Part 9: Physical Storage of Relations

References:

- Ramez Elmasri, Shamkant B. Navathe: Fundamentals of Database Systems, 3rd Edition. Section 5.5,5.7.
- Raghu Ramakrishnan, Johannes 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.
- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: Database System Implementation. Chapter 3.
- Theo Härder, Erhard Rahm: Datenbanksysteme, Konzepte und Techniken der Implementierung (in German).
- Michael J. Corey, Michael Abbey, Daniel J. Dechichio, Ian Abramson: Oracle8 Tuning.
- Jason S. Couchman: Oracle8i Certified Professional: DBA Certification Exam Guide with CDROM. Osborne/ORACLE Press, ISBN 0-07-213060-1, ca. 1257 pages, ca. $99.99.
- Mark Gurry, Peter Corrigan: Oracle Performance Tuning, 2nd Edition (with disk).
- Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques.
- 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.

Objectives

After completing this chapter, you should be able to:

- write a short paragraph explaining how blocks are allocated in Oracle (mentioning segments, extents).
- find physical storage information in the data dictionary.
- use the `ANALYZE TABLE` command in Oracle.
- explain how relations are stored in Oracle (row format and block format).
- explain how TIDs/ROWIDs are kept stable (migrated rows).
- estimate the number of blocks needed for a given table.
- set the basic storage parameters for relations in Oracle for good performance.

Overview

1. Disk Space Management: Segments, Extents
2. Block Format, TIDs/ROWIDs
3. Block Free Space Management in Oracle
4. Row Format
5. Data Format

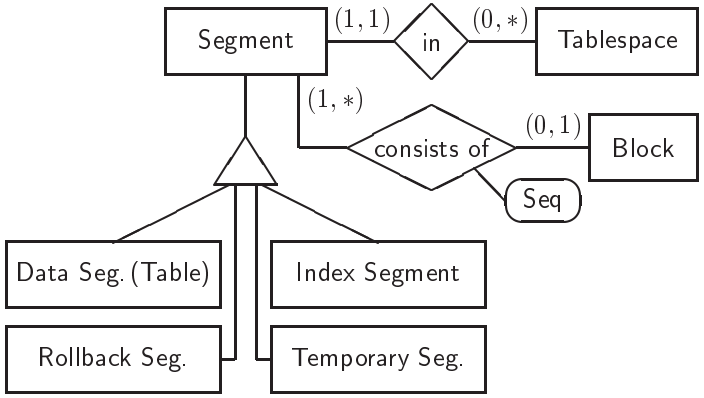
Segments (1)

- If tablespaces are the "logical disks" of Oracle, segments are the "logical files".
- Segments are sequences of data blocks within a tablespace.
  The sequence does not have to be the physical sequence.
  The blocks are not necessarily stored in contiguous places.
- Segments can grow (further blocks can be appended at the end) and shrink (blocks are removed at the end).
  In Oracle, segments shrink only when explicitly requested.
- The used storage in a tablespace is partioned into segments.
  Every data block can belong to at most one segment.
- A tablespace can contain many segments.

Segments (2)

- For every table, index etc. Oracle creates a "segment" inside the tablespace that is mentioned in the `CREATE TABLE` etc.
  The four basic kinds of segments are: Data segments (for tables), index segments, rollback segments (for storing old versions of blocks), temporary segments (for sorting during query evaluation).
- Normally, the relationship between data segments and tables is 1:1. But in general, it can be n:m:
  - Partioned tables (see Chapter 17) have more than one segment (usually in different tablespaces on different disks).
  - Clusters (see Chapter 17) can contain rows from several tables having one or more attributes in common.

Segments (3)



Segments (4)

- The data dictionary view `DBA_SEGMENTS` contains one row for each segment. It has the following columns:
  - `OWNER`: User who created the table etc.
  - `SEGMENT_NAME`: Table name, index name, etc.
  - `PARTITION_NAME`: For partitioned tables (normally null).
  - `SEGMENT_TYPE`: Type of the segment, e.g. `TABLE`.
    Possible values: `TABLE`, `INDEX`, `CLUSTER`, `TABLE PARTITION`, `INDEX PARTITION`, `ROLLBACK`, `DEFERRED ROLLBACK`, `TEMPORARY`, `CACHE`, `LOBINDEX`, `LOBSEGMENT`.
  - `TABLESPACE_NAME`: TS in which the segment is stored.
  - `HEADER_FILE`, `HEADER_BLOCK`: Storage position of segment header block (first block, contains control info).

