Physical Database Organization and Index Design

ACS 575: Database Systems

Instructor: Dr. Jin Soung Yoo, Professor
Department of Computer Science
Purdue University Fort Wayne

References

- Lemanhieu, et al., Principles of Database Management: The Practical Guide to Storing, Managing and Analyzing Big and Small Data, Ch 13.1
- Elmastri et al., Fundamentals of Database Systems, Ch 17
- Ramakrishnan et al., Database Management Systems, Ch 20
- Oracle Administrator's Guide
- Oracle Developer's Guide

Outline

- Physical Database Organization
- Index Design
- Table Clustering

Physical Database Design

- The purpose of physical database design is to

  translate a logical data model into an internal data model

  - translate the logical description of data into the *technical specifications* for storing and retrieving data

  and design indexes to speed up data access
- Joint responsibility of database designer and DBA
- We focus on how the generic file organization principles are implemented in particularly a relational DBMS.
- **NOTE**: SQL does not impose any standardization on the internal data model or on the way in which a logical relational data model is implemented physically. Different DBMS vectors all have their own approaches.

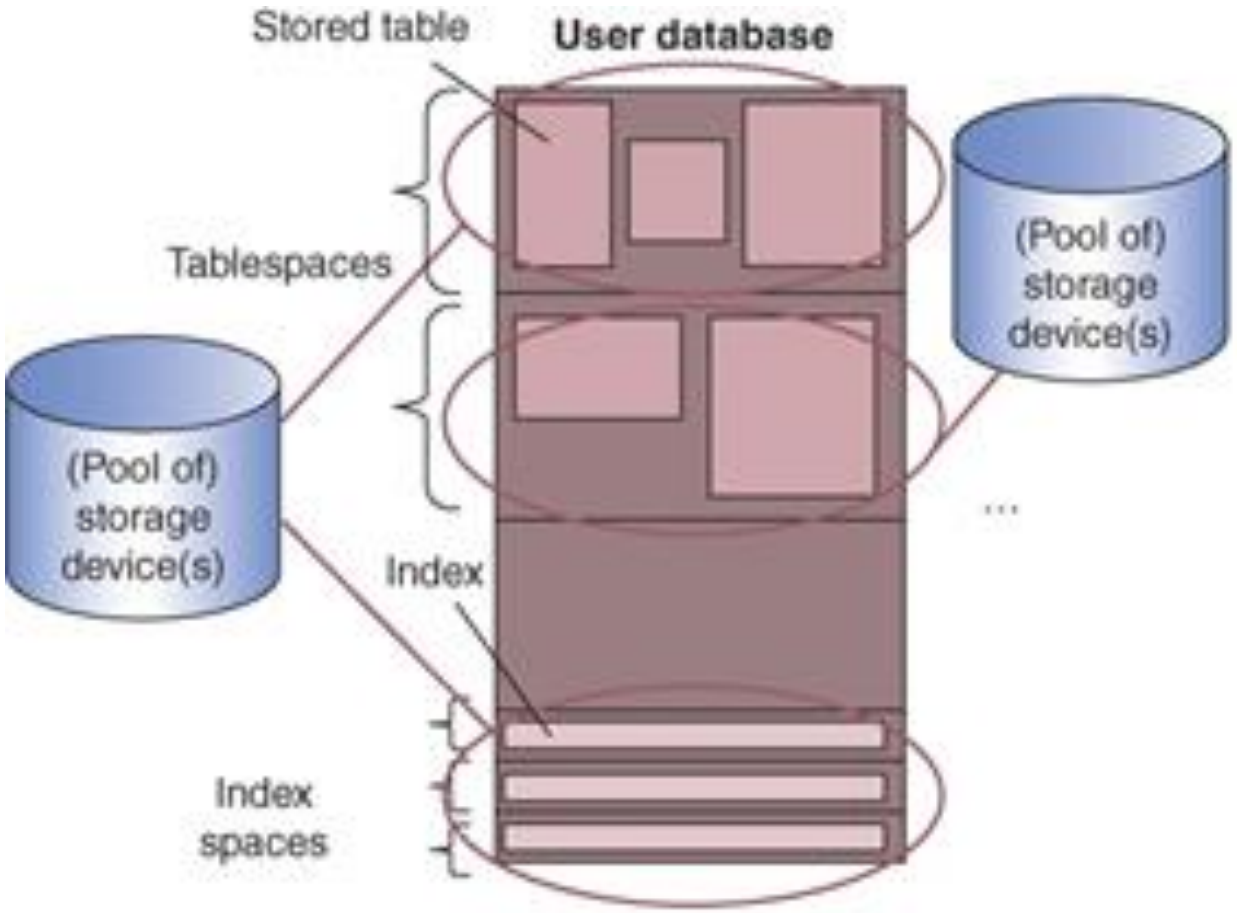
Databases

- **User databases**
  - The user databases have a physical structure that is determined by the database designer and contain data that represents an organization's state
  - A **physical user database** is a collection of index files and data files, organized according to the principles of record organization and file organization
- **System databases**
  - The system database's structure is predefined by the database vendor and holds the catalog tables that support the DBMS's exploitation.

From Database to Tablespace

- Many DBMS implementations introduces an additional level of indirection between a logical database and the physical data files, called a *tablespace*.
- A **tablespace** can be considered as a physical container of database objects. It consists one or more physical files possibly distributed over multiple storage devices
- Every logical table is assigned to a tablespace to be persisted physically (**stored table**)
  - A stored table occupies one or more disk blocks or pages in the tablespace
- A tablespace can contain indexes as well (**index spaces** are used for this purpose)

