module11-tables_clusters.htm; updated June 9, 2012; Some figures shown in these notes are from Oracle document D11321GC11.

Module 11 – Tables and Clusters

Objectives

This lesson focuses on the creation and modification of tables as the primary object used to store database information.

- Learn the Oracle data types.
- Learn the structure of rows in tables, and the management of storage structures in a table.
- Reorganize, truncate and drop tables including dropping table columns.
- Create tables with constraints.
- Create clusters.

A separate set of notes covers index creation applicable to tables.

Introduction

Tables are the most basic data storage object in Oracle. Data are stored in rows and columns.

- Each column has a name and data type.

- Width is specified, e.g, **VARCHAR2(25)**, or precision and scale, e.g., **NUMBER(7,2)** or may be predetermined as with the **DATE** data type.
- Columns may have defined integrity constraints
- **Virtual columns** may be defined - derived from an expression.
- Data encryption can be transparent.

There are four types of tables used to store data in an Oracle database.

Tables: Also called a **regular table** or **ordinary table**, this is the **default** table type and the primary focus of this instructional module. Rows are stored to tables of this type in any order so this is also referred to as a **heap** structure, which is an unordered collection of data rows. A table has a single segment.

Partitioned Tables: This table type is used to build scalable applications.

- A partitioned table has one or more partitions. Each partition stores rows that are partitioned by either range, hash, composite, or list partitioning.
- Each partition is a separate segment, sometimes in different tablespaces (thus the scalable application).

Index-Organized Tables (IOT): The IOT structure is a variant of a B-tree structure.

- Data is stored in an index-organized table in a B-tree index structure in primary key sorted order.
- Instead of maintaining a table in one storage location and the primary key index in another location, each entry in the B-tree also stores non-key column values as well, so the nodes in the tree can become quite large.

- An overflow segment may exist for the storage of longer row lengths.
- These tables provide fast key-based access for queries involving exact matches and range searches.
- Storage requirements are reduced as the primary key is only stored in the table, not duplicated in a separate index.

Clustered Tables: This table type is comprised of a single or group of tables that share the same data block, thus an individual data block can have rows stored in it from more than one table.

- The tables must share a common column(s) that stores a common domain of values called the **cluster key**.
- Clustering is transparent to the applications that query and manipulate the data for the clustered tables.
- The cluster key does not have to be the same as the primary key – it may be the same or a different column set.
- Clusters are created to improve performance. Random access to individual record sets in a cluster is faster, but table scans are slower.

Data Types (Built-In)

Oracle uses several built-in data types to store scalar data, collections, and relationships. These data types vary a bit from the ANSI standard for Structured Query Language – the Oracle data types encompass the ANSI standard.

Data Type	Definition/Use

CHAR – NCHAR	Fixed-length character data types, such as CHAR and NCHAR, are stored with padded blanks. NCHAR is a Globalization Support data type that enables the storage of either fixed-width or variable-width character sets. The maximum size is determined by the number of bytes required to store one character, with an upper limit of 2,000 bytes per row. The default is one character or one byte, depending on the character set.
VARCHAR2 - NVARCHAR2	Variable-length character data types use only the number of bytes needed to store the actual column value, and can vary in size for each row, up to 4,000 bytes. VARCHAR2 and NVARCHAR2 are examples of variable-length character data types.
NUMBER	Numbers in an Oracle database are always stored as variable-length data. They can store up to 38 significant digits. Numeric data types require: One byte for the exponent; One byte for every two significant digits in the mantissa; One byte for negative numbers if the number of significant digits is less than 38 bytes.
DATE	The Oracle server stores dates in fixed-length fields of seven bytes. An Oracle DATE always includes the time.
TIMESTAMP	This data type stores the date and time including fractional seconds up to nine decimal places. TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE can use time zones to factor items such as daylight savings time. TIMESTAMP and TIMESTAMP WITH LOCAL TIME ZONE can be used in primary keys, TIMESTAMP WITH TIME ZONE cannot.
RAW	This data type enables the storage of small binary data. The Oracle server does not perform character set conversion when RAW data is transmitted across machines in a network or if RAW data is moved from one database to another using Oracle utilities. The number of bytes needed to store the actual column value, and can vary in size for each row, up to 2,000 bytes.
LONG, LONG RAW, and LOB	CLOB and LONG store large fixed-width character data. NCLOB stores large fixed-width national character set data. BLOB and LONG RAW store unstructured data. LONG and LONG RAW data types were previously used for unstructured data, such as binary images, documents, or geographical information, and are primarily provided for backward compatibility. These data types are superseded by the LOB data types. LOB data types are distinct from LONG and LONG RAW, and they are not interchangeable. LOBs will not support the LONG application programming interface (API), and vice versa.

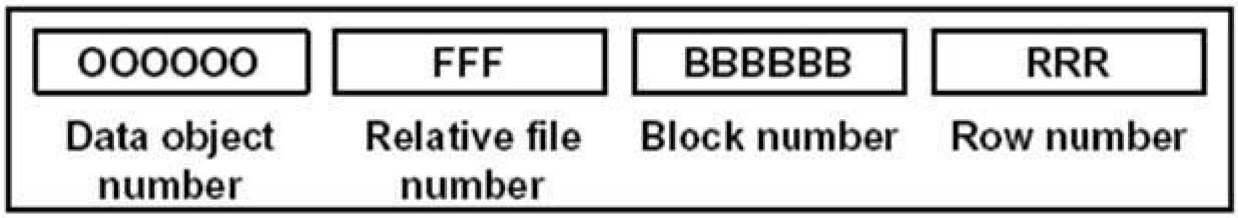
ROWID and UROWID	ROWID and UROWID can be queried along with other columns in a table. ROWID is a unique identifier for each row in the database. ROWID is not stored explicitly as a column value. Although the ROWID does not directly give the physical address of a row, it can be used to locate the row. ROWID provides the fastest means of accessing a row in a table. ROWIDs are stored in indexes to specify rows with a given set of key values. UROWID supports ROWIDs of foreign tables (non-Oracle tables) and can store all kinds of ROWIDs. For example: A UROWID DataType is required to store a ROWID for rows stored in an index-organized table (IOT). The value of the parameter COMPATIBLE must be set to Oracle8.1 or higher to use UROWID.
VARRAY	Varying arrays (VARRAY) are useful to store lists that contain a small number of elements, such as phone numbers for a customer. VARRAYs have the following characteristics: An array is an ordered set of data elements; All elements of a given array are of the same data type; Each element has an index, which is a number corresponding to the position of the element in the array; The number of elements in an array determines the size of the array; The Oracle server allows arrays to be of variable size, which is why they are called VARRAYs, but the maximum size must be specified when declaring the array type.
NESTED TABLES	Nested tables provide a means of defining a table as a column within a table. They can be used to store sets that may have a large number of records such as number of items in an order. Nested tables generally have the following characteristics: A nested table is an unordered set of records or rows; The rows in a nested table have the same structure; Rows in a nested table are stored separate from the parent table with a pointer from the corresponding row in the parent table; Storage characteristics for the nested table can be defined by the database administrator; There is no predetermined maximum size for a nested table.
REFs	Relationship types are used as pointers within the database. The use of these types requires the **Objects** option. As an example, each item that is ordered could point to or reference a row in the PRODUCTS table, without having to store the product code.
XMLType	Stores XML data.

Oracle also allows a system user (programmer usually) to define abstract data types for use within programming applications.

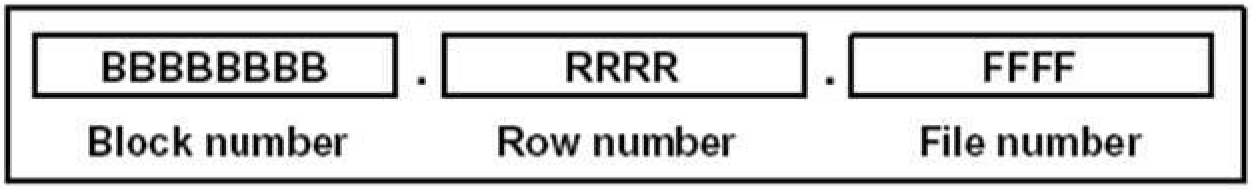
ROWID Format

The ROWID data type is used to store internal row identifiers. The format for ROWID columns is shown in the figures below.

