full table scans will run longer.
- If there are many migrated rows: Export all rows, empty or recreate the table, import the rows again.

  And of course `PCTFREE` should be changed. This can be done with `ALTER TABLE`. It effects all future insertions.

PCTFREE (3)

- If rows are only inserted and deleted, but not updated (or at least to not become longer by updates), $\texttt{PCTFREE} = 0$ can be chosen.
- $\texttt{PCTFREE} = 10$ is a common value (default value).
- $\texttt{PCTFREE} = 20$ would be chosen if it is known that rows quite significantly grow because of updates.
- In general, the following formula can be used:

$$\frac{\text{Rowsize after Update} - \text{Rowsize at Insertion}}{\text{Rowsize after Update}} * 100$$

This value can be too large: Simplified calculation+problem on Slide 36.

PCTFREE (4)

- Suppose that rows are inserted at 40 Bytes length, but they all will become 60 Bytes due to updates.
- Then a block of 2048 bytes can contain rows. $(2048 - 90)/(60 + 2) = 31$

  90 Bytes are the overhead for the block header, 2 Bytes the overhead for the entry in the row directory.
- Thus, $31 * (60 - 40) = 620$ Bytes should remain free at insertion, i.e. $\texttt{PCTFREE} = 620/2048 = 30\%$.

PCTFREE (5)

- The above calculation assumes that a block is filled with short rows before the first row grows.

  This would hold e.g. when there are only insertions into a table (no deletions), and when the time difference between the insertion and the update is longer than the time needed to fill a block.

- If this is not the case, `PCTFREE` can be chosen (much) smaller.

  Basically, the `PCTFREE` model does not treat this case. No formula can be applied, only rules of thumb. Advanced exercise: Propose other ways to control the space reserve.

PCTUSED (1)

- For each table/segment, Oracle manages a linked list of blocks that still have space for new rows.

  Oracle can manage more than one such list (parameter `FREELISTS`) for tables with many concurrent insertions.

- The Parameter `PCTUSED` determines which blocks are kept on this free list.

- When Oracle wants to insert a new row, it looks at the first block on the free list. If after the insertion, there would be still `PCTFREE` free space left, the insertion is done.

PCTUSED (2)

- Otherwise (insertion attempt fails), Oracle removes the block from the free list, unless is is filled to less than PCTUSED percent.
- This exception ensures that exceptionally long rows do not remove blocks with a reasonable amount of free space from the free list.
- Blocks are removed from the free list only if an insertion attempt fails. Only then PCTUSED becomes important (for insertions).

  PCTUSED is also important for deletions, see below.

PCTUSED (3)

Exercise:

- Suppose the block size is 1000 (to simplify the calculation). Let `PCTFREE=20` and `PCTUSED=60`.
- The free list looks as follows:

- What happens if the following rows are inserted?
  - Row A: 200 Byte,
  - Row B: 500 Byte,
  - Row C: 200 Byte.

PCTUSED (4)

- The sum of PCTFREE and PCTUSED can be no more than 100, but it should be less in order to allow blocks to be removed from the free list.
- If the sum is 100, blocks are in effect not removed from the free list (unless they are filled exactly to the right byte).
- Then INSERTs will take a long time since they have to scan a large number of blocks for free space.

PCTUSED (5)

- Suppose that all rows are 200 Bytes long, and let PCTFREE: 10, blocksize: 2048, header size: 90 Bytes.
- Blocks have $2048 - (90 + 205) = 1753$ Bytes available space, rows need $200 + 2$ byte, so after 8 rows are inserted, the next insertion fails.
- Only $(8 * 202) + 90 = 1706$ bytes are actually used, so PCTUSED must be less than $1706/2048 = 83\%$ in order to remove the block from the free list.

  Choosing PCTFREE smaller has no effect for insertions, still 8 rows are inserted until PCTUSED is considered.

PCTUSED (6)

- The maximum value for `PCTUSED` can be computed as follows (it leaves space for one average row, so that when such an insertion fails, the block is removed from the free list):

$$\frac{\text{Available Space} - \text{Length of one Row}}{\text{Blocksize}} * 100$$

where the available space is
$(\text{Blocksize} * (100 - \texttt{PCTFREE})/100) - \text{Header Size}$.

- The default value is `PCTUSED=40`.

PCTUSED (7)

- If rows are deleted from a block, the block is put on the free list once less than PCTUSED space is used.

  Since there is some overhead involved in putting blocks on the free list and removing them again, it makes sense that there should be space for several rows before a block is put back on the free list.

- The smaller PCTUSED is chosen, the longer it takes until the block is again considered having free space after deletions.

- E.g. if in the example PCTUSED were 50%, less than $2048 * 0.50 = 1024$ Bytes must be used (4 rows) before the block is put back on the free list.