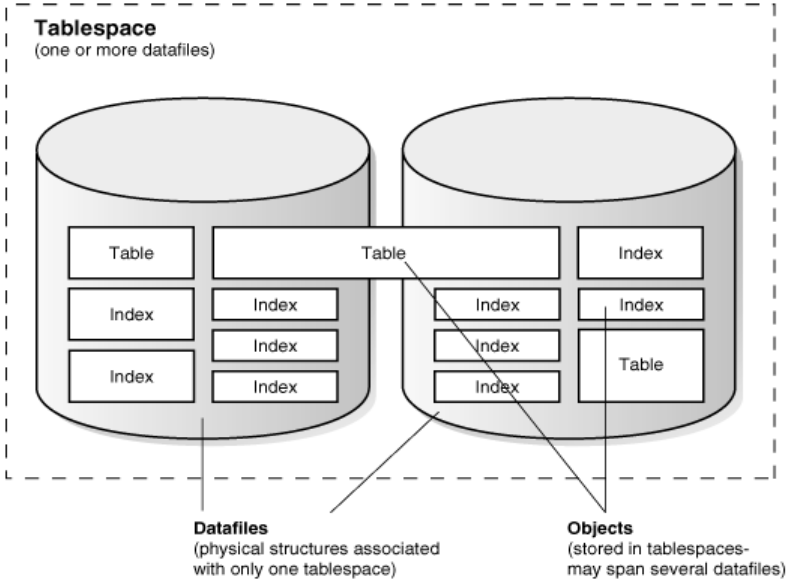
Stored Tables, Indexes, Tablespaces, Index Spaces

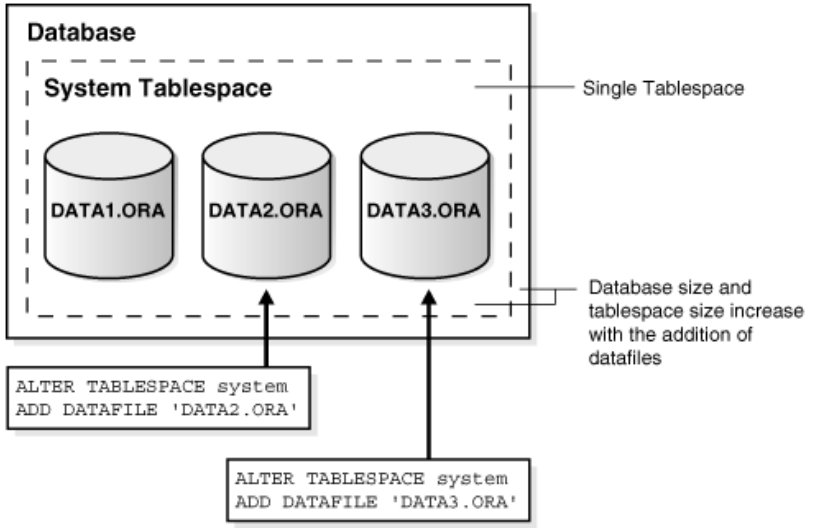


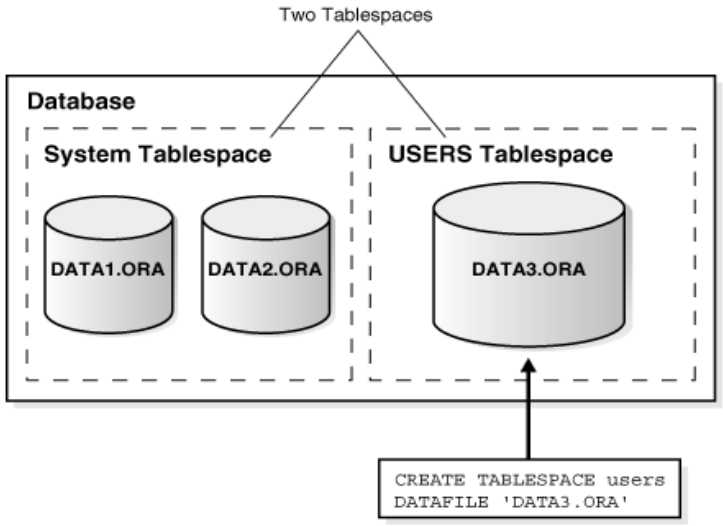
Decisions

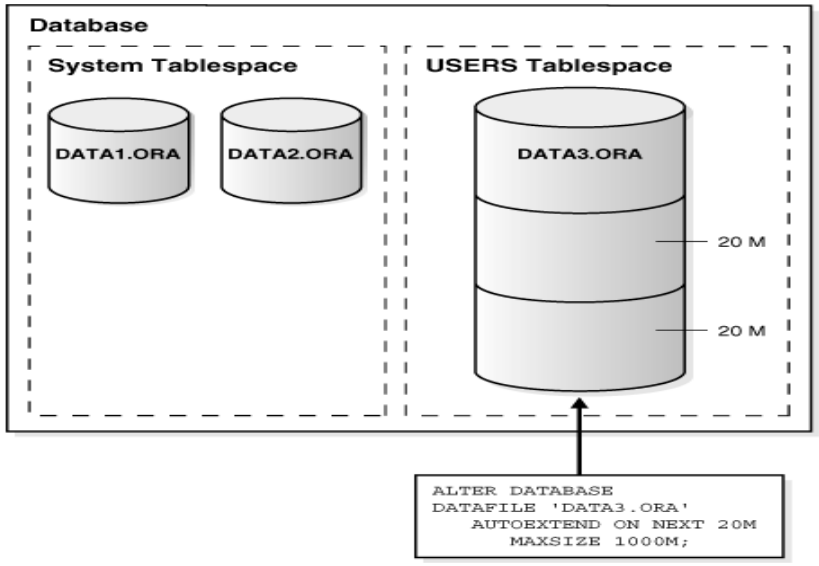
- Choice of storage devices
- On the chosen storage devices, files are created which form the tablespaces (and index spaces).
- Then tables belong to a tablespace.
  - If desired, tables can be distributed over multiple physical files
- Indexes are assigned to tablespaces or to separate index spaces.
  - A primary or clustered index determines the physical ordering of the data records in a file. So creation or deletion of such an index deserves particular consideration