Extended ROWID Format – used in partitioned tables.



Restricted ROWID Format – used in non-partitioned tables.



Data Object Number: Assigned to each data object (table or index) when it is created and is unique within the database.

Relative File Number: Unique to each file within a tablespace.

Block Number: The position of the block containing the row within the file.

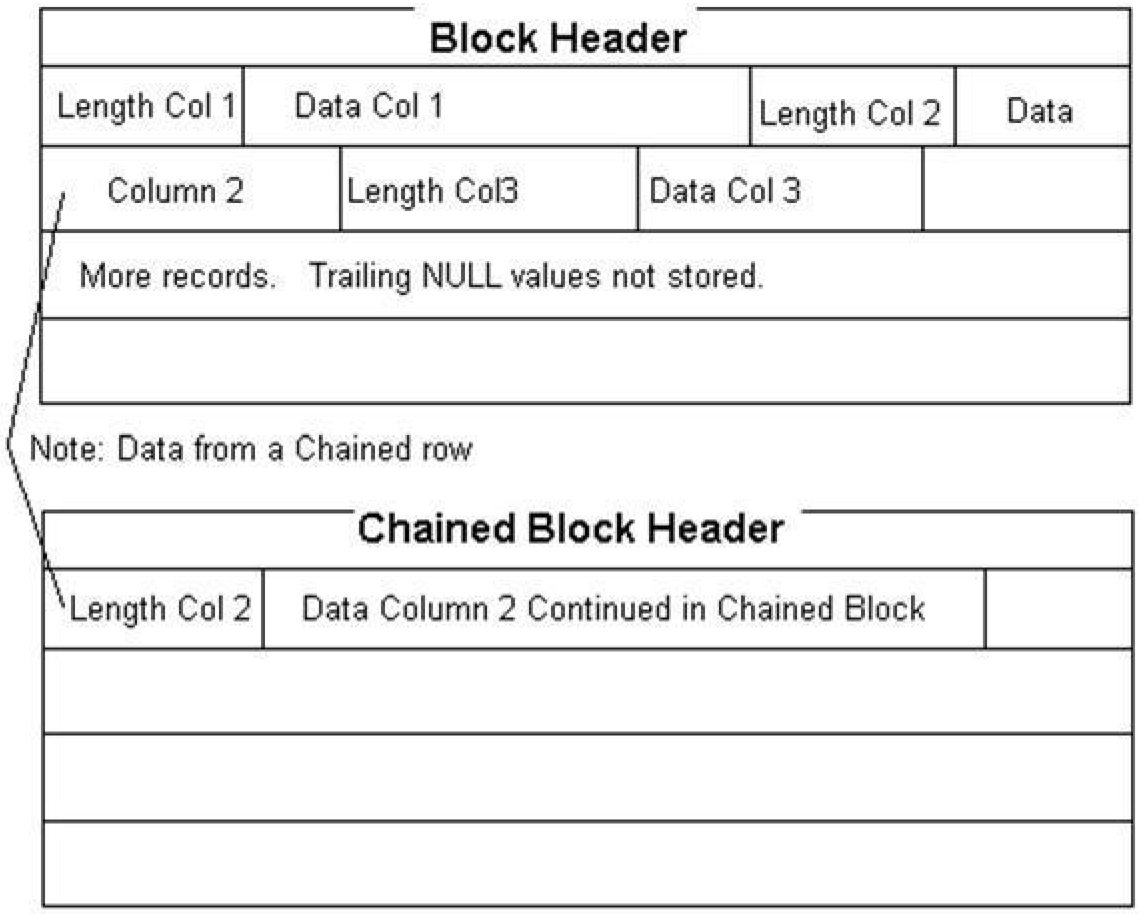
Row Number: The position of the row directory stored in the block header.

```
SELECT DepartmentNumber As Dno, DepartmentName, Rowid
FROM Department;
SELECT Department_id As Dno, Department_Name, Rowid
FROM Departments;

       DNO DEPARTMENTNAME            ROWID
---------- ------------------------- ------------------
         1 Medical Surgical Ward 1   AAAM5yAAEAAAAB0AAA
         2 Radiology                 AAAM5yAAEAAAAB0AAB
         3 Emergency-Surgical        AAAM5yAAEAAAAB0AAC
         4 Oncology Ward             AAAM5yAAEAAAAB0AAD
         5 Critical Care-Cardiology  AAAM5yAAEAAAAB0AAE
         6 Pediatrics-Gynecology     AAAM5yAAEAAAAB0AAF
         7 Pharmacy Department       AAAM5yAAEAAAAB0AAG
         8 Admin/Labs                AAAM5yAAEAAAAB0AAH
         9 OutPatient Clinic         AAAM5yAAEAAAAB0AAI

9 rows selected.
```

Here, **AAAM5y** is the data object number, **AAE** is the relative file number, **AAAAB0** is the block number, and **AAA** (**AAB**, **AAC**) are the row numbers for department #7, #3, and #1, respectively.

Row Structure

Earlier we examined the structure of a row in a database block. We review the structure here with the figure shown below.


Block Header
Length Col 1
Data Col 1
Length Col 2
Data
Column 2
Length Col3
Data Col 3
More records. Trailing NULL values not stored.
Note: Data from a Chained row
Chained Block Header
Length Col 2
Data Column 2 Continued in Chained Block

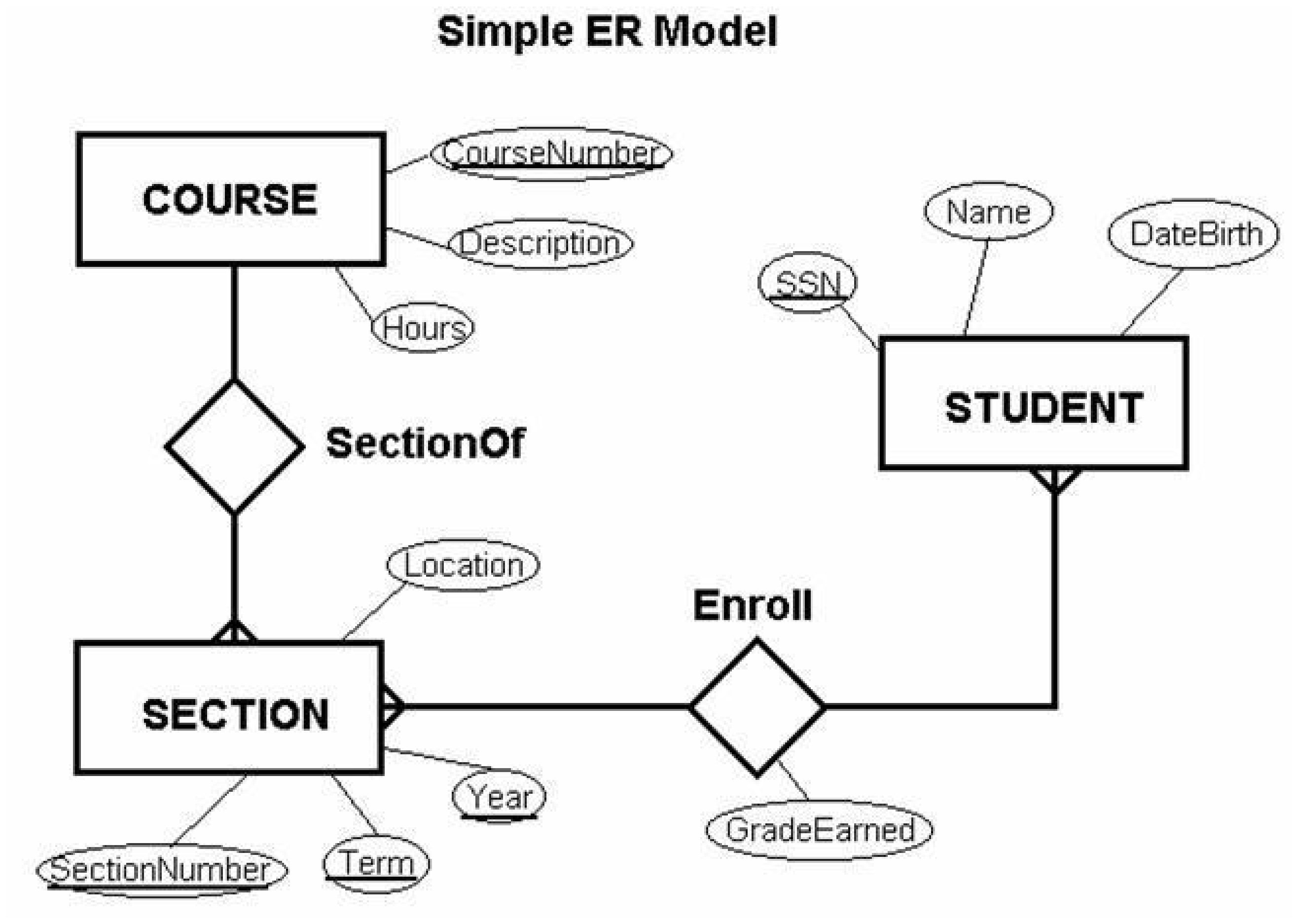
Each row has a **row header** (not shown above) that stores the number of columns in a row, any information about chained columns, and the row lock status.