Auto Segment Sp. Mgmt. (1)

- Since Oracle9i, there is an alternative to the use of freelists: Automatic segment space management (ASSM).
- It uses bitmaps that distinguish between
  - blocks that are full (no candidate for insertion),
  - blocks with 0-25% of free space (FS1),
  - blocks with 25-50% of free space (FS2),
  - blocks with 50-75% of free space (FS3),
  - blocks with 75-100% of free space (FS4),
  - blocks that are not yet formatted.

Auto Segment Sp. Mgmt. (2)

- Probably "unformatted blocks" are blocks above the high water mark and "full blocks" correspond to blocks into which an insertion has failed.
- One can query the number of blocks of each class with the procedure DBMS_SPACE.DBMS_SPACE_USAGE.

```
dbms_space.space_usage(
segment_owner      IN  VARCHAR2,
segment_name       IN  VARCHAR2,
segment_type       IN  VARCHAR2,
unformatted_blocks OUT NUMBER,
unformatted_bytes  OUT NUMBER,
fs1_blocks         OUT NUMBER,   fs1_bytes      OUT NUMBER,
fs2_blocks         OUT NUMBER,   fs2_bytes      OUT NUMBER,
fs3_blocks         OUT NUMBER,   fs3_bytes      OUT NUMBER,
fs4_blocks         OUT NUMBER,   fs4_bytes      OUT NUMBER,
full_blocks        OUT NUMBER,   full_bytes     OUT NUMBER,
partition_name     IN  VARCHAR2  DEFAULT NULL);
```

Auto Segment Sp. Mgmt. (3)

- Example for calling this procedure:
  - First declare SQL*Plus variables, e.g.

```
variable unformatted_blocks number;
```

  - Use a PL/SQL block to call the procedure:

```
begin
    DBMS_SPACE.SPACE_USAGE('USER1', 'TAB1',
        'TABLE', :unformatted_blocks, ...);
end;
/
```

  - Then print the variable values:

```
print unformatted_blocks;
```

Auto Segment Sp. Mgmt. (4)

- The advantage of automatic segment space management is that the DBA does not have to think about `PCTUSED`, `FREELISTS`, and `FREELIST GROUPS`:
  These parameters become obsolete.
- Automatic segment space management can better adapt to varying workloads.

  With manual segment space management, one must increase the parameter `FREELISTS` if there are many parallel insertions (otherwise there is contention on the single freelist). If one has a DB with many server machines (in an Oracle Real Application Clusters environment), the `FREELIST GROUPS` determine how machines are mapped to freelists.
  With automatic segment space management, these situations are handled automatically.

Auto Segment Sp. Mgmt. (5)

- The kind of segment space management is selected on the level of the tablespace (and cannot be changed after tablespace creation):

```
CREATE TABLESPACE TBSP1
DATAFILE '/data/tbsp1.dbf' size 100M
EXTENT MANAGEMENT LOCAL AUTOALLOCATE
SEGMENT SPACE MANAGEMENT AUTO;
```

- Alternative (use of freelists):

```
SEGMENT SPACE MANAGEMENT MANUAL;
```

- Automatic segment space mgmt. is now the default.

  It works only in tablespaces with local extent management.

CREATE TABLE Syntax

Example (clauses discussed in this course):

```
CREATE TABLE STUDENT(
             SID   NUMERIC(4)  PRIMARY KEY,
             FIRST VARCHAR(20),
             LAST  VARCHAR(20) NOT NULL)
TABLESPACE USER_DATA
STORAGE(INITIAL 10K
        NEXT 10K -- Not with locally managed TBSP
        PCTINCREASE 50 -- This, too
        BUFFER_POOL KEEP)
PCTFREE 20
PCTUSED 60 -- Not with automatic segm. sp. mgmt.
CACHE;
```

Full Table Scan (1)

- As explained above, the row manager module must be able to find all blocks that contain rows (or might contain rows) of a given table.

  The disk manager below returns a list of blocks for each table (a segment), but not all blocks do necessarily contain rows.

- Oracle manages a "high water mark" for each table, that is the number of blocks that were ever used for storing rows of this table.

- In a full table scan, Oracle will read all blocks until this "high water mark".

Full Table Scan (2)

- Suppose that a table contains 100000 rows, stored in 1000 blocks. Even if then all rows are deleted, a full table scan will nevertheless read all 1000 blocks.
- Normally, such extreme situations do not happen.
- But if there should be a large number of deletes, consider exporting and reimporting the table.

  Unless a similar number of insertions is expected soon.
- To delete all rows from a table use the `TRUNCATE` command. This resets the high water mark.

  No `ROLLBACK` is possible for this command.

ANALYZE TABLE (1)

- The following Oracle SQL command gathers statistical information about a table, e.g. `EMP`:

```
ANALYZE TABLE EMP COMPUTE STATISTICS
```

- This command stores size information about the table in the data dictionary, e.g.
  - The number of rows in the analyzed table.
  - The average row length in bytes.
  - The number of blocks that ever contained rows.
  - How full these blocks are on average.
  - How many different values each attribute has.