Oracle: Tablespaces and Physical Data files



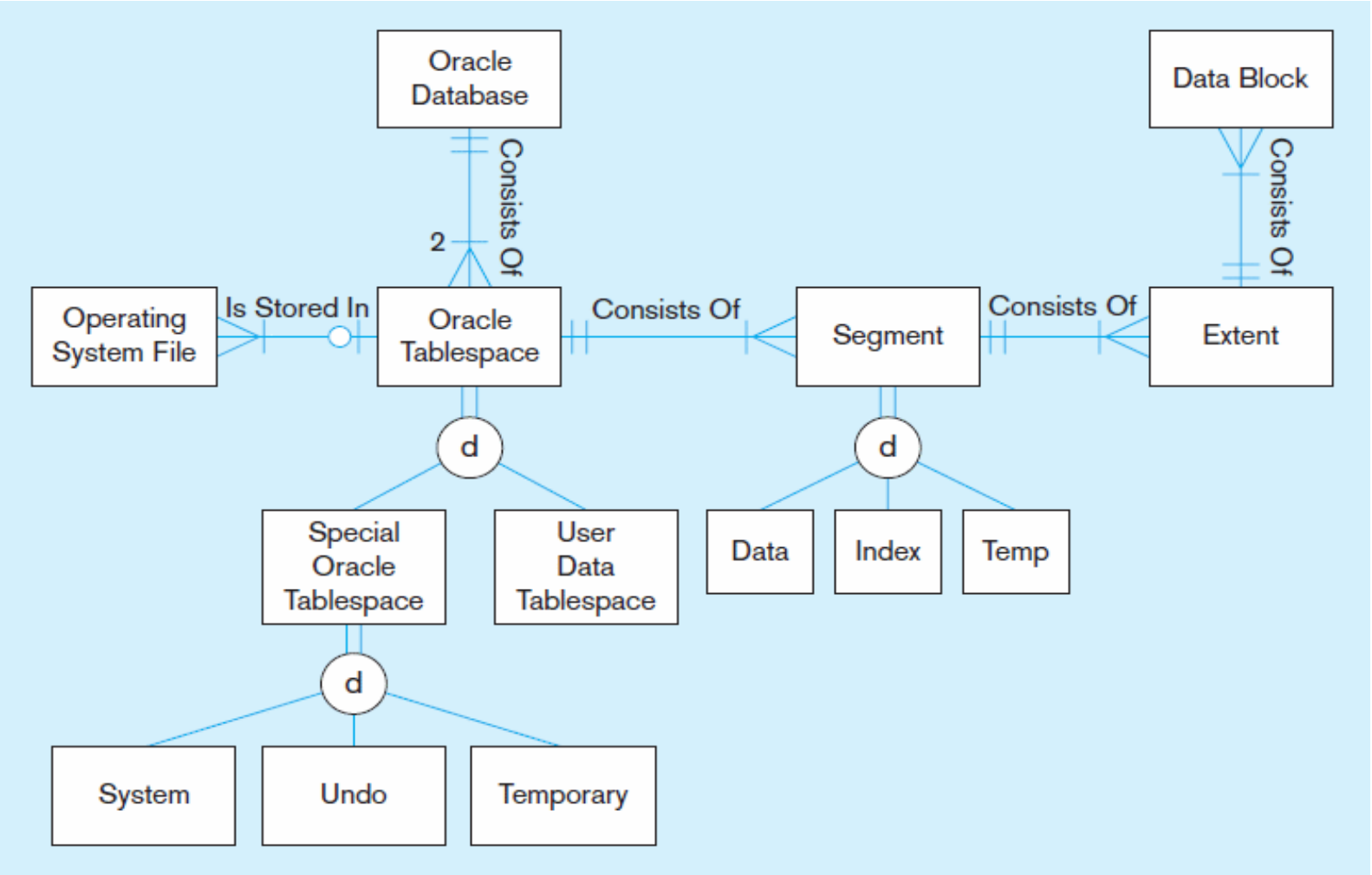






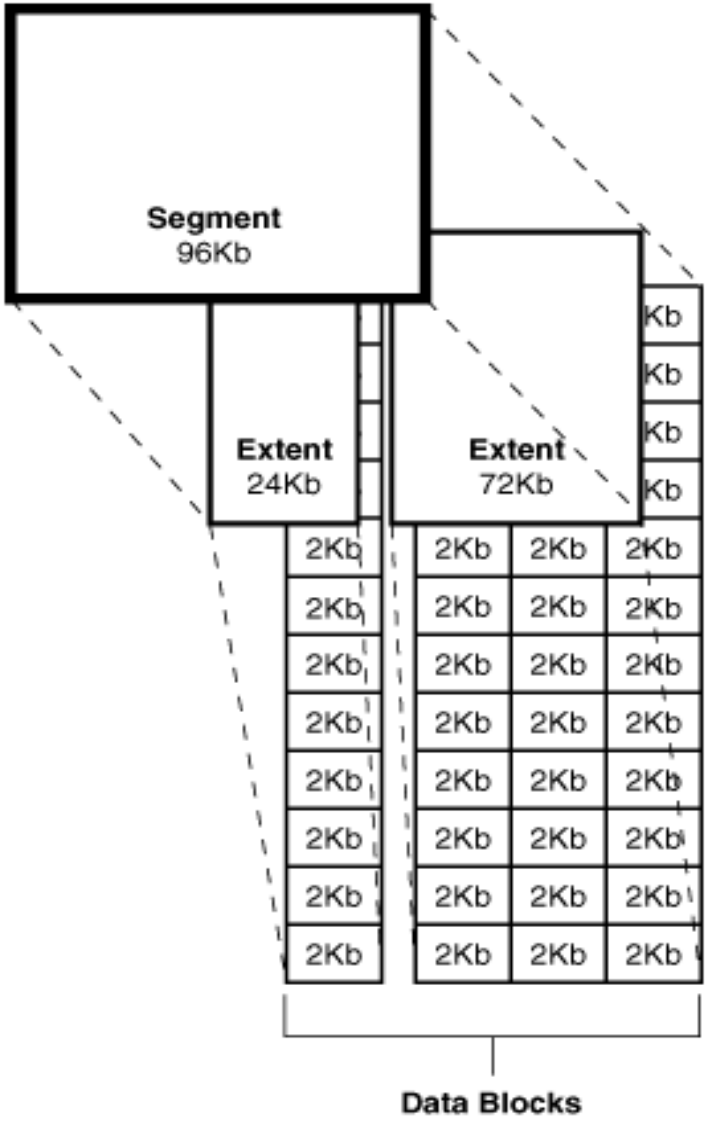
http://download.oracle.com/docs/cd/B19306_01/server.102/b14220/physical.htm

Data Storage in Oracle 11g Environment



From Jeffrey et al., Modern Database Management

Data Blocks, Extents, and Segments



- Oracle Database allocates logical database space for all data in a database.
- **The units of database space allocation** are **data blocks**, **extents**, and **segments**.
- At the finest level of granularity, Oracle Database stores data in **data blocks** (also called **logical blocks**, **Oracle blocks**, or **pages**).
- One data block corresponds to a specific number of bytes of physical database space on disk.