Columns are stored in the order in which they were defined and trailing **NULL** columns are not stored. Each column stores the column length followed immediately by the column value. Adjacent rows do not have any space in between them. Each row has a slot in the row directory in the block header that points to the row.

Creating a Regular Table

The **CREATE TABLE** command creates relational (regular) tables and object tables. An object table differs from a regular table in that it can use an object type to define a column.

We will base our discussion of creating tables on the following Entity-Relationship diagram as we will include the clauses needed to enforce referential integrity among the rows stored in the tables.



The ER diagram depicts minimal non-key data in order to simplify the discussion without any loss of generality to a real-world situation. The following rules apply:

- The primary key identifier for the **COURSE** entity is the **CourseNumber** attribute.

- The primary key identifier for the **SECTION** entity is the concatenation of the **SectionNumber**, **Term**, and **Year** attributes, each course being assigned a unique **SectionNumber** within a specific **Term** and **Year**.

- The primary key identifier of the **STUDENT** entity is the **SSN** (social security number) attribute.

Types of Data Integrity

Creating tables requires a basic understanding of the various types of data integrity that need to be enforced. Data integrity is often enforced by specifying **constraints**.

Each constraint is named using some type of naming convention.

- Use a suffix or prefix (such as PK for primary key) to denote the type of constraint.
  - **PK** stands for **Primary Key** constraint.
  - **NN** stands for **Not Null** constraint.
  - **FK** stands for **Foreign Key** constraint.
  - **CK** stands for a **Check** constraint.
  - **UN** stands for a **Unique** constraint.
- If you don't name constraints, then Oracle assigns default names to the constraints that are not very meaningful, thus constraints are almost always named by the DBA in order that they may be examined later by using the data dictionary.

Nulls. A **Null** rule defined on a column allows or disallows inserts and/or updates of rows containing a **Null** value. You can specify **Not Null** as an integrity constraint for a column. If a column is allowed to have a null value, you simply do not use an integrity constraint for that column. Example:

```
Student_Name        VARCHAR(50)
    CONSTRAINT Stu_Name_NN NOT NULL,
```

Unique Column Values. This constraint restricts row insertion and updates to unique values for the specified column or set of columns. Example:

```
Student_ID        CHAR(8)
    CONSTRAINT StudentID_UN Unique,
```

Primary Key Values. This constraint specifies uniqueness for the **primary key** and can be applied to one column or a set of columns. Example – this this example creates the constraint and creates a primary key index stored in the tablespace named DATA_INDEX and the index has a PCTFREE of 5%:

```
SSN                CHAR(9)
   CONSTRAINT Student_PK PRIMARY KEY
      USING INDEX TABLESPACE Data_Index PCTFREE 5,
```

Check Values. This is a constraint on a column or set of columns that enforces a specified condition.

- Example – the **Grade** column must have one of the values or must be **NULL**:

```
Grade              CHAR(2)
    CONSTRAINT Grade_Check_CK
        CHECK (Grade IN ('A','B','C','D','E','WP')),
```

- Example the Enrolled_hours column must be less than 6.

```
Enrolled_Hours      NUMBER(2)
    CONSTRAINT Enrolled_Hours_CK
        CHECK (Enrolled_Hours < 6),
```

- A single column can have multiple **Check** constraints that reference the column in its definition.
- There is no limit to the number of **Check** constraints that you can define on a column.
- You must ensure that the constraints do not conflict with one another as Oracle will not test for conflicts.
- Example – the **Grade** column must have one of the values and must also be **NOT NULL**.

```
Grade               CHAR(2)
    CONSTRAINT Grade_Check_CK
        CHECK (Grade IN ('A','B','C','D','E','WP'))
    CONSTRAINT Grade_NN NOT NULL,
```

- A **Check** integrity constraint on a column or set of columns requires that a specified condition be true or unknown for every row of the table. If a data manipulation language (DML) statement results in the condition of the **Check** constraint evaluating to FALSE, the statement is rolled back and the row is not stored in the table because the data would violate the **Check** constraint.

- **Check** constraints enable you to enforce very specific or sophisticated integrity rules by specifying a check condition. The condition of a **Check** constraint has some limitations:
    - It must be a **Boolean** expression evaluated using the values in the row being inserted or updated.
    - It cannot contain subqueries, sequences, the SQL functions SYSDATE, UID, USER, or USERENV, or the pseudocolumns LEVEL or ROWNUM.

Referential Integrity. Oracle supports a number of referential integrity rules to guarantee that the value stored in a column in a table is a valid reference to an existing row in another table. That is, the row inserted must reference a valid row in another table.

For example, when storing a value for **Course_Number** in the **SECTION** table, the database management system software (DBMS) will ensure that the value for **Course_Number** exists within the referenced **Course_Number** column of the **COURSE** table. The **SECTION** table references the **COURSE** table.

Referential Integrity **also** permits foreign keys to have null values; however, this is uncommon. As an example, a student might have a **Major** and this would reference the **MAJORS** Table (not shown in our ER diagram). Some students, however, would not have declared a Major. If a foreign key is part of a primary key, then null values are not allowed.

The referential integrity rules are as follows:

- **Restrict**. This constraint disallows the update or deletion of referenced data. This rule is **not** enforceable by a clause that is used as part of a CREATE TABLE command – enforce it by creating a trigger.

- **Set Null**. This constraint ensures that any action that updates or deletes referenced data causes associated data values in other tables to be set to **Null** when an appropriate referenced value is no longer defined. Example:

```
/* Create Owner Table –

Testing Referential Integrity Clauses */

CREATE TABLE Course (

    Course_Number    CHAR(7)

        CONSTRAINT Course_PK PRIMARY KEY,

            Description      VARCHAR(40)

        CONSTRAINT Description_NN NOT NULL,

    Hours            Number(2)

        CONSTRAINT Hours_Less_12_CK

            CHECK (Hours < 12)

        CONSTRAINT Hours_Greater_Zero

            CHECK (Hours > 0)

    )

PCTFREE 5 PCTUSED 60

TABLESPACE users;


INSERT INTO Course VALUES ('CMIS460','Advanced VB
Programming',3);

INSERT INTO Course VALUES ('CMIS565','Oracle DBA',3);
```

The **TABLESPACE** clause specifies the tablespace where a table is created – here it is **USERS**. If this clause is omitted, a table is created in the default tablespace of the schema owner who creates the table.