ANALYZE TABLE (2)

- This information is used by the query optimizer in order to estimate the cost (execution time) for each alternative query evaluation plan.
- Oracle does not automatically keep this information up-to-date.
- If the DBMS wanted to keep the number of rows of a table (table size) current, any insertion on table R would lock the data dictionary entry for R.

  Then no parallel insertions would be possible, e.g. different users could not enter orders concurrently.

ANALYZE TABLE (3)

- The query optimizer does not need exact values for size parameters.
- In the worst case it chooses a query evaluation plan that takes longer than the optimal one.
- Therefore, it is no problem that the data about the table size are slightly outdated.
- One should execute the `ANALYZE TABLE` again from time to time, at least after significant changes in the size of the table.

ANALYZE TABLE (4)

- The `ANALYZE TABLE` command can take a long time to execute for large tables.

  E.g. in order to compute the number of different values in each attribute, it must sort the set of attribute values.

- Therefore, one can also request to estimate statistics from a sample of rows

  ```
  ANALYZE TABLE EMP ESTIMATE STATISTICS
  ```

  One can also add e.g. "`SAMPLE 10 PERCENT`".

  The DBA should execute the `ANALYZE TABLE` outside of the main business hours.

ANALYZE TABLE (5)

- The output of the `ANALYZE TABLE` command is stored in the data dictionary tables, especially `TABS`, `COLS`, `USER_TAB_COL_STATISTICS`.

  The entries in `COLS` remain only for backward compatibility, Oracle suggests to use now `USER_TAB_COL_STATISTICS` (`USER_PART_COL_STATISTICS` for partitioned tables). More information about the data distribution in a column can be collected with histograms (explained in the chapter about query optimization).

- The command itself prints only "Table analyzed.".
- All data dictionary columns that contain output from the `ANALYZE TABLE` are null until the table is analyzed for the first time.

Data Dictionary: TABS (1)

- TABS is a synonym for USER_TABLES. It contains one row for each table owned by the current user (not including views). It has 44 columns, e.g.:
  - TABLE_NAME: Name of the table.
  - TABLESPACE_NAME: Tablespace in which the table is stored.
  - PCT_FREE, PCT_USED, INITIAL_EXTENT, NEXT_EXTENT, MIN_EXTENTS, MAX_EXTENTS, PCT_INCREASE, FREELISTS: Storage parameters set in the CREATE TABLE.

    PCTFREE etc. are reserved words. Therefore the different spelling.

Data Dictionary: TABS (2)

- Columns of TABS, continued:
  - NUM_ROWS: Number of rows in the table.
  - BLOCKS: The number of used data blocks.

    This is the "high water mark" mentioned above (i.e. blocks that ever contained rows), not the total number of blocks allocated for the table.
  - EMPTY_BLOCKS: Number data blocks that are allocated for the table, but not yet used.

    Since every segment needs one header block, the total number of allocated blocks (segment size) is BLOCKS+EMPTY_BLOCKS+1.
  - CHAIN_CNT: Number of rows which are split between blocks (includes migrated rows).

Data Dictionary: TABS (3)

- Columns of TABS, continued:
  - AVG_ROW_LEN: Average length of a row in bytes.
  - AVG_SPACE: Average amount of free space (in bytes) in blocks below the high water mark.
  - AVG_SPACE_FREELIST_BLOCKS: Average free space in blocks on the free list (used for insertions).
  - NUM_FREELIST_BLOCKS: Number of blocks on the free list (the free list contains only blocks below the high water mark).
  - LAST_ANALYZED: Date of last ANALYZE TABLE.

References

- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Edition. Section 5.5,5.7.
- Ramakrishnan/Gehrke: Database Management Systems, 2nd Edition. Section 7.3, 7.5–7.8.
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Ed., Chap 10.
- Kemper/Eickler: Datenbanksysteme (in German), Chap. 7, Oldenbourg, 1997.
- Garcia-Molina/Ullman/Widom: Database System Implementation. Chapter 3.
- Härder/Rahm: Datenbanksysteme, Konzepte und Techniken der Implementierung (in German). Chapter 2, 5.
- Jason S. Couchman: Oracle8i Certified Professional: DBA Certification Exam Guide with CDROM. Osborne/ORACLE Press, ISBN 0-07-213060-1, ca. 1257 pages.
- Mark Gurry, Peter Corrigan: Oracle Performance Tuning, 2nd Edition (with disk).
- Gray/Reuter: Transaction Processing: Concepts and Techniques. 1993.
- Oracle 8i Concepts, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76965-01.
- Oracle 8i Designing and Tuning for Performance, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76992-01.