Data Blocks, Extents, and Segments

- The next level of logical database space is an **extent**. An extent is a specific number of contiguous data blocks allocated for storing a specific type of information.
- The level of logical database storage greater than an extent is called a **segment**. A segment is a set of extents, each of which has been allocated for a specific data structure and all of which are stored in the same tablespace.
  - For example, each table's data is stored in its own **data segment**, while each index's data is stored in its own **index segment**. If the table or index is partitioned, each partition is stored in its own segment.
  - When the existing extents of a segment are full, DBMS allocates another extent for that segment. The extents of a segment may or may not be contiguous on disk.

Storage Parameters

- INITIAL
  - The size, in bytes, of the first extent allocated when a segment is created. This parameter cannot be specified in an ALTER statement.
- MINEXTENTS
  - The total number of extents to be allocated when the segment is created. This allows for a large allocation of space at creation time, even if contiguous space is not available.
- MAXEXTENTS
  - The total number of extents, including the first, that can ever be allocated for the segment.

Storage Parameters (Cont.)

- NEXT
  - The size, in bytes, of the next incremental extent to be allocated for a segment.
  - The second extent is equal to the original setting for NEXT. From there forward, NEXT is set to the previous size of NEXT multiplied by (1 + PCTINCREASE/100).
- PCTINCREASE
  - The percentage by which each incremental extent grows over the last incremental extent allocated for a segment.
  - If PCTINCREASE is 0, then all incremental extents are the same size. If PCTINCREASE is greater than zero, then each time NEXT is calculated, it grows by PCTINCREASE.
  - The new NEXT equals 1 + PCTINCREASE/100, multiplied by the size of the last incremental extent (the old NEXT) and rounded up to the next multiple of a block size.

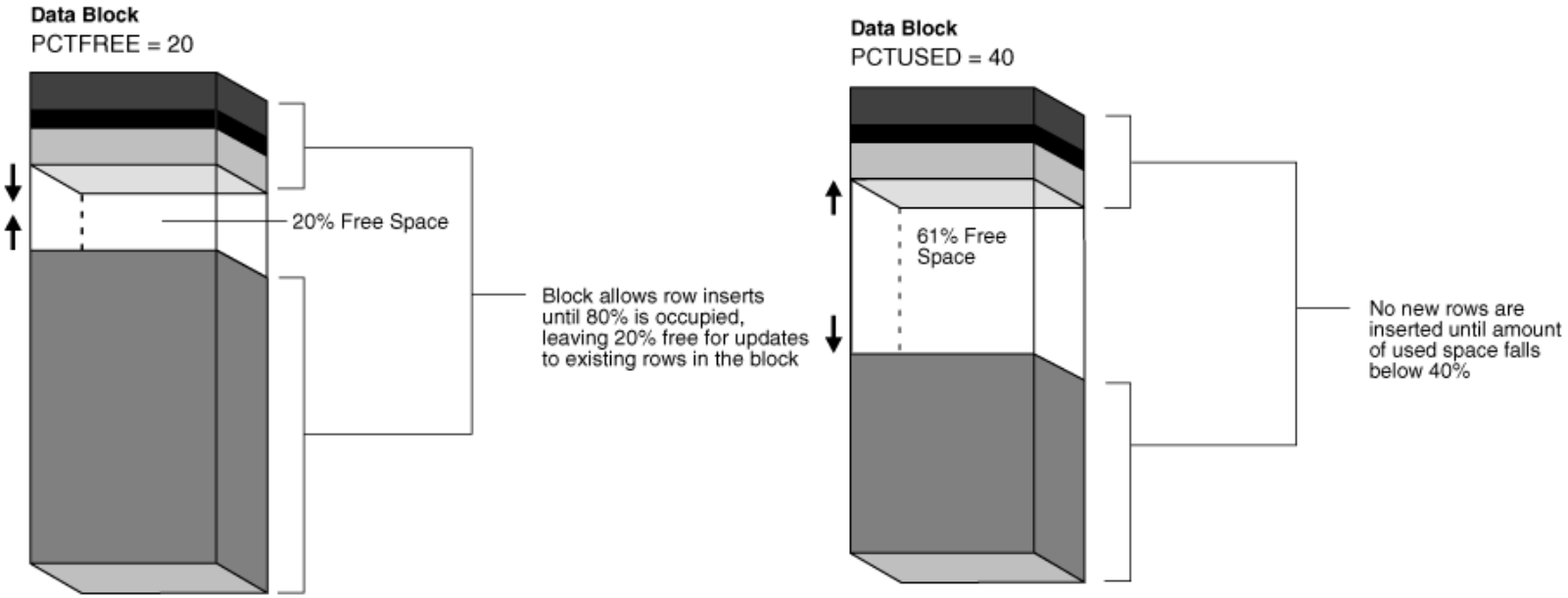
Example: Specifying Table Storage

```
CREATE TABLE divisions
(div_no     NUMBER(2),
 div_name   VARCHAR2(14),
 location   VARCHAR2(13) )
 STORAGE  ( INITIAL 100K  NEXT 50K
            MINEXTENTS 1  MAXEXTENTS 50  PCTINCREASE 5);
```