The **USING INDEX** clause to cause index creation for the primary key to be stored to the tablespace named **INDEX01**.

The **PCTFREE** clause for the Index tablespace specifies the amount of free space left in data blocks that store index entries. The **PCTFREE** and **PCTUSED** specify parameters for the table.

```
/* Create Member Table */

CREATE TABLE Section (

    Section_Number  Number(6),

    Section_Term    CHAR(6)   CONSTRAINT Term_NN NOT NULL,

    Section_Year    Number(4) CONSTRAINT Year_NN NOT NULL,

    Course_Number   CHAR(7)   CONSTRAINT Section_To_Course_FK

        REFERENCES Course(Course_Number)

        ON DELETE SET NULL,

    Location        CHAR(10)  CONSTRAINT Location_NN NOT NULL,

  CONSTRAINT Section_PK

    PRIMARY KEY (Section_Number, Section_Term, Section_Year)


    )

PCTFREE 20 PCTUSED 65
```

```
TABLESPACE users;


/* Insert valid rows into member for owner record 11 and 22 */

INSERT INTO Section

    VALUES (111111,'Summer', 2012,'CMIS565','FH-3208');

INSERT INTO Section

    VALUES (222222,'Fall', 2012, 'CMIS565','FH-3103');

INSERT INTO Section

    VALUES (111112,'Fall', 2012, 'CMIS460','FH-3208');

INSERT INTO Section

    VALUES (111113,'Summer', 2012, 'CMIS460','FH-0301');
```

The **Course_Number** foreign key in **SECTION** table references the primary key (**Course_Number** shown in parentheses) for the **COURSE** table. The constraint is named **Section_To_Course_FK** and enforced by the **CONSTRAINT** clause.

Note the use of the **CONSTRAINT** and **PRIMARY KEY** clauses to create the **concatenated** primary key and the storage of the key in the **Index01** tablespace.

```
/* Display Owner and Member rows */

SQL> SELECT * FROM Course;


COURSE_ DESCRIPTION                                   HOURS

------- ---------------------------------- ----------
```

```
CMIS460 Advanced VB Programming                     3

CMIS565 Oracle DBA                                  3


SQL> SELECT * FROM Section;


SECTION_NUMBER SECTIO SECTION_YEAR COURSE_ LOCATION

-------------- ------ ------------ ------- ----------

        111111 Summer         2012 CMIS565 FH-3208

        222222 Fall           2012 CMIS565 FH-3103

        111112 Fall           2012 CMIS460 FH-3208

        111113 Summer         2012 CMIS460 FH-0301


/* Test what happens when Course CMIS460 is deleted.

   The Section table will retain the rows but the

   Foreign key column will be NULL */


DELETE FROM Course

    WHERE Course_Number = 'CMIS460';


SELECT * FROM Section;

SECTION_NUMBER SECTIO SECTION_YEAR COURSE_ LOCATION

-------------- ------ ------------ ------- ----------

        111111 Summer         2012 CMIS565 FH-3208
```

```
222222 Fall          2012 CMIS565 FH-3103

111112 Fall          2012         FH-3208

111113 Summer        2012         FH-0301
```

- **Set Default**. This constraint ensures that any action that updates or deletes referenced data values causes associated data values to be set to a default value. This rule is **not** enforceable by a clause that is used as part of a CREATE TABLE command – enforce it by creating a trigger.

- **Cascade**. When a referenced row is deleted, all dependent rows in other tables are also deleted. Likewise, when a key value is updated in a master table, the corresponding foreign key values are updated in referenced tables. Example – this shows the specification using a constraint where the column SSN in the "member" table references the primary key column in the **STUDENT** table.:

```
CONSTRAINT Enr_Stu_SSN_FK FOREIGN KEY (SSN)

    REFERENCES Student

        ON DELETE CASCADE,
```

- **No Action**. This constraint differs from **Restrict** in that it disallows the update or deletion of referenced data, but the rule is checked at the end of a statement or transaction. This is the default action used by Oracle – no rule is necessary.

Testing Referential Integrity and Uniqueness Constraints for the SECTION Table

Insert a good data row into the **SECTION** table.

```
INSERT INTO Section
    VALUES ('444444','Summer', 2012, 'CMIS460','FH-3208');
```

Insert a row with an invalid **NULL** Location field that violates the **Location_NN** constraint. Oracle rejects the insertion and rolls back the command, then displays an appropriate error message.

```
INSERT INTO Section
    VALUES ('555555', 'Fall', 2012, 'CMIS460', NULL);

ERROR at line 2:

ORA-01400: cannot insert NULL into
("DBOCK"."SECTION"."LOCATION")
```

Test the referential integrity constraint between the **SECTION** and **COURSE** tables by attempting to insert a row into **SECTION** for a non-existent **CMIS342 COURSE** value. Again Oracle rejects the insertion and displays an appropriate error message.

```
INSERT INTO Section
    VALUES ('555555', 'Fall', 2012, 'CMIS342',
'FH-3103');

ERROR at line 1:

ORA-02291: integrity constraint
(DBOCK.SECTION_TO_COURSE_FK) violated - parent key not
found
```

Test the primary key constraint by attempting to insert a row with a **duplicate primary key**. Oracle rejects the insertion and displays an error message a row with primary key 001 already exists.

```
INSERT INTO Section
    VALUES ('111111', 'Summer', 2012, 'CMIS270',
'FH-3208');

ERROR at line 1:

ORA-00001: unique constraint (DBOCK.SECTION_PK)
violated
```

Creating the STUDENT Master Table and Inserting Data

The **COURSE** and **SECTION**, created thus far are **master tables**.

One additional master table is required, the **STUDENT** table.

The many-to-many **ENROLL** relationship will be implemented as a table with a concatenated primary key from the **SECTION** and **STUDENT** tables. This is an **intersection** or **association table**.

We will assume that the University's inventory of courses is fairly stable so that the **PCTFREE** value for the tables are very low.

This gives the **CREATE TABLE** commands to create the **STUDENT** table.

```
CREATE TABLE Student (

    SSN                 CHAR(9)

        CONSTRAINT Student_PK PRIMARY KEY

           USING INDEX Tablespace Index01
           PCTFREE 5,

    Student_Name        VARCHAR(50)

        CONSTRAINT Student_Name_NN NOT NULL,

    Account_Balance     NUMBER(7,2),

    Date_Birth          Date Default NULL

    )

PCTFREE 10 PCTUSED 40

Tablespace Data01;
```

Insert several **good** data rows into **STUDENT**.

```
INSERT INTO Student VALUES ('111111111',
   'Charley Daniels',1200.00,'01-MAR-80' );

INSERT INTO Student VALUES ('222222222',
   'Faith Hill',2400.00,'05-FEB-81');
```

Verify the creation of the tables and the indexes by querying the data dictionary.

```
COLUMN Table_Name FORMAT A14;

COLUMN Tablespace_Name FORMAT A15;
```

```
SELECT Table_Name, Tablespace_Name
    FROM Tabs
    WHERE Table_Name = 'STUDENT'

        Or Table_Name = 'COURSE'

        Or Table_Name = 'SECTION';


TABLE_NAME     TABLESPACE_NAME

-------------- ---------------

COURSE         DATA01

SECTION        DATA01

STUDENT        DATA01


COLUMN Index_Name FORMAT A15;

SELECT INDEX_NAME, INDEX_TYPE, TABLE_NAME, UNIQUENESS

    FROM Dba_Indexes

    WHERE Table_Name = 'STUDENT'

        Or Table_Name = 'COURSE'

        Or Table_Name = 'SECTION';



INDEX_NAME      INDEX_TYPE     TABLE_NAME     UNIQUENES

--------------- -------------  -------------- ---------

STUDENT_PK      NORMAL         STUDENT        UNIQUE

COURSE_PK       NORMAL         COURSE         UNIQUE
```

```
SECTION_PK      NORMAL          SECTION         UNIQUE
```

Creating the ENROLL Table and Testing Constraints

Next create the **ENROLL** table which implements the many-to-many relationship between **STUDENT** and **SECTION**.