Segments (5)

- Columns of `DBA_SEGMENTS`, continued:
  - `BYTES`, `BLOCKS`: Current size of the segment.
  - `EXTENTS`, `INITIAL_EXTENT`, `NEXT_EXTENT`, `MIN_EXTENTS`, `MAX_EXTENTS`, `PCT_INCREASE`: See below.
  - `FREELISTS`, `FREELIST_GROUPS`: For management of blocks with free space within the segment, usually both are 1.
  - `RELATIVE_FNO`: Segment header block (for Parallel Server).
  - `BUFFER_POOL`: Buffer pool for caching blocks from this segment.
- There is also `USER_SEGMENTS` listing the segments owned by the current user (some columns are missing).

Extents (1)

- Oracle allocates storage for segments in units called "extents". An extent is sequence of contiguous disk blocks.
  Thus, an extent can be especially fast read from the disk.
  Of course, each extent is contained in only one data file.
- An extent belongs to a single segment and thus to a single table (or index etc.). A segment can consist of many extents.
- Extent sizes are specified in the table declaration:

```
CREATE TABLE STUDENTS(SSN CHAR(9), ...)
             TABLESPACE USER_DATA
             STORAGE(INITIAL 200K
                     NEXT 50K
                     PCTINCREASE 100)
```


Extents (2)

- When the table is created, the initial extend is allocated.
  Although it does not yet contain any rows, it needs disk space for the initial extent (200 KB in the example).
- Whenever the disk space allocated for a table is full, another extent will be allocated.
- However, it is not good if a table is stored in many extents.
  Different extents can be stored on very different locations on the disk. If a table consists of many extents, the disk arm will have to move a lot during a full table scan.
  Also, the list of extents should fit into one block.
- Ideally, each table should be stored in a single extent.
  Thus one must plan how much space will be needed.
  More than about 100–500 extents are certainly bad.

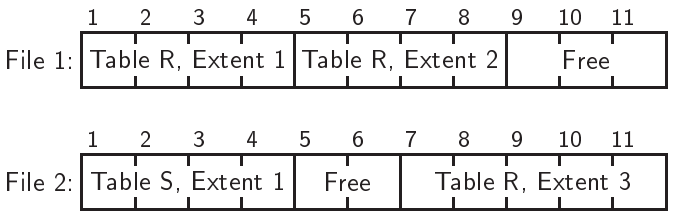
Extents (3)

- In the example, the second extent will be 50 KByte (`NEXT`).
- Normally, all following extents have the `NEXT` size. However, one can request that each following extent will be larger than the previous one (in order to reduce the number of extents).
- `PCTINCREASE 100` means that the extent size is doubled. The third extent will be 100 KB, the fourth 200 KB, etc.
  If the extent size grows so fast, there will certainly not be very many extents. However, since one soon gets very large extents, space may be wasted. One would choose this if one has actually no idea how large the table will be.
- The extent size should be a multiple of `DB_BLOCK_SIZE` $*$ `DB_FILE_MULTIBLOCK_READ_COUNT`.

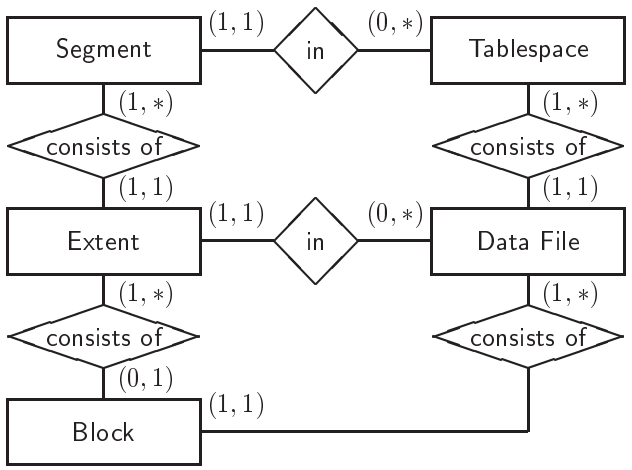
Extents (4)

Example:

- A tablespace consists of two data files, File 1 and File 2.
- Table R consists of a three extents: (1) Block 1 to 4 in File 1, (2) Block 5 to 8 in File 1, and (3) Block 7 to 11 in File 2.
  Oracle does not merge contiguous extents of a table.
- Table S consists of a single extent (Block 1 to 4 in File 2).