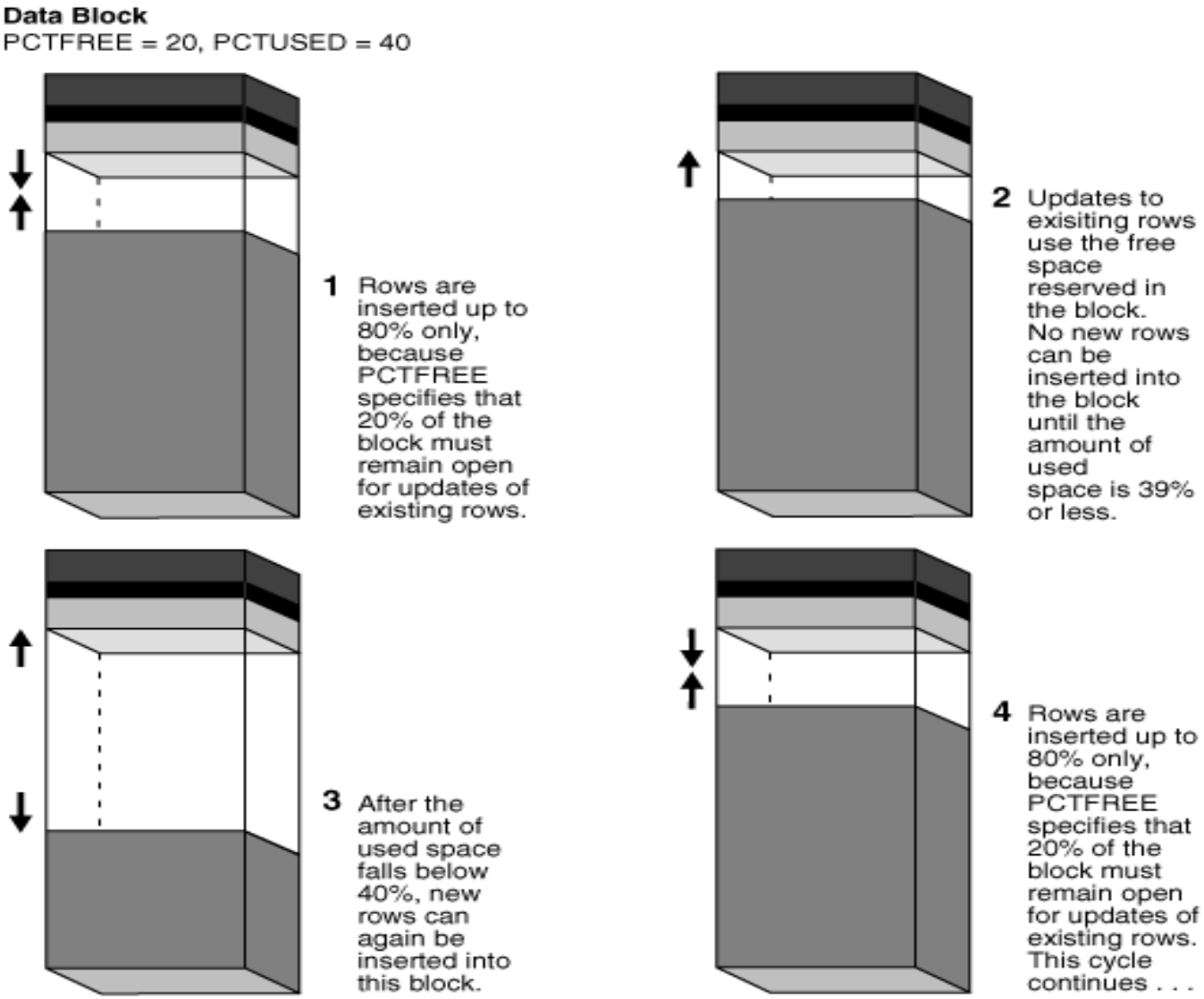
- The MINEXTENTS value is 1, so DBMS allocates 1 extent for the table upon creation.
- The INITIAL value is 100K, so the size of the first extent is 100 K.
- If the table data grows to exceed the first extent, then Oracle allocates a second extent. The NEXT value is 50K, so the size of the second extent will be 50 K.
- If the table data subsequently grows to exceed the first two extents, then Oracle allocates a third extent. The PCTINCREASE value is 5, so the calculated size of the third extent is 5% larger than the second extent, or 52.5 kilobytes. If the data block size is 2 K, then Oracle rounds this value to 52 K.
- If the table data continues to grow, then Oracle allocates more extents, each 5% larger than the previous one.
- The MAXEXTENTS value is 50, so Oracle can allocate as many as 50 extents for the table.

Free Space Management of Data Block

- PCTFREE
  - Set the minimum percentage of a data block to be reserved as free space for possible updates to rows that already exist in that block
- PCTUSED
  - Set the minimum percentage of a data block that can be used for row data plus overhead before new rows are added to the block.



How PCTFREE and PCTUSED Work Together



Example: Specifying Table Storage

```
CREATE TABLE  players
(    code  NUMBER(10) PRIMARY KEY,
     lastname  VARCHAR(20),
     firstname VARCHAR(15),
     position  VARCHAR2(20),
     team VARCHAR2(20)       )
     PCTFREE 10
     PCTUSED 40
     STORAGE
          (INITIAL 25K
           NEXT 10K
           MAXEXTENTS 10
           MINEXTENTS 3);
```

Storage Parameters

- Even though most DBMSs provide ample auto-configuring and self-tuning features, you (DBA) can set **storage parameters** for various data structures such as
  - Tablespaces (used as storage parameter defaults for all segments)
  - Tables, partitions, clusters, materialized views, and materialized view logs (data segments)
  - Indexes (index segments)
  - Rollback segments

CASE Example

- **Case 1:** Common activity includes UPDATE statements that increase the size of the rows.
  - Settings: PCTFREE=20, PCTUSED=40
  - PCTFREE is set to 20 to allow enough room for rows that increase in size as a result of updates. PCTUSED is set to 40 so that less processing is done during high update activity, thus improving performance.
- **Case 2:** Most activity includes INSERT and DELETE statements, and UPDATE statements that do not increase the size of affected rows.
  - Settings: PCTFREE=5, PCTUSED=60
  - PCTFREE is set to 5 because most UPDATE statements do not increase row sizes. PCTUSED is set to 60 so that space freed by DELETE statements is used soon, yet processing is minimized.

CASE Examples (Cont.)