Note the column constraint for the **Grade** column, the table constraint used to specify the composite primary key, and the referential integrity constraint enforced through the **FOREIGN KEY** clauses. This provides an alternative form of the **Foreign Key** enforcement to delete related rows if the related **STUDENT** or **SECTION** rows are deleted.

Also note the use of the **ON DELETE CASCADE** clause.

```
CREATE TABLE Enroll (

    SSN                 CHAR(9),

    Section_Number      NUMBER(6),

    Enroll_Term         CHAR(6),

    Enroll_Year         NUMBER(4),

    Grade               CHAR(2)

        CONSTRAINT Grade_Check_CK

            CHECK (Grade IN

                ('A','B','C','D','E','WP')),
    CONSTRAINT Enr_Stu_SSN_FK FOREIGN KEY (SSN)
```

```
        REFERENCES Student
            ON DELETE CASCADE,
    CONSTRAINT Enr_Section_Number_FK
        FOREIGN KEY (Section_Number,
            Enroll_Term, Enroll_Year)
        REFERENCES Section
            ON DELETE CASCADE,
    CONSTRAINT Enroll_PK
        PRIMARY KEY (SSN, Section_Number,
            Enroll_Term, Enroll_Year)
        USING INDEX Tablespace Index01
        PCTFREE 5
    )
PCTFREE 30 PCTUSED 65
Tablespace Data01;
```

Next we insert both valid and invalid rows in order to test the various constraints for the **ENROLL** table.   The first **INSERT** command inserts a valid row with a **Null** grade value. Note that the system allows a **Null** value since **Not Null** was not specified as a constraint for the **Grade** column.

```
INSERT INTO Enroll
    VALUES ('111111111', 111111, 'Summer', 2012, NULL);
```

The next **INSERT** command attempts to insert a row that violates referential integrity to the **STUDENT** table because a student with SSN=999-99-9999 is not stored in the **STUDENT** table.

```
INSERT INTO Enroll
    VALUES ('999999999', 111111, 'Summer', 2012, 'A');

ERROR at line 1:

ORA-02291: integrity constraint (DBOCK.ENR_STU_SSN_FK)
violated - parent key not found
```

The next **INSERT** command attempts to insert a row that violates referential integrity to the **SECTION** table.

```
INSERT INTO Enroll
    VALUES ('222222222', 999999, 'Summer', 2012, 'A');

ERROR at line 1:

ORA-02291: integrity constraint
(DBOCK.ENR_SECTION_NUMBER_FK) violated - parent key not
found
```

Attempt to insert a row that violates the **Check Constraint** for the **Grade** column. The row is rejected because the letter grade **'G'** is not in the acceptable list of values.

```
INSERT INTO Enroll
    VALUES ('222222222', 111111, 'Summer', 2012, 'G');


ERROR at line 1:
```

```
ORA-02290: check constraint (DBOCK.GRADE_CHECK_CK)
violated
```

Virtual Column

New to Oracle 11g is the ability to specify a **virtual column**.

- Treat them much as other columns.
- Not stored on disk - the value is derived on demand by computing it as a set of expressions or functions.
- Can be used in queries, DML, and DDL.
- Can be indexed - this is equivalent to a function-based index.
- Can have statistics collected on them.
- If the data type is not specified, Oracle will determine a data type based on the underlying expressions
- Uses the keywords **GENERATED ALWAYS.**
- The **AS *column_expression*** clause determines the column content.
- Cannot use the **SET** clause of an **UPDATE** statement to specify a value.
- Can be used in the **WHERE** clause of a **SELECT**, **UPDATE** or **DELETE** statement.

We cannot create these in Oracle 10g; however, a column definition for an Oracle 11g database would be like this example:

```
    Account_Balance        NUMBER(7,2),
    Monthly_Due            NUMBER(6,2)
        GENERATED ALWAYS AS (Account_Balance/12),
```

Using the Data Dictionary

The queries shown in this section are used to display information about the tables and indexes that were created by the CREATE TABLE commands. We also verify the existence of the specified constraints.

```
COLUMN Table_name FORMAT A12;
COLUMN Tablespace_name FORMAT A15;
SELECT Table_name, Tablespace_name
    FROM User_tables
    WHERE Table_name IN ('STUDENT', 'COURSE',
'SECTION', 'ENROLL');
```

```
TABLE_NAME   TABLESPACE_NAME
------------ ---------------
COURSE       DATA01
ENROLL       DATA01
SECTION      DATA01
```

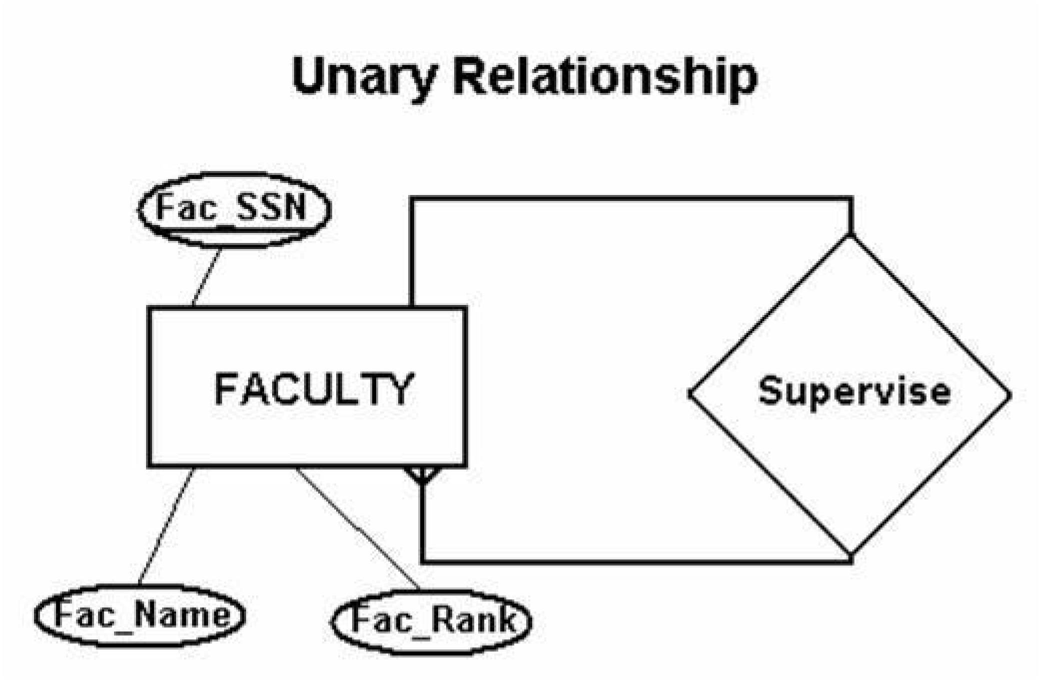
```
STUDENT      DATA01


COLUMN Index_name FORMAT A12;

SELECT Index_name, Table_name, Tablespace_name
    FROM User_indexes

    WHERE Table_name IN ('STUDENT', 'COURSE',

        'SECTION', 'ENROLL');


INDEX_NAME   TABLE_NAME   TABLESPACE_NAME

------------ ------------ ---------------

COURSE_PK    COURSE       INDEX01

STUDENT_PK   STUDENT      INDEX01

SECTION_PK   SECTION      INDEX01

ENROLL_PK    ENROLL       INDEX01


COLUMN Constraint_name FORMAT A21;

COLUMN CT FORMAT A2;

COLUMN Search_condition FORMAT A37;

SELECT Constraint_name, Constraint_type CT, Table_Name,
Search_Condition
    FROM User_constraints

    WHERE Table_name IN ('STUDENT', 'COURSE',

        'SECTION', 'ENROLL');
```

```
CONSTRAINT_NAME        CT TABLE_NAME   SEARCH_CONDITION
---------------------- -- ------------ -----------------
STUDENT_NAME_NN        C  STUDENT      "STUDENT_NAME" IS
                                        NOT NULL
LOCATION_NN            C  SECTION      "LOCATION" IS NOT
                                       NULL
YEAR_NN                C  SECTION      "SECTION_YEAR" IS
                                       NOT NULL
TERM_NN                C  SECTION      "SECTION_TERM" IS
                                       NOT NULL
GRADE_CHECK_CK         C  ENROLL       Grade IN
                                  ('A','B','C','D','E','WP')
HOURS_GREATER_ZERO     C  COURSE       Hours > 0
HOURS_LESS_12_CK       C  COURSE       Hours < 12
DESCRIPTION_NN         C  COURSE       "DESCRIPTION" IS
                                       NOT NULL
STUDENT_PK             P  STUDENT
SECTION_PK             P  SECTION
ENROLL_PK              P  ENROLL
COURSE_PK              P  COURSE
SECTION_TO_COURSE_FK   R  SECTION
ENR_SECTION_NUMBER_FK  R  ENROLL
```

```
ENR_STU_SSN_FK         R   ENROLL


15 rows selected.
```