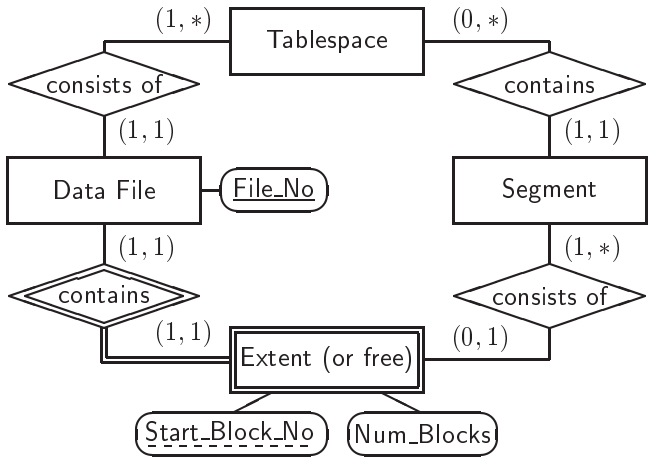


Extents (5)



Extents (6)

Alternative Design:



Extents (7)

- `DBA_EXTENTS` contains one row for each extent.
  It has the following columns:
  - `OWNER`, `SEGMENT_NAME`, `PARTITION_NAME`: Identification of the segment to which this extent belongs.
  - `SEGMENT_TYPE`, `TABLESPACE_NAME`: See `DBA_SEGMENTS`.
  - `EXTENT_ID`: Extent number within the segment: $0, 1, 2, \ldots$
  - `FILE_ID`: File containing the extent.
  - `BLOCK_ID`: Start of the extent within the file.
  - `BYTES`, `BLOCKS`: Size of the extent.
  - `RELATIVE_FNO`: Relative file number of first block (?).

Extents (8)

- `DBA_FREE_SPACE` contains one row for each contiguous sequence of blocks that is currently not allocated to a segment ("free extents").
  Oracle checks only from time to time (or when necessary) whether adjacent pieces can be merged ("coalesced").
- Columns are:
  - `TABLESPACE_NAME`, `FILE_ID`: Tablespace and data file.
  - `BLOCK_ID`: First block of free piece.
  - `BYTES`, `BLOCKS`: Size of free piece.
  - `RELATIVE_FNO`: Relative file number of first extent block.
- See also: `USER_EXTENTS`, `USER_FREE_SPACE`.

Extents and Tablespace Declaration (1)

- In the CREATE TABLESPACE command, default values for the extent allocation parameters can be specified:

```
CREATE TABLESPACE USER_DATA
DATAFILE 'D:\User1.ora' SIZE 20M
MINIMUM EXTENT 32K
DEFAULT STORAGE (INITIAL 100K NEXT 50K
                 PCTINCREATE 5
                 MINEXTENTS 1 MAXEXTENTS 50
                 BUFFER_POOL KEEP)
```

- DBA_TABLESPACES shows these default values that are used for all segments in the tablespace unless otherwise specified.

Extents and Tablespace Declaration (2)

- The DEFAULT STORAGE parameters have no meaning for the tablespace itself, they only apply to tables created within it.
- E.g. if one does not specify PCTINCREASE for a table, it will not be 0, but the value defined in the tablespace declaration.
  If one does not define it there, defaults set by Oracle are used: PCTINCREASE=50, 5 blocks for INITIAL and NEXT.
- If one needs to create many similar tables in a tablespace, it is easier to set default values for the tablespace instead of setting the values for each table.
- For temporary segments (created during query evaluation), one cannot explicitly set the physical storage parameters. But default values for the temporary tablespace can be set.

Extent Allocation in Oracle (1)

- This is basically the explanation from the manual. Experiments show that extents are sometimes slightly larger than expected.
- First it searches through the list of all "free extents" of the requested tablespace for an exactly fitting piece of disk space.
  - Of course, the requested extent size is rounded up to the next multiple of DB_BLOCK_SIZE.
    Or to the minimal extent size declared for the tablespace.
  - The first extent must consist of at least two blocks.
    The first block of each segment is the segment header and cannot be used for table data.
- If an extent is found, the data dictionary and the segment header are updated to reflect the allocation of the disk space.

