Part 3: 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 (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.

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 (blocks can be appended at the end) and shrink (blocks are removed at the end).

  In Oracle, segments shrink only when explicitly requested.

Segments (2)

- 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.
- For every table, Oracle creates a segment inside the tablespace that is mentioned in the `CREATE TABLE`.
- In the same way, each index is stored in a segment.

  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).

Segments (3)

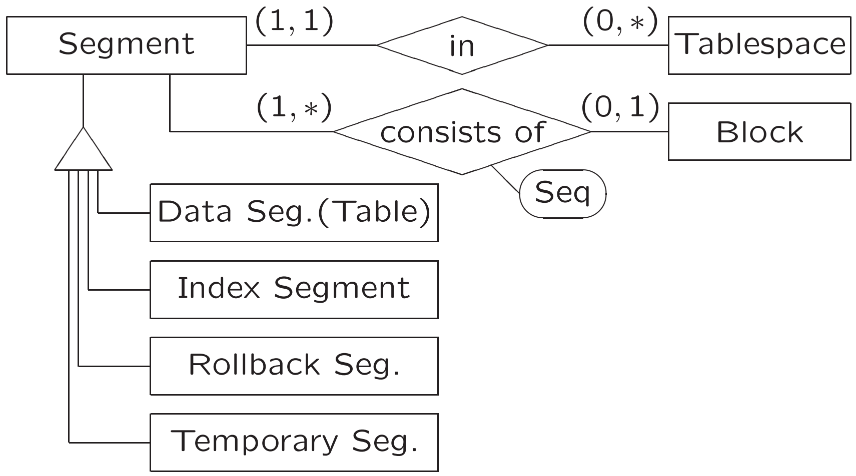
- Normally, the relationship between data segments and tables is 1:1. But in general, it can be n:m:
  - ◇ Partioned tables have more than one segment (usually in different tablespaces on several disks).

    A partitioned table is stored in several pieces, where each piece is basically a table with the same structure: The complete table is then the union of the pieces. When rows are inserted, conditions on the data determine in which piece the row is stored.

  - ◇ Clusters can contain rows from several tables having one or more attributes in common.

    Clusters are an Oracle-specific data structure that permits very efficient joins because the rows to be joined together are already stored together (ideally, in the same block).

Segments (4)



Segments (5)

- 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 (else null).
  - ◇ `SEGMENT_TYPE`: Type of the segment, e.g. `TABLE`.

    `TABLE`, `INDEX`, `CLUSTER`, `TABLE PARTITION`, `INDEX PARTITION`, `ROLLBACK`, `DEFERRED ROLLBACK`, `TEMPORARY`, `CACHE`, `LOBINDEX`, `LOBSEGMENT`.

  - ◇ `TABLESPACE_NAME`: Tablespace in which the segment is stored.

Segments (6)

- Columns of DBA_SEGMENTS, continued:
  - ◇ HEADER_FILE, HEADER_BLOCK: Storage position of segment header block.

    This is the first block of the segment. It contains control information and is not available for table data.
  - ◇ BYTES, BLOCKS: Current size of the segment.

    BYTES is simply BLOCKS ∗ DB_BLOCK_SIZE.
  - ◇ EXTENTS: Number of storage pieces.
  - ◇ INITIAL_EXTENT, NEXT_EXTENT, MIN_EXTENTS, MAX_EXTENTS, PCT_INCREASE: Parameters for allocating storage pieces, see below.

Segments (7)

- Columns of DBA_SEGMENTS, continued:
  - ◇ FREELISTS, FREELIST_GROUPS: For management of blocks with free space within the segment.

    Usually both are 1, but if there are many parallel users that insert data, these parameters can be increased.
  - ◇ RELATIVE_FNO: File containing seg. header block.

    For Parallel Server (Please explain if you know).
  - ◇ BUFFER_POOL: Buffer pool for caching blocks from this segment.
- USER_SEGMENTS lists the segments owned by the current user (some of the above columns are missing).

Extents (1)

- Oracle allocates storage in units called “extents”.
- An extent is sequence of contiguous disk blocks.

  Thus, an extent can be especially fast read from the disk.
- An extent belongs to a single segment and thus to a single table (or index etc.).
- A segment can consist of many extents. But too many extents give bad performance.

  The disk head has to move between the extents (a segment with many extents is “fragmented”). Also, the list of extents should fit into one block. More than 100–500 extents are certainly bad. A single extent would be perfect. One must plan how much space will be needed.