Unary Relationships

A **unary** relationship (also called a **recursive** relationship) occurs when the rows within an individual table are related to other rows within the table. For example, a database table for **FACULTY** can store the same type of information for each faculty member. The chairperson of an academic department is both a faculty member and a supervisor of other faculty members so the **Supervisor** relationship would represent an association among faculty members. This relationship is depicted below.



The **CREATE TABLE** command shown below creates the **FACULTY** table. The foreign key **Fac_Supervisor_FK** implements the one-to-many relationship among the **FACULTY** rows.

The **Supervise** relationship is implemented through use of referential integrity constraints named **FK_Fac_Supervisor**. Also note that if a supervisor is deleted, the value for the cascading change to supervised faculty is set to **Null** through the **ON DELETE** clause of the foreign key.

```
CREATE TABLE Faculty (
    Fac_SSN             CHAR(9),
    First_Name          VARCHAR(25)
        CONSTRAINT First_Name_NN NOT NULL,
    Last_Name           VARCHAR(25)
        CONSTRAINT Last_Name_NN NOT NULL,
    Fac_Dept            VARCHAR(12),
    Fac_Supervisor_SSN  CHAR(9),
    CONSTRAINT Faculty_PK
        PRIMARY KEY (Fac_SSN)
        USING INDEX Tablespace Index01
        PCTFREE 5,
    CONSTRAINT Fac_Supervisor_FK
        FOREIGN KEY (Fac_Supervisor_SSN)
            REFERENCES Faculty
            ON DELETE SET NULL
    )
```

```
PCTFREE 15 PCTUSED 65

Tablespace Data01;
```

The **INSERT** commands shown here insert **good** rows into the **FACULTY** table. Note that the Supervisor of the Department (Bock) has a **NULL** value for the **Fac_Supervisor_SSN** column.

```
INSERT INTO Faculty
    VALUES ('123456789', 'Douglas', 'Bock', 'CMIS',
NULL);
INSERT INTO Faculty
    VALUES ('234567890', 'Susan', 'Yager', 'CMIS',
'123456789');
INSERT INTO Faculty
    VALUES ('345678901', 'Anne', 'Powell', 'CMIS',
'123456789');
```

Select rows from the **FACULTY** table. Note the value for the **Fac_Supervisor_SSN** column (labeled FAC_SUPER below).

```
COLUMN First_Name FORMAT a10;
COLUMN Last_Name FORMAT a10;
SELECT * FROM Faculty;
```

```
FAC_SSN   FIRST_NAME LAST_NAME  FAC_DEPT     FAC_SUPER

--------- ---------- ---------- ------------ ---------

123456789 Douglas    Bock       CMIS

234567890 Susan      Yager      CMIS         123456789

345678901 Anne       Powell     CMIS         123456789
```

Now we test the referential integrity constraint by attempting to store a row that violates the referential integrity results in an error message from Oracle.

```
INSERT INTO Faculty
    VALUES
('456789012','Joe','Prof','CMIS','444444444');

ERROR at line 1:

ORA-02291: integrity constraint
(DBOCK.FAC_SUPERVISOR_FK) violated - parent key not
found
```

We next test the **ON DELETE** cascading constraint by deleting Bock's record. This should reset the **Fac_Supervisor_SSN** values for the subordinate faculty rows to a **Null** value.

```
DELETE FROM Faculty WHERE Fac_SSN='123456789';

1 row deleted.


SELECT * FROM Faculty;


FAC_SSN   FIRST_NAME LAST_NAME  FAC_DEPT     FAC_SUPER

--------- ---------- ---------- ------------ ---------

234567890 Susan      Yager      CMIS

345678901 Anne       Powell     CMIS
```

Temporary Tables

Temporary tables are created to store session-private data that exists only while a transaction is processing or while a session exists.

- The tables are only visible to the individual system user creating them.
- Useful approach if temporary tables are needed to support special programming requirements.

The **CREATE GLOBAL TEMPORARY TABLE** command shown here has **ON COMMIT DELETE ROWS** to specify that rows will only be visible within a transaction. Alternatively, the **ON COMMIT PRESERVE ROWS** will cause rows to be visible for an entire session.

You can also create indexes, views, and triggers for temporary tables.

```
CREATE GLOBAL TEMPORARY TABLE admin_work_area (
  Start_date      DATE,
  End_date        DATE,
  Class_Desc      CHAR(20))
      ON COMMIT DELETE ROWS;
```

```
CREATE GLOBAL TEMPORARY TABLE department_temp

    ON COMMIT PRESERVE ROWS

    AS SELECT * FROM dbock.department;
```

- Rows of a temporary table are stored to the default temporary tablespace of the user creating the table.

- The TABLESPACE clause can be used to specify another tablespace for row storage -- useful when the default temporary tablespace has large extents meant for sorting.

- Segments to temporary tables and their index are not allocated until the first INSERT takes place.

- Backup/recovery of a temporary table is not available in the case of a system failure - the data is after all temporary.

Index-Organized Tables

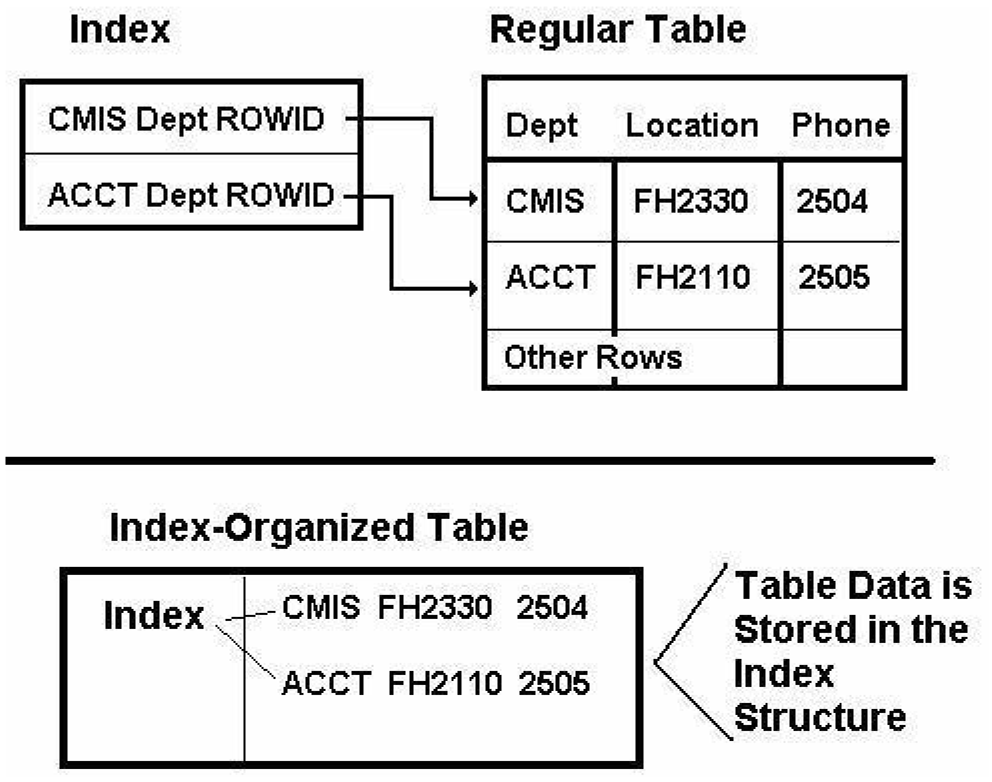
An **Index-Organized Table (IOT)** has a storage organization that is a variant of a primary B-tree structure.