- **Case 3**: The table is very large and storage is a primary concern. Most activity includes read-only transactions.
  - Settings: PCTFREE=5, PCTUSED=40
  - PCTFREE is set to 5 because this is a large table and you want to completely fill each block.

Outline

- Physical Database Organization
- ☞ **Index Design**
- Table Clustering

Index

- Indexes are the most important tuning instrument to the database designer and database administrator
- Indexes directly impact the decisions made by the *query optimizer* with respect to how a query is implemented and optimized physically.

Index Design

- What indexes should we create?
  - Which relations should have indexes?
  - What field(s) should be the search key?
  - Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered?
  - Hash/tree?

Index Creation

- There is no standard SQL syntax for index creation. Still most DBMSs use a similar syntax

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column_name [order]{, column_name[order]})
[CLUSTER]
```

- Example
  - ```
    CREATE UNIQUE INDEX prodnr_index
    ON product(prodnr ACS);
    ```
  - ```
    CREATE INDEX productdata_index
    ON product(prodprice DESC, prodtype ASC);
    ```
  - ```
    CREATE INDEX prodname_index
    ON product(prodname ASC)
    CLUSTER;
    ```
    ```

Index Type

Index type	Impacts physical ordering of tuples	Unique search key	Dense or sparse
Primary	Yes	Yes	Sparse
Clustered	Yes	No	Dense or sparse
Secondary	No	Yes	Dense
		No	Dense or inverted file

Index Type

- **Dense Index**
  - In a dense index, there is an index record for every search key value in the database. This means that every tuple in the table has a corresponding index entry which points to it.
  - Dense indexes can be more efficient for finding records as there's a direct pointer to every record, but they take up more space as the index grows with the table size.
- **Sparse Index**
  - In a sparse index, index records are only created for some of the search key values. These entries generally point to the first occurrence of a search key value, and then a range of tuples can be searched sequentially.
  - Sparse indexes use less space because not every search key value is indexed; however, they might lead to less efficient lookups compared to dense indexes, as additional searching may be required after the initial index lookup.

Reasons for Index Creation

- Efficient retrieval of rows according to certain queries or selection criteria
  - Retrieval is much more efficient for search keys that are indexed
  - Selection criteria aiming at a unique tuple can be supported by a primary index or by a secondary index on a candidate key
  - Selection criteria targeting a set of tuples can be supported by a clustered index or by a secondary index on a non-key attribute type or combination of attribute types
  - Even if some of the attribute type of a composite search key are indexed, retrieval performance will increase

Reasons for Index Creation (cont.)

- Efficient performance of join queries
  - The join is one of the most expensive operations, performance wise, in query execution
  - Indexes can be used to perform the search for related tuples in both tables (i.e., tuples that match the join criteria) in a more efficient way.
  - A foreign key is often indexed

Reasons for Index Creation (cont.)

- Enforce uniqueness on a column value or combination of column values
  - When creating an index on attribute type(s) which requires a uniqueness constraint, DBMS can efficiently enforce the uniqueness constraint with searching only the index to make sure there are no duplicate values.
  - A unique index is generated automatically for every primary key
  - Other unique indexes can be defined explicitly over other candidate keys by the database designer.

Reasons for Index Creation (cont.)

- Logical ordering of rows in table
  - Every index is ordered in a certain way (ACS or DESC)
  - With secondary indexes, we can define numerous logical ordering criteria on the same physically ordered stored table.

Reasons for Index Creation (cont.)

- Physical ordering of rows in table
  - The ordering of the index entries can also be used to determine the physical order of the rows in a stored table.
  - In this case, if the values to the index's search key is unique, the index is a primary index. Otherwise, it is a clustered index
  - Range queries are very efficient.
  - Only one index for every table can determines the physical ordering of the rows. The choice of which attribute type(s) to involve should be made very carefully, in light of the expected frequency and importance of this type of query.

More about Index

- Only **one** primary or clustered index per table but as many secondary indexes as desired
- Cost of index:
  - It consumes storage capacity
  - It may slow down update (insert/update/delete) operation with index update overhead
- Physical data independence: indexes can be added or deleted without affecting logical data model or applications
- Secondary indexes can be constructed or removed without affecting actual data files

General Index Design Guideline

- Attribute types that are typically candidates to be indexes are **primary keys** and **other candidate keys** to enforce uniqueness, **foreign keys** and **other attribute types** that are often used in join conditions, and attributes that occur frequently as selection criteria in queries.
- It is better to avoid attribute types that are large or variable in size (e.g., large character strings or varchar attribute types)
- An index over a small table often induces more overhead than performance gain.
- The proportion of retrieval and update – Typically, retrieval is sped up, updates are slowed down.

Guide for Index Design and Tuning

- (Guideline 1) **Whether to Index**
  - Do not build an index unless some query benefits from it.
  - Add indexes only when absolutely necessary
- (Guideline 2) **Choice of Search Key**
  - Attributes frequently in WHERE
- (Guideline 3) **Index every primary key and most foreign keys in the database**
  - Most joins occurs between primary keys and foreign keys.
  - In most DBMS, when primary keys are explicitly declared in SQL, indexes are formed as a side effect.
  - Even for small tables with a few dozen rows, consider indexes.

Guide for Index Design and Tuning (cont.)

- (Guideline 4) **Multi-Attribute Search Keys**
  - More than one attribute of a relation in WHERE
  - **Index-only strategy**
    - Avoid retrieving tuples from relations.
    - Suppose D.dno has an index and E.dno has an index.
    - E.g., No need to access Employee data file

```
SELECT D.mgr
FROM  Emp E, Dept D
WHERE  E.dno=D.dno
```

      - No attributes of Employees are retained.
      - Compare with

```
SELECT D.mgr, E.eid
FROM  Emp E, Dept D
WHERE  E.dno=D.dno
```

    - E.g., No need to access both Employee and Dept data files.
    - **Very useful, but do not overuse with**

making many extremely query-specific indexes

```
SELECT D.dno
FROM  Emp E, Dept D
WHERE  E.dno=D.dno
```

Guide for Index Design and Tuning (cont.)