Extent Allocation in Oracle (2)

- If no free space is found that has a size equal to the requested amount, Oracle searches the list again for a piece that is larger than the requested one.
  - If the first piece found is larger by 5 blocks or more, a piece of the requested size is cut off.
  - If the piece found is larger by less than 5 blocks, it is completely allocated as the new extent.
- If all existing pieces of free space are smaller than requested, Oracle merges adjacent pieces. Then both steps are repeated.
- If still no piece is found, and AUTOEXTEND is on for at least one data file, the data file is extended (i.e. more disk space is requested from the OS). Else error: "tablespace full".

Local Extent Management

- Since Oracle8i, free space can alternatively be managed by an array of bits showing which "extents" are allocated.
  For such tablespaces, one can either define a uniform extent size (then one bit is used for each piece of that size) or let Oracle determine the extent size (the algorithm for that is not disclosed in the documentation).
- The bitmaps are stored in each data file, and not in the data dictionary, thus the name.
- When using such bitmaps, there is no need to search for adjacent pieces of free space in order to merge them.
  It also avoids recursive calls: The data dictionary entry might itself need space. Also requires to change a rollback segment.

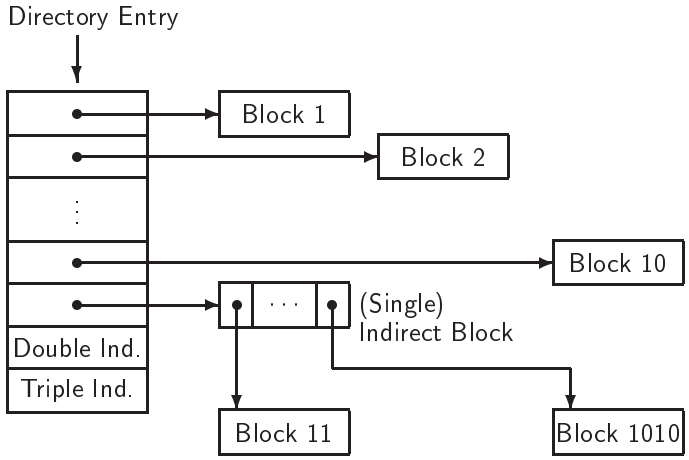
Summary: Tasks of the Disk Manager

- Create a segment with a given initial size.
- Delete a segment.
- Grow a segment by a given number of blocks.
- Shrink a segment (might not be really required).
- Return all blocks of the segment in the logical sequence (i.e. open scan, read next block, close scan).
- It is also necessary to create pointers to blocks (in indexes):
  - If the segment management never moves blocks, pointers can be physical block addresses.
  - Else needed: "Return i-th block of the segment".
- Auxillary function: Free space management for tablespace.

Comparison with UNIX File System (1)

- The file system of any operating system has to implement the same basic functions as the Oracle segment management.
- In UNIX, there are no extents: It manages a list of block addresses for every file.
  It is not a linear list, but a tree (see next slide). In this way, one can efficiently jump to a specific file location.
- If the blocks were randomly distributed over the entire disk, this would be a big performance problem.
- Therefore, a disk is divided in several cylinder groups.
  As far as possible, files are kept in a single cylinder group (the block allocation routine even allows to specify a block near to which the new block should be allocated).

Comparison with UNIX File System (2)



Overview

1. Disk Space Management: Segments, Extents
2. Block Format, TIDs/ROWIDs
3. Block Free Space Management in Oracle
4. Row Format
5. Data Format

Row Manager / Heap Files (1)

- Table rows are stored physically in disk blocks. Normally, each block stores rows from one table only.
  Clusters 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.
- So the basic operations of the row manager are to insert, update, and delete a row, and to return all existing rows.

Row Manager / Heap Files (2)

- 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.
- In many database management systems, the heap file is the only way to store table data.
  Of course, they also have indexes. 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.
  TID (tuple identifier) is a synonym for ROWID.
- 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/TID).
- The access via the ROWID should be especially fast, i.e. normally only a single block access should be required.
- 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.
- Most systems guarantee that ROWIDs do not change for the entire lifetime of a tuple (except export/import).