- The figure shown here illustrates the difference in organization between a regular table and an IOT.

- Instead of storing each row's ROWID value in an index, the non-key column values are stored.

- Each B-tree entry for an IOT has the following structure (PK=Primary Key):

[PK column value, Non-PK column value1, Non-PK column value2, …]



- Programming applications access and modify data stored in an IOT in the same fashion as is done for a regular table – through the use of SQL and PL/SQL commands.

- The Oracle Server performs all of the operations required to maintain the structure of the B-tree index structure that comprises the IOT.

- This table compares regular table and IOT characteristics.

Ordinary Table	**Index-Organized Table**
Rowid uniquely identifies a row. Primary key can be optionally specified	Primary key uniquely identifies a row. Primary key must be specified

Ordinary Table	**Index-Organized Table**
Physical rowid in ROWID pseudocolumn allows building secondary indexes	Logical rowid in ROWID pseudocolumn allows building secondary indexes
Access is based on rowid	Access is based on logical rowid
Sequential scan returns all rows	Full-index scan returns all rows
Can be stored in a cluster with other tables	Cannot be stored in a cluster
Can contain a column of the LONG datatype and columns of LOB datatypes	Can contain LOB columns but not LONG columns

Benefits of IOT

IOTs improve database performance by providing faster access to table rows by the primary key or any key that is a valid prefix of the primary key.

- The storage of non-key columns for a row in the B-tree leaf block itself avoids an additional block access.
- Because rows are stored in primary key order, range access by the primary key (or a valid prefix) involves minimum block accesses.

In order to allow even faster access to frequently accessed columns, you can use a row overflow storage option (as described below) to push out infrequently accessed non-key columns from the B-tree leaf block to an optional (heap-organized) overflow storage area.

- This allows limiting the size and content of the portion of a row that is actually stored in the B-tree leaf block.
- This can result in a higher number of rows in each leaf block and a smaller B-tree structure.

Some storage space is saved because primary key columns are only stored once in the IOT where with regular tables, the primary key column values are stored in the table and in the associated index.

- Because rows are stored in primary key order as opposed to the unordered regular table, some storage space can be saved by using key compression

- This is specified with the **COMPRESS** clause—Oracle Server handles the key compression.

Primary key access for IOTs is based on logical rowids (the key column values) where regular tables are accessed by physical rowed values as you saw earlier. This approach also applies to secondary indexes.

IOT Row Overflow

As you will see when we study B-tree indexes later in this course, the index nodes for indexes that access regular tables are usually **very small** because they include just the **logical key value** and a **ROWID** that points to the row location in a regular table.

With IOT, the B-tree index nodes can become quite **large** because they store the entire row contents, especially where rows are 3000 characters or more in size.

- This can adversely affect performance of this index approach.

- In order to work around this potential imitation the Oracle Server can store rows by using an overflow tablespace or area.

- Columns that are accessed quite often are stored within the original IOT structure.

- Columns that are infrequently accessed can be stored in an overflow area. This enables the B-tree nodes to be smaller – more entries (rows) will fit within a single block and retrieval performance for data will improve.

The Oracle Server will automatically determine if a row should be stored in two parts (in the index and in overflow). You specify two clauses: **PCTTHRESHOLD** and **INCLUDING** to control the partitioning of a row.

- **PCTTHRESHOLD** specifies a threshold value as a percentage of block size.
    - If all non-key column values fit within the specified size limit, the row is not broken into two parts.
    - Otherwise, the first non-key column that will not fit within the threshold and all remaining non-key columns are moved to an overflow tablespace.

- The **INCLUDING** clause lets the DBA specify the column name within the CREATE TABLE statement where the break for overflow is to be made.

The CREATE TABLE Command for an IOT

When you execute a **CREATE TABLE** command to create an IOT, additional information must be specified in the command.

ORGANIZATION INDEX clause: This indicates that this is an IOT.

PRIMARY KEY clause: A primary key must be specified for an IOT.

OVERFLOW and **PCTTHRESHOLD clauses**: These specify the overflow tablespace and the PCTTHRESHOLD value that defines the percentage of space reserved in the index block for the IOT and the portion of a row that should be moved to overflow. Thus, the index entry contains the **key value**, the **non-key column values** that **fit** the specified threshold, and a **pointer** to the rest of the row.

Example #1: CREATE TABLE command for an IOT: he

```
CREATE TABLE states_iot (
    State_ID          CHAR(2)
        CONSTRAINT State_ID_PK PRIMARY KEY,
    State_Name        VARCHAR2(50),
    Population        NUMBER )
  ORGANIZATION INDEX
  TABLESPACE Data01
  PCTTHRESHOLD 20
  OVERFLOW TABLESPACE Data01;
```

Example #2 (from the Oracle Database Administrator's Guide):

- This example specifies an index-organized table, where the key columns and non-key columns reside in an index defined on columns that designate the primary key **(token, doc_id)** for the table.
- The break for data columns moved to overflow is specified as the **token_frequency** column by the **INCLUDING** clause.

- Primary key compression will be done by the Oracle Server because the **COMPRESS** clause is specified.

```
CREATE TABLE admin_doc_iot(
    token           char(20),
    doc_id          NUMBER,
    token_frequency NUMBER,
    token_offsets   VARCHAR2(512),
        CONSTRAINT admin_doc_iot_PK
            PRIMARY KEY (token, doc_id))
  ORGANIZATION INDEX COMPRESS
  TABLESPACE Data01
  PCTTHRESHOLD 20
  INCLUDING token_frequency
  OVERFLOW TABLESPACE Data01;
```

Parallelizing Table Creation

Parallel execution enables faster creation of tables. The **CREATE TABLE . . . . AS SELECT** statement has two parts:

- **CREATE** (this is DDL)
- **SELECT** (this is a query.
- Oracle can parallelize both parts if a **PARALLEL** clause is specified in the CREATE statement.

```
CREATE TABLE Test_Parallel

    PARALLEL COMPRESS

    AS SELECT * FROM PRODUCT

    WHERE Retail_Price > 200;
```

```
CREATE TABLE Test_Parallel
  PARALLEL
  AS SELECT * FROM employees
  WHERE salary > 6000;
```

Indexed Clusters

General Concepts

Clusters store data from different tables in the same physical data blocks.

- This approach can improve performance when application processing requires that records from related tables be frequently accessed simultaneously.
- Example: Suppose that order processing requires the application program to access frequently a row from a Customer **Order** table and the associated **Order_Details** table rows.
    - The **Order** table is referred to as the "**owner**" table in the relationship.
    - The **Order_Details** table is the "**member**" table in the relationship.

While it is possible to create a cluster with more than two tables in the cluster, this is only done when there is a **single** "owner" table and **multiple** "member" table types. Even then, the clustering of more than two tables is rare because of the potential negative effects on performance for other queries.