- (Guideline 5) **Avoid index with value distribution that are extremely skewed.**
  - Don’t allow just a few index values to have much higher levels of frequency than other values
  - When index values have a highly uneven frequency distribution, the index becomes less effective.
  - Example: if an index were created on a gender column in a database where the number of males is vastly greater than females, the index would be inefficient.

Guide for Index Design and Tuning (cont.)

- (Guideline 6) **Hash vs. Tree Index**
  - A tree index is usually preferable because it supports range query as well as equality query.
    - It is used for column comparisons in expressions that use the =, >, >=, <, <=, BETWEEN AND, or LIKE
  - A hash index is better for a very important equality query, no range queries.
    - It is used only for equality comparisons that use the =.
  - Reference: MySQL tree index vs hash index
    - https://docs.oracle.com/cd/E17952_01/mysql-5.7-en/index-btree-hash.html#:~:text=Hash%20Index%20Characteristics&text=They%20are%20used%20only%20for,find%20a%20range%20of%20values.

Guide for Index Design and Tuning (cont.)

- (Guideline 7) **Whether to Cluster**
  - Choose carefully **one** clustered index for each table
    - A cluster index dictates how data is to be physically stored for a specific table.
    - Any other index on that table cannot be a clustered index.
  - Create a clustered index on the most important query.
    - Particularly efficient on columns that are often searched for ranges of values since data in table is physically sorted on that columns
    - It is good to build a clustered index on a primary key with range searches

Guide for Index Design and Tuning (cont.)

- (Guideline 7) **Whether to Cluster** (cont.)
  - In MS SQL Server or Sybase case
    - The PRIMARY KEY constraints create clustered indexes automatically if no clustered index already exists on the table
    - Otherwise, specify a nonclustered index option on PRIMARY KEY
      - If you have a different attribute or set of attributes that are more frequently queried and would benefit from being the clustered index.
    - E.g., Nonclustered primary key in MS SQL Server

```
Create Table Customer
(CustID number primary key nonclustered,
  …
);
```

Guide for Index Design and Tuning (cont.)

- (Guideline 8) **Balancing the Cost of Index Maintenance**
  - Use attributes for indexes with caution when they are frequently updated.
    - Maintaining an index can slow down frequent update operations.
  - Keep in mind, however, index may speed up an update or delete operation.

    e.g., Increase the salary of a given employee id when id has an index.

```
UPDATE employee
SET salary = salary * 1.1
WHERE id = '7788';
```

  - Avoid or remove redundant indexes

Guide for Index Design and Tuning (cont.)

- (Guideline 9) **Keep up index maintenance on a regular basis**
  - Index maintenance is needed constantly when table rows are constantly updated with insertions, deletions and changes.
  - Index can be deleted and rebuilt

Supplementary: Guidelines for Application-Specific Indexes in Oracle

Reference: Oracle Developer's Guide

Guidelines for Application-Specific Indexes

- **Create Indexes After Inserting Table Data**
- **Index the Correct Tables and Columns**
  - Create an index if you frequently want to retrieve less than 15% of the rows in a large table.
  - Index columns used for joins to improve performance on joins of multiple tables.
  - Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key
  - Small tables do not require indexes

Guidelines for Application-Specific Indexes (Cont.)

- Index the Correct Tables and Columns (Cont.)
  - Some columns are **strong candidates for indexing**.
    - Values are relatively unique in the column.
    - There is a wide range of values (good for regular indexes).
    - There is a small range of values (good for bitmap indexes).
    - The column contains many nulls, but queries often select all rows having a value.
  - Columns with the following characteristics are **less suitable for indexing**
    - There are many nulls in the column and you do not search on the non-null values.

Guidelines for Application-Specific Indexes (Cont.)

- **Limit the Number of Indexes for Each Table**
  - The more indexes, the more overhead is incurred as the table is altered.
- **Choose the Order of Columns in Composite Indexes**
  - The order of columns in the CREATE INDEX statement can affect query performance. In general, put the column expected to be used most often first in the index.

Guidelines for Application-Specific Indexes (Cont.)

- **Gather Statistics to Make Index Usage More Accurate**
  - Example: In Oracle, collect statistics (related to the data distribution, number of rows and others) on the ‘EMPLOYEE’ table that belongs ‘SCOTT’ schema.

```
EXEC  DBMS_STATS.gather_table_stats(‘SCOTT', 'EMPLOYEE’);
```

    - http://www.dba-oracle.com/t_dbms_stats_gather_table_stats.htm
    - https://docs.oracle.com/en/database/oracle/oracle-database/19/arpls/DBMS_STATS.html#GUID-F04388AD-E984-42C9-8489-05B84CD050E2