ROWIDs/TIDs (3)

- The reason that ROWIDs should be kept stable is that there can be many indexes for the same table which would all have to be updated if the ROWID should change.
- Some systems (e.g. Oracle) make it possible for the expert user to use ROWIDs explicitly in tables to speed up joins.
  E.g. foreign keys could be supported by an additional column that contains the ROWID of the referenced tuple.
  But when the DB is exported and imported again, the ROWIDs must be recomputed with the real foreign key.
- 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.

ROWIDs/TIDs (4)

- 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 are similar to object identifiers (OIDs):
  - Even if two rows agree in all attributes (bad design), they can be distinguished by their ROWIDs.
  - The ROWID remains stable even if primary key attributes are updated (bad design, too).
- 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 *`.
- Of course, it is not possible to update the column `ROWID`.
  It is not actually stored, but computed based on 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 Oracle7 ROWID had the form BBBBBBBB.RRRR.FFFF, specifying block (B), row (R), and file (F) in hexadecimal.
- An Oracle8 ROWID includes an object number (segment):
  - `SUBSTR(ROWID,1,6)` is the data object number.
  - `SUBSTR(ROWID,7,3)` is the relative file number.
  - `SUBSTR(ROWID,10,6)` is the block number in the file.
  - `SUBSTR(ROWID,16,3)` is the row number in the block.
- The numbers are no longer printed in hexadecimal, instead a base 64 encoding is used.
  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
```

- The data object number is e.g. shown in `USER_OBJECTS`.
- Rows in a block are numbered 0, 1, 2, ... (holes: rows deleted).
- 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 have 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).
- 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.

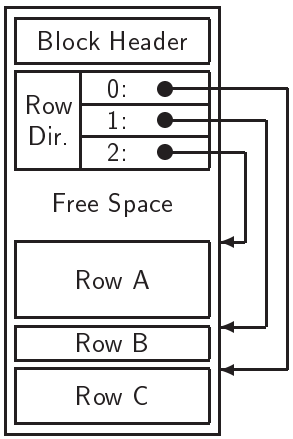
Fixed-Length Rows (2)

- 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.
- 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.
- In this case, 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.

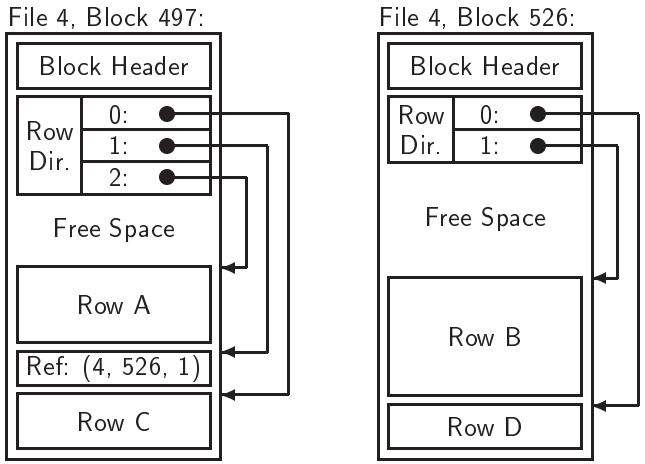
Variable-Length Rows (4)

- The block header may e.g. contain
  - Block address, type of segment, table name.
  - The size of the row directory, size of free space.
  - The next block in the linked list of blocks with free space.
  - A serial version number for this block which is incremented for every update (needed in crash recovery).
  - A bit pattern to detect partially written/destroyed blocks.
    The pattern at the begin and end of the block must agree, they are both inversed on every write.
- Oracle says that the complete block overhead together has in average a length of 84–107 Byte (Gurry/Corrigan use 90).

Variable Length Rows (5)

- 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.
- If a row grows and there is not enough free space left in the block, it must be moved ("migrated") to another block.
- However, we must leave a pointer behind in this block, so that the row can still be found via its ROWID.
  So e.g. its entry in the row directory is still used.

Variable Length Rows (6)



Variable Length Rows (7)

- 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.
- 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 (8)

- 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 database).
  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.
  - Check whether there are 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 in the scan.
  - Close a scan.
- Get a row given its ROWID.
- Insert/Update/Delete a row.

Summary: Row Manager (2)

- So 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 (ROWIDs).
    Of course, the access via ROWIDs should be efficient (usually one block access, sometimes two).
- The row manager determines the block format (a small part might be used by the disk manager to implement segments).