Extents (2)

- Extent sizes are specified in the table declaration:

```
CREATE TABLE STUDENTS(SID NUMERIC(3), ...)
             TABLESPACE USER_DATA
             STORAGE(INITIAL 200K
                     NEXT 50K
                     PCTINCREASE 100)
```

- When the table is created, the initial extent is allocated.

  Although it does not yet contain any rows, it needs disk space for the initial extent (200 KB in the example). The extent size should be a multiple of DB_BLOCK_SIZE ∗ DB_FILE_MULTIBLOCK_READ_COUNT (the size that Oracle reads during a full table scan in one disk access).

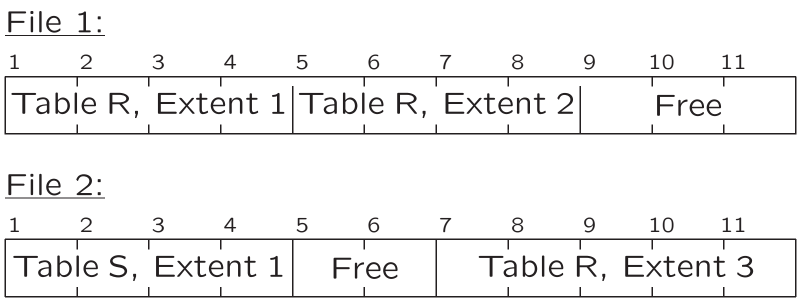
Extents (3)

- Whenever the disk space allocated for a table is full, another extent will be allocated.
- In the example, the second extent will be 50 KByte (NEXT). Normally, all following extents have this size.
- However, with the parameter PCTINCREASE one can request that each following extent will be larger than the previous one (reduces number of extents).

  PCTINCREASE 100 means that the extent size is doubled. Third extent: 100 KB, 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.

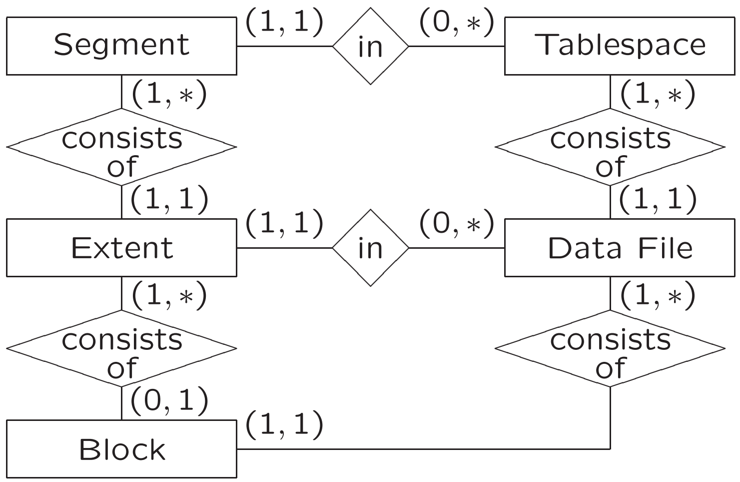
Extents (4)

Example:



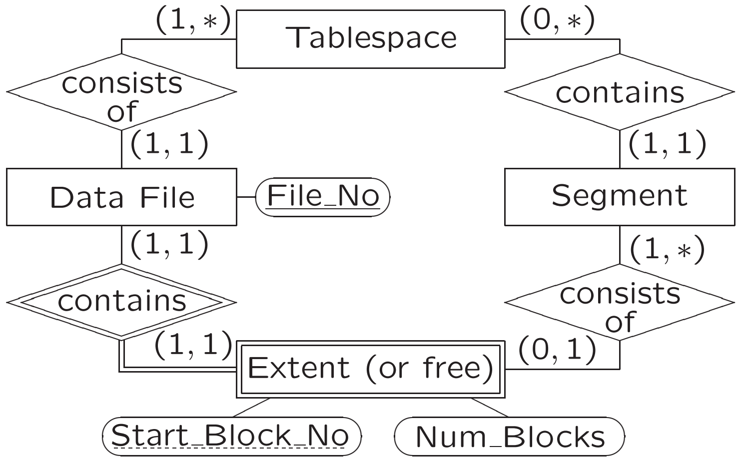
Tables R and S are stored in a tablespace which consists of two data files. Table R has three extents: Block 1 to 4 in File 1, Block 5 to 8 in File 1, and Block 7 to 11 in File 2. Oracle does not merge contiguous extents of a table. Table S consists of a single extent (Block 1 to 4 in File 2).