- **Drop Indexes That Are No Longer Required**

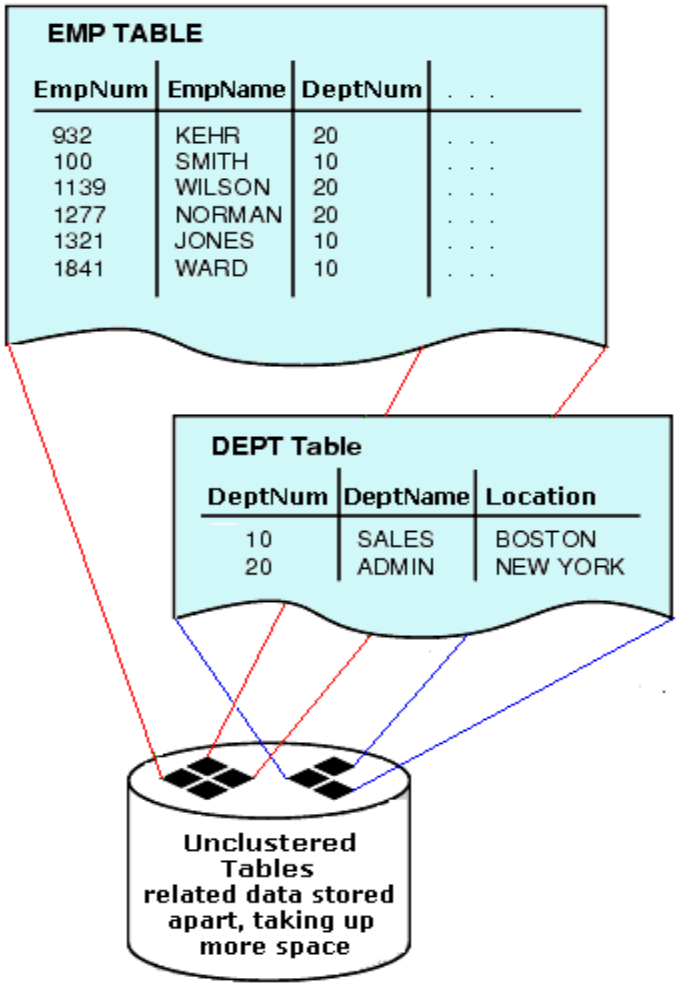
Outline

- Physical Database Organization
- Index Design
- ☞ **Table Clustering**

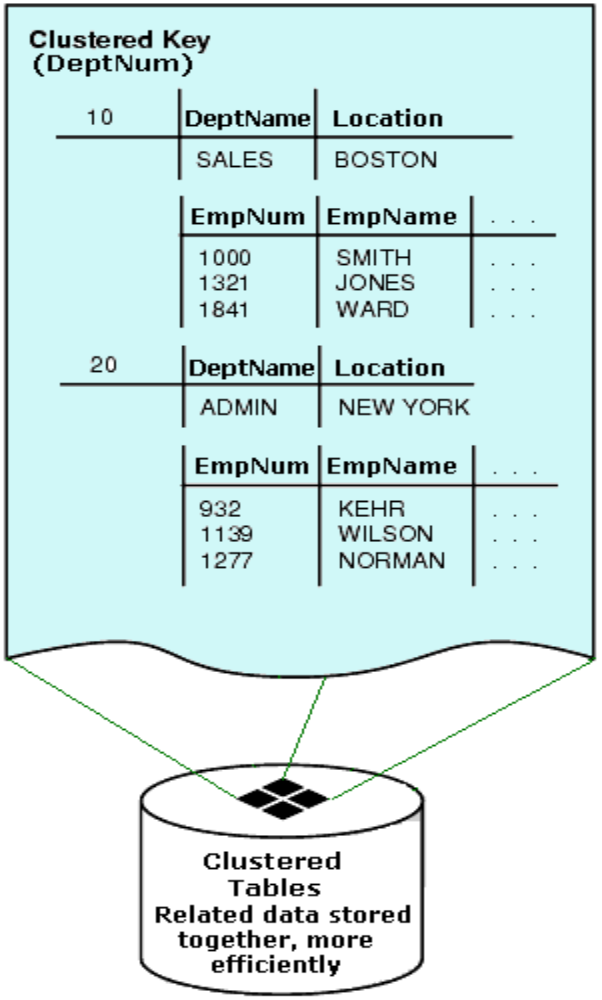
Table Clustering

- **Table clustering** in a relational database system refers to the storage of related data from multiple tables in the same database blocks.
- This technique is used to improve the efficiency of data retrieval operations, especially when performing joins on tables that are often accessed together.

Unclustered Tables vs. Clustered Tables



Unclustered tables



Clustered tables

Key Concepts in Table Clustering

- **Cluster**: A cluster is a schema object that contains data from one or more tables that are related based on common columns. These common columns are typically the columns on which tables are joined.
- **Cluster Key**: The cluster key is the column or set of columns that the tables in the cluster have in common. Rows with the same cluster key value from all the clustered tables are stored together in the same blocks.
- **Hash Clusters**: RDBMS such as Oracle also supports hash clusters, where a hash function is used to determine the database block in which the rows are stored. This can provide very efficient retrieval for equality searches based on the hash key.

Advantages

- **Improved Disk I/O**: Clustering related rows reduces the number of disk reads required to fetch the data.
- **Join Efficiency**: Since related rows from different tables are stored close together, joins on clustered tables can be faster.
- **Data Cohesion**: Clustering maintains the logical cohesion of related data, which can make application development more intuitive.
- **Less Storage:** Less storage is required to store related table and index data because the cluster key value is not stored repeatedly for each row.

When Not to Use Table Clustering

- **Tables are frequently updated**: Clustering can decrease performance because it may necessitate moving rows to maintain cluster integrity
- **Tables require frequent full table scans**: Clustering can lead to less efficient table scans since it groups row based on the cluster key
- **Tables that require truncating**: Truncating a table usually means quickly removing all rows. In clustered table, because rows are grouped by with related data from other tables, truncating operations may become complex or restricted.

Considerations for Implementation

- **Access Patterns**: Clustering is beneficial when tables are often joined on the cluster key. It's less useful if tables are typically accessed independently.
- **Cluster Key Choice**: The columns chosen as the cluster key should be such that the related rows actually need to be retrieved together frequently.
- **Maintenance:** Clusters can require more maintenance than non-clustered tables, as updates to the cluster key may necessitate moving data between blocks.

Example: Table Clustering in Oracle

1. **Define a Cluster**: Use the CREATE CLUSTER statement to define a cluster by specifying its name, the cluster key, and its storage characteristics.
2. **Add Tables to Cluster**: Use the CREATE TABLE statement with the CLUSTER clause to create tables within the defined cluster.
3. **Load Data**: Insert data into the clustered tables. Oracle will automatically store rows with the same cluster key value together.
4. **Gather Statistics**: Use Oracle's DBMS_STATS package to gather statistics on the cluster for the optimizer.
5. **Monitor Performance**: Regularly monitor the performance to ensure that clustering is providing the desired benefits

Example: Table Clustering in Oracle (cont.)

- ```
  CREATE CLUSTER employee_cluster (department_id NUMBER)
  STORAGE (INITIAL 10K NEXT 10K);
  ```
- ```
  CREATE TABLE employees
    (employee_id NUMBER,
     department_id NUMBER,
     name VARCHAR2(100))
    CLUSTER employee_cluster (department_id);
  ```
- ```
  CREATE TABLE departments
    (department_id NUMBER,
     department_name VARCHAR2(100))
    CLUSTER employee_cluster (department_id);
  ```
- ```
  EXEC DBMS_STATS.gather_table_stats('SCOTT',
  'employees', 'employee_cluster');
  ```