Overview

1. Disk Space Management: Segments, Extents
2. Block Format, TIDs/ROWIDs
3. Block Free Space Management in Oracle
4. Row Format
5. Data Format

Storage Parameters: PCTFREE (1)

- In order to avoid migrated rows, some free space in each block should be reserved for possible growth of the rows.
  So `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 the space reserve (in percent of the block size) which should be left in each block.
- E.g. if `PCTFREE` is 20, and the block size is 2KB (2048 Byte), a block with 410 Bytes of free space is already treated as full (for `INSERT`s).
  It is slightly more complicated since the $(20/100) * 2\text{KB}$ have to remain free after the `INSERT`.

Storage Parameters: PCTFREE (2)

- During physical DB design, it is necessary to estimate how much the row length will grow over the row's lifetime.
  This especially happens when some attributes are null when the row is inserted, and then 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 (shown in `USER_TABLES`, see below), export all rows, empty or recreate the table, and import the rows again.
  And of course `PCTFREE` should be changed. This can be done with `ALTER TABLE`. It effects all future insertions.

Storage Parameters: PCTFREE (3)

- If rows are only inserted and deleted, but not updated (or at least to not become longer by updates), a low value for `PCTFREE` can be chosen (e.g. 5).
- `PCTFREE=10` is a common value (the default set by Oracle).
- `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, see next slide.

Storage Parameters: PCTFREE (4)

- Suppose that rows are inserted with 40 Bytes length, but they all will become 60 Bytes due to updates.
- Then a block of 2048 bytes can contain $(2048 - 90)/(60 + 2) = 31$ rows.
  90 Bytes are the overhead for the block header, 2 Bytes the overhead for the entry in the row directory.
- So $31 * (60 - 40) = 620$ Bytes should remain free at insertion, i.e. `PCTFREE` $= 620/2048 = 30\%$.
- However, this assumes that a block is filled with short rows before the first row grows.
- If this is not the case, `PCTFREE` can be chosen (much) smaller.

Storage Parameters: PCTUSED (1)

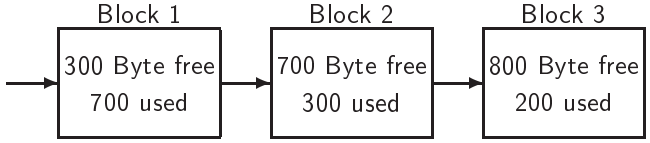
- For each table/segment, Oracle manages a linked list of blocks in which new rows can be inserted.
  Oracle can manage more than one such list (parameter `FREELISTS`) for tables with many concurrent insertions.
- `PCTUSED` determines which blocks are kept on the free list.
- When Oracle wants to insert 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.
- Otherwise (insertion attempt fails), Oracle removes the block from the free list, unless is is filled to less than `PCTUSED` percent (in case the row was exceptionally long).

Storage Parameters: PCTUSED (2)

- 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.

- Exercise: Suppose the block size is 1000. 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.

Storage Parameters: PCTUSED (3)

- 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). So then INSERTs will take a long time since they have to scan a large number of blocks for free space.

- 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.

Storage Parameters: PCTUSED (4)

- Suppose that all rows are 200 Bytes long, PCTFREE is 10, the blocksize is 2048 bytes, the header size is 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 at this point.
- Choosing PCTFREE smaller has no effect for insertions, still 8 rows are inserted until PCTUSED is considered.
- The default value is PCTUSED=40.

Storage Parameters: PCTUSED (5)

- 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

$$(\text{Blocksize} * (100 - \texttt{PCTFREE})/100) - \text{Header Size}.$$

- If PCTUSED is chosen smaller, it might take longer until the block is again considered having free space after deletions.
- E.g. if in the example PCTUSED is chosen 50%, less than $2048 * 0.50 = 1024$ Bytes must be used (i.e. only 4 rows).

CREATE TABLE Syntax

Example:

```
CREATE TABLE STUDENT(
            SSN CHAR(11) PRIMARY KEY,
            FIRST VARCHAR(20),
            LAST VARCHAR(20) NOT NULL)
TABLESPACE USER_DATA
STORAGE(INITIAL 10K
        NEXT 10K
        PCTINCREASE 50)
PCTFREE 20
PCTUSED 60;
```


Full Table Scan

- Oracle has a "high water mark" for each table, that is the number of blocks that were ever used for storing rows.
- In a full table scan, Oracle will read all blocks until this "high water mark".
- If 100 000 rows are inserted into a table, using 1000 blocks, and then all rows are deleted, a full table scan will nevertheless read all 1000 blocks.

  To delete all rows from a table use the `TRUNCATE` command.
- 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.

ANALYZE TABLE (1)

- In order to gather statistical information about a table, e.g. `EMP`, the following Oracle SQL command must be executed:

  ```
  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 every contained rows.

    This is the "high water mark" mentioned above.
  - 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 current.
- E.g. if it wanted to keep the current number of rows of each table in the data dictionary, any insertion on the table would lock the data dictionary entry for the table.

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

ANALYZE TABLE (3)

- One should execute the ANALYZE TABLE again from time to time, at least after significant changes in the size of the table.
- 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 (4)

- The output of the ANALYZE TABLE command is stored in the data dictionary tables TABS and COLS.

  The command itself prints only "Table analyzed.".
- TABS is a synonym for USER_TABLES. It contains one row for each table owned by the current user (not including views).
- TABS has 44 columns, e.g. (continued on next slides):
  - 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 statement.

ANALYZE TABLE (5)

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

    This is the "high water mark" mentioned above (i.e. blocks that every contained rows), not the total number of blocks allocated for the table.
  - EMPTY_BLOCKS: Number of allocated, but never used data blocks of the table.

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

ANALYZE TABLE (6)

- Columns of TABS = USER_TABLES, continued:
  - AVG_ROW_LEN: average length of a row in bytes.
  - AVG_SPACE: Average amount of free space (in bytes) in a block allocated to the table (and below high water mark).
  - AVG_SPACE_FREELIST_BLOCKS: Average amount of free space in blocks on the free list (used for insertions).
  - NUM_FREELIST_BLOCKS: Number of blocks on the free list (does not include blocks which were never used).
  - LAST_ANALYZED: Date of last ABALYZE TABLE.
- All columns that contain output from the ANALYZE TABLE are null until the table is analyzed for the first time.

ANALYZE TABLE (7)

- Some columns in `COLS` = `USER_TAB_COLUMS`:
  - `TABLE_NAME`, `COLUMN_NAME`: Identification of the column.
  - `NUM_DISTINCT`: Number of distinct data values.
  - `NUM_NULLS`: Number of rows for which this column is null.
  - `LOW_VALUE`, `HIGH_VALUE`: Smallest and greatest value.
    They are shown in the internal format (not readable).
- These entries remain only for backward compatibility,
  Oracle suggests to use now `USER_TAB_COL_STATISTICS`.
  `USER_PART_COL_STATISTICS` for partitioned tables.
- More information can be collected with histograms
  (explained in the chapter about query optimization).

Overview

1. Disk Space Management: Segments, Extents
2. Block Format, TIDs/ROWIDs
3. Block Free Space Management in Oracle
4. Row Format
5. Data Format

Row Format (1)

- Normal row format in Oracle (not chained, not clustered):

Row Header	1st Col. Length	1st Column Data	2nd Col. Length	2nd Column Data	...

- The row header contains the number of columns and the number of chain pieces (3 bytes in total).
- The column length is encoded in one byte if below 250. Otherwise it needs three bytes.
- The length of the column data depends on the data type.
  E.g. a `VARCHAR`-string with 5 characters needs 5 byte.
  See below for more information.

Row Format (2)

- In the order of columns is normally the order of declaration in the `CREATE TABLE` statement.
  But `LONG` columns are moved towards the end. Columns added with `ALTER TABLE` are also added at the end.
- Null values need only the length byte (0).
  If the columns at the end are all filled with null values, they are not stored at all.
- The Oracle Row Format is quite compact.
- However, if Oracle wants to access e.g. the fifth attribute, it needs to look at each of the preceeding column lengths.
- Normally the bottleneck is disk I/O, but not the CPU.

Overview

1. Disk Space Management: Segments, Extents
2. Block Format, TIDs/ROWIDs
3. Block Free Space Management in Oracle
4. Row Format
5. Data Format

Data Formats (1)

- The storage size of any data value can be determined with the function `VSIZE`:

```
SELECT SSN, VSIZE(SSN), LNAME, VSIZE(LNAME)
FROM   STUDENT
```

- This is also possible without storing the value in a table:

```
SELECT VSIZE(-1.2), VSIZE('abc')
FROM   DUAL
```

- `CHAR`(n): A fixed-length string is stored in n Bytes (one character per byte, filled with blanks to the length n).
- `VARCHAR`(n): Here only the actual characters are stored. (If a `VARCHAR(10)` column contains `'Jim'`, it needs 3 Byte.)