Extents (5)



Extents (6)

Alternative Design:



Extents (7)

- DBA_EXTENTS contains one row for each extent.
  - ◇ 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 segment.

    Counted from 0, i.e. $0, 1, 2, \ldots$.
  - ◇ 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.

    I am not sure what relative file number means. Please help.

Extents (8)

- DBA_FREE_SPACE contains one row for each contiguous sequence of blocks that is currently not allocated to a segment ("free extents").
  - ◇ TABLESPACE_NAME, FILE_ID: Tablespace, data file.
  - ◇ BLOCK_ID: First block of free piece.
  - ◇ BYTES, BLOCKS: Size of free piece.
  - ◇ RELATIVE_FNO: Relative file no of first extent block.

    DBA_FREE_SPACE might contain two adjacent pieces. Oracle checks only from time to time (or when necessary) whether adjacent pieces can be merged ("coalesced").
- See also: USER_EXTENTS, USER_FREE_SPACE.

TS Declaration: Extents (1)

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

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

- DBA_TABLESPACES lists these values (used for all segments in the tablespace unless otherwise specified).

TS Declaration: Extents (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. The small default values for INITIAL and NEXT show that at least for large tables, it is important to set these parameters

TS Declaration: Extents (3)

- 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 (1)

- The following is basically the explanation from the Oracle manual.

  Experiments show that extents are sometimes slightly larger than expected.
- First, Oracle 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, because the first block of each segment is the segment header and cannot be used for table data.

Extent Allocation (2)

- If an extent is found, the data dictionary and the segment header are updated to reflect the allocation of the disk space.
- 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.

Extent Allocation (3)

- 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 the operation fails and an error message is returned ("tablespace full").

Local Extent Management (1)

- Since Oracle 8i, 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 is not disclosed in the documentation).
- The bitmaps are stored in each data file, and not in the data dictionary, thus the name.

Local Extent Management (2)

- When using bitmaps for free space management, 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.
- The parameters `NEXT` and `PCTINCREASE` are not possible for such tablespaces.

  But `INITIAL` is of course possible.

Summary (1)

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).

Summary (2)

Tasks of the Disk Manager, continued:

- 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".

    Operating systems usually have a call to set the current position in an open file without reading all intermediate blocks (lseek).
- Auxillary function:

  Free space management for tablespace.

UNIX File System (1)

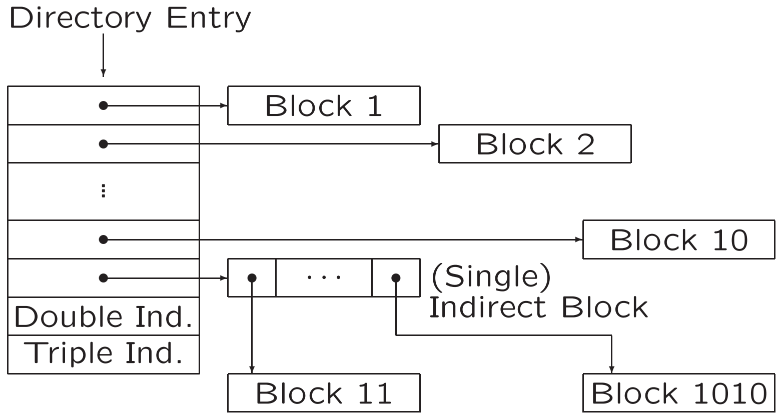
- Segments are very similar to files.
- The file system of any operating system has to implement the same basic functions as the Oracle segment management.
- In UNIX, there are no extents: It manages a list of block addresses for every file.
- It is not a linear list, but a tree (see below).
- In this way, one can efficiently jump to a specific file location.

UNIX File System (2)

- If the blocks of a file were randomly distributed over the entire disk, this would be a big performance problem.
- Therefore, the disk is divided in several cylinder groups (within a cylinder group, the disk head must move only very little).
- If 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.

UNIX File System (3)



Overview

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.

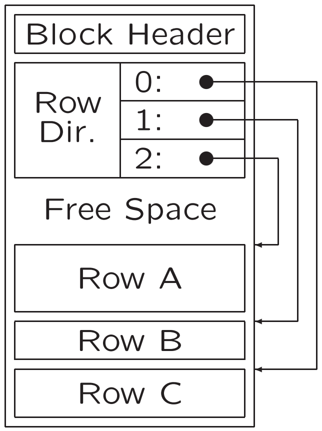
  E.g. because of `VARCHAR` columns.

- 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.

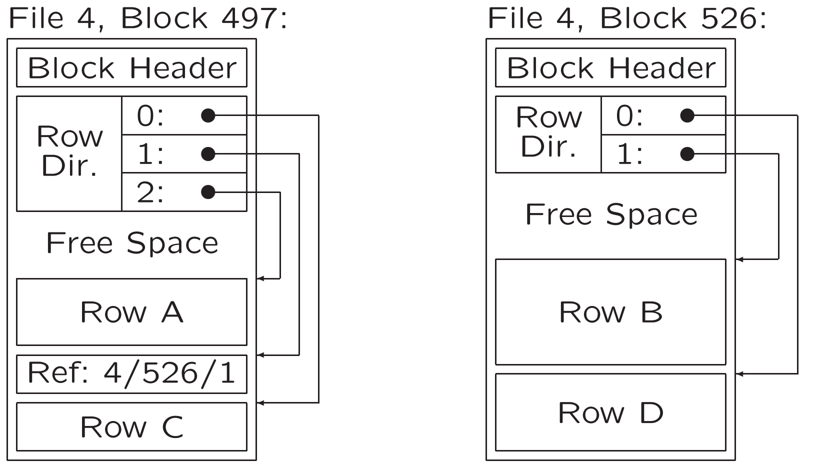
Variable-Length Rows (6)

- 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)

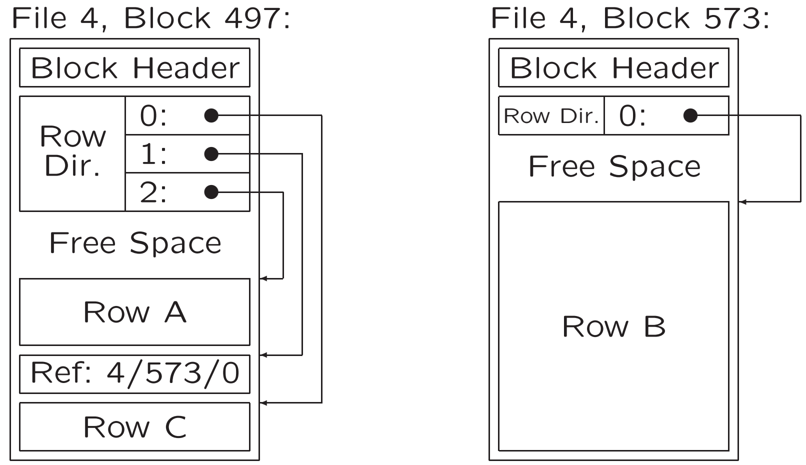
- If Row B should have to move again, the original reference in block 497 is updated. In this way, two block accesses remains the maximum to retrieve a row with given ROWID.

  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.

Overview

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), `PCTFREE` = 0 can be chosen.
- `PCTFREE` = 10 is a common value (default value).
- `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 3-65.

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

$$(2048 - 90)/(60 + 2) = 31$$

  rows.

  90 Bytes are the overhead for the block header, 2 Bytes the overhead for the entry in the row directory.

- Thus, $31 * (60 - 40) = 620$ Bytes should remain free at insertion, i.e. `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:

Block 1	→	Block 2	→	Block 3
300 Byte free 700 used	→	700 Byte free 300 used	→	800 Byte free 200 used

- 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.

CREATE TABLE Syntax

Example:

```
CREATE TABLE STUDENT(
             SID   NUMERIC(4)  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 (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 100 000 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.

Data Dictionary: COLS

- COLS (a synonym for USER_TAB_COLUMS) contains the following information set by the ANALYZE TABLE:
  - ◇ TABLE_NAME, COLUMN_NAME: Identifies 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/greatest value.

    They are shown in the internal format (not readable).
- See also USER_TAB_COL_STATISTICS.

Overview

Row Format (1)

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

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

Row Format (2)

- 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.

- 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.

Row Format (3)

- 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.

  An additional advantage is that when Oracle has to work with these data elements, e.g. strings, it can simply pass a pointer to the length around. So one can see the format also as the concatenation of data values, which encode their own length.

- Exercise: Discuss alternative row formats.

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:

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

- To see the internal representation of e.g. 123, use

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

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

Data Formats (2)

- 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 (3)

- 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 types 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 (4)

- 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 $1.23 * 100^1$.
  Note that NUMBER(p,s) is an Oracle-specific synonym for NUMERIC(p,s).

- 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.

Data Formats (5)

- So a positive number with n digits needs

$$1 + \text{ceil}(n/2)$$

  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 (6)

- 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.

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';
```

Experiments/Exercises (2)

- 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 (3)

- 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 (4)

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