Data Formats (2)

- The strings are stored in the DB character set (e.g. ASCII).
  The character set can be chosen at DB creation time, but until Oracle 8i it could not be changed later. Now it can be changed to supersets that have the same codepoint values for the subset. Clients can use a different character set and Oracle does the conversion. Oracle can manage multi-byte character sets. Oracle has also datatypes `NCHAR`/`NVARCHAR` for storing strings in a second, national character set.
- `RAW`(n): Variable-length string of data which is not interpreted / not converted between different character sets.
  Input/output is in form of a string of hexadecimal digits. `RAW(10)` means max. 10 Byte, but `'00FF'` needs 2 Byte.

Data Formats (3)

- `NUMBER`(p), `NUMBER`(p,s): Numbers are stored in scientific notation with mantissa and exponent. E.g. $123 = 1.23 * 10^2$.
  It seems that Oracle really stores it as $1.23 * 100^1$.
- The exponent needs always one byte, the mantissa needs one byte per two digits (leading/training zeros are not stored).
  Even if the column is `NUMBER(30)`, 123 needs only 3 Byte.
- So a positive number with n digits needs $1 + \mathrm{ceil}(n/2)$ bytes.
  The Oracle 8 Concepts manual says something different.
- Negative numbers need one more byte for the sign.
- Oracle can store up to 38 significant digits, so a number needs at most 21 Byte (or 20 Byte if positive).

Data Formats (4)

- ROWID: Physical pointer to a row, needs 10 bytes.
  Maybe: Object 4 Byte, File+Block 4 Byte, Row 2 Byte (?).
- DATE: Timestamp (Date and Time), needs 7 Byte.
  Year: 2 Byte, Month: 1 Byte, Day: 1 Byte, Hours: 1 Byte, Minutes: 1 Byte, Seconds 1 Byte. In the default format for input/output (DD-MON-YY) only the date portion can be specified and Oracle assumes 0:00am (midnight). However, SYSDATE returns not only the current date, but also the time.
- To see the internal representation of e.g. 123, use

```
SELECT DUMP(123, 16) FROM DUAL
```

  The bytes are printed in hexadecimal notation (thus 16).
  This also works with other data types, e.g. DUMP('ab',16).

Experiments/Exercises (1)

- CREATE TABLE R(A NUMBER(4), B VARCHAR(50)).
- INSERT INTO R VALUES (12, 'abcde').
- What will be the output of this query?

```
SELECT A, VSIZE(A), B, VSIZE(B) FROM R;
```

- ANALYZE TABLE R COMPUTE STATISTICS.
- What will be the output of this query?

```
SELECT AVG_ROW_LEN FROM TABS
WHERE  TABLE_NAME = 'R';
```

- TABS also reports INITIAL_EXTENT=10240, BLOCKS=1, NUM_ROWS=1, EMPTY_BLOCKS=3, AVG_SPACE=1944, NUM_FREELIST_BLOCKS=1. Please explain (blocksize is 2048).

Experiments/Exercises (2)

- When another row is inserted, e.g. (34,'uvwxy'), the AVG_SPACE shrinks to 1930. Please explain.
- When this row is deleted again, AVG_SPACE grows only to 1942. Why?
- The procedure on the next slide is used to insert rows of the above length until the first block is full. 125 rows are inserted before the system starts to use a second block.
  This table is declared with PCTFREE=10. TABS shows e.g.: BLOCKS=2, NUM_ROWS=126, EMPTY_BLOCKS=2, AVG_SPACE=1076, NUM_FREELIST_BLOCKS=1, AVG_SPACE_FREELIST_BLOCKS=1944. Please explain.

Experiments/Exercises (3)

```
(1)  CREATE OR REPLACE PROCEDURE P AS
(2)     N NUMBER;
(3)  BEGIN
(4)     N := 1;
(5)     WHILE N < 2 LOOP
(6)        INSERT INTO R VALUES(34, 'uvwxy');
(7)        SELECT COUNT(DISTINCT DBMS_ROWID.
(8)                ROWID_BLOCK_NUMBER(ROWID))
(9)        INTO N FROM R;
(10)    END LOOP;
(11) END;
```