The use of a cluster **decreases** the number of database block reads that are necessary to retrieve the needed data for the cluster.

- This improves performance for queries that require data stored in both (or more than two if that is the case) tables in the cluster.

- While clustered retrieval performance for all tables in the cluster may improve, performance for other types of transactions may suffer.

- This is especially true for transactions that **only require one** of the tables in the cluster because all data within the data blocks will be transferred from disk to memory for these transactions even though the rows from one of the tables are not needed.

- Further performance gains can be obtained because the distinct value (**cluster key**) in each cluster column is stored only once, regardless of the number of times that it occurs in the tables and rows; thereby, reducing the storage space requirements.

Cluster Key and Index

The columns within a cluster index are often also used as the ***cluster key***. This is a set of **one or more columns** that both tables in the cluster have in common.

- The cluster key must be columns of the same size and data type in both tables, but need not be the same column name.

- Typically, the cluster key is the **foreign key** from the **member** table that references the **primary key** from the **owner** table in the cluster. The cluster key may also be part of a concatenated primary key in the member table.

For **Indexed Clusters** (as opposed to Hash Clusters), clusters must be indexed on the cluster key.

- This key must be created prior to the storage of any data in the cluster.

- There is one exception. If you specify a **Hash Cluster** form, you don't need to create an index on the cluster key. Instead, Oracle will use a hash function to store table rows.

When you query tables in a cluster, the row data in a result table continues to be displayed as if all of the data stored for the key values are stored in separate tables even though cluster keys are only stored once for a data block. In other words, the system user can't tell that the tables are clustered.

The maximum length of all cluster columns combined is 239 characters and tables with **LONG** columns **cannot** be clustered.

Steps in Creating an Indexed Cluster

The steps in creating a cluster are:

(1) create the cluster definition.

(2) create the cluster index definition.

(3) create the tables that form the cluster by defining them.

(4) load data into the tables.

As rows are inserted into the tables that form a cluster, the database stores the cluster key and its associated rows in each cluster database block.

In this example a cluster of two tables will be created: **TestOrders** and **TestOrderDetails**. These two tables have the following fields and characteristics.

Table	FieldName	FieldType/Size/Characteristics
TestOrders	OrderId	Number(3) Primary Key
	OrderDate	Date NOT NULL
	OrderTotal	Number(10,2)
TestOrderDetails	ProductId	Number(5) Primary Key
	OrderId	Number(3) Primary Key
	QuantityOrdered	Number(3)
	ItemPrice	Number(10,2)

The cluster index will be on the **OrderId** field since this column is in common between the two tables. This column also serves as the **Primary Key** of **TestOrders**, the "**owner**" table in this relationship.

The **PCTFREE** and **PCTUSED** parameters are set during creation of the cluster with the **CREATE CLUSTER** command. **Do not** set values for the **PCTFREE** and **PCTUSED** parameters for tables that form a cluster - if you set them in the **CREATE TABLE** command, they will be ignored. Also, clustered tables **always use** the **PCTFREE** and **PCTUSED** settings of the cluster.

The **CREATE CLUSTER** command has an optional argument, **SIZE**. This parameter is the estimated number of bytes required by an average cluster key and associated rows.

Oracle uses **SIZE** to estimate the number of cluster keys (and rows) that fit into a clustered data block; and to limit the number of cluster keys placed in a clustered data block.

If all rows for a unique cluster key will not fit into a single data block, extra data blocks are chained to the first block to improve speed of access to all values for a given cluster key.

By default only a single cluster key and associated rows are stored in each data block of a cluster's data segment if you do not specify the **SIZE** parameter.

The cluster is named **OrderCluster** in the **CREATE CLUSTER** command shown here.

```
CREATE CLUSTER OrderCluster

    ( OrderId NUMBER(3) )

    PCTUSED 60

    PCTFREE 40

    SIZE 1200

    Tablespace Data01;
```

Oracle responds with the **Cluster created** message if no errors occur during the process.

In step 2, the cluster index is created. This step is skipped for a hashed cluster. Note that the keyword **UNIQUE** is **not** allowed with a cluster index.

```
CREATE INDEX OrderClusterIndex

    ON CLUSTER OrderCluster

        INITRANS 2
        MAXTRANS 5
    Tablespace Index01;
```

Oracle responds with the **Index created** message if no errors occur during the process.

In step 3, the index cluster tables are created.

- Note that you cannot specify a **TABLESPACE** clause for a table that is part of a cluster because that option was specified when the cluster was created.

- Attempting to specify a tablespace will return an Oracle error code.

```
CREATE TABLE TestOrders (
    OrderId           NUMBER(3)
      CONSTRAINT TestOrders_PK Primary Key
      USING INDEX Tablespace Index01,
    OrderDate         DATE
      CONSTRAINT OrderDate_NN NOT NULL,
    Order_Amount       NUMBER(10,2) )
  CLUSTER OrderCluster (OrderId) ;

CREATE TABLE TestOrderDetails (
    ProductId          NUMBER(5),
    OrderId            NUMBER(3),
    Quantity_Ordered   NUMBER(3)
      CONSTRAINT Quantity_Ordered_CK
        CHECK (Quantity_Ordered >= 0),
    ItemPrice          NUMBER(10,2),
  CONSTRAINT TestOrderDetails_FK FOREIGN KEY (OrderId)
      REFERENCES TestOrders
      ON DELETE CASCADE,
  CONSTRAINT TestOrderDetails_PK
      PRIMARY KEY (ProductId, OrderId)
      USING INDEX Tablespace Index01 )
  CLUSTER OrderCluster (OrderId) ;
```

In each case, Oracle responds with the **Table created** message if no errors occur during the process.

In step 4, data rows are inserted into the tables. In this first example, valid data rows are entered for the **TestOrders** table and the associated **TestOrderDetails** records for a single order with **OrderId = 111** for two items that total to $75.95. The **ProductId** values for the items are "**55555**" and "**66666**".

```
INSERT INTO TestOrders
    VALUES (111,'23-Jun-05',75.95);
INSERT INTO TestOrderDetails
    VALUES (55555,111,1,50.00);
```

```
INSERT INTO TestOrderDetails
    VALUES (66666,111,1,25.95);
```

The data are listed from the tables to verify the existence of the rows

```
SELECT * from TESTORDERS;

   ORDERID ORDERDATE ORDER_AMOUNT
---------- --------- ------------
       111 23-JUN-05        75.95

SELECT * from TESTORDERDETAILS;

PRODUCTID    ORDERID QUANTITY_ORDERED  ITEMPRICE
---------- ---------- ---------------- ----------
     55555        111                1         50
     66666        111                1      25.95
```

Next we test the referential integrity constraints by attempt to insert a row in **TestOrderDetails** for which there is no corresponding **TestOrders** row. Note that **OrderId** "**222**" does not yet exist in the **TestOrders** table.

```
INSERT INTO TestOrderDetails
    VALUES (66666,222,1,25.95);
```

```
ORA-02291: integrity constraint
(DBOCK.TESTORDERDETAILS_FK) violated - parent key not
found
```

Potential Referential Integrity Errors

It is possible for an application developer or DBA to introduce errors into the system without recognizing the problem. For example, if the **TestOrderDetails** table was created without the **FOREIGN KEY** clause, then the system would be perfectly happy allowing entry of a row in **TestOrderDetails** that has no corresponding "**owner**" row in the **TestOrders** table.

This error can be demonstrated by dropping the **TestOrderDetails** table, then recreating it without the **FOREIGN KEY** clause. The attempted row insertion will now produce the following results:

```
INSERT INTO TestOrderDetails

    VALUES (66666,222,1,25.95);
```

```
Oracle responded with: 1 row created.
```

Following this, the **TestOrderDetails** table was again dropped and created using the original correct **CREATE TABLE** specification including the **FOREIGN KEY** clause. Data were again inserted into the table successfully.

Deleting Cluster Rows

If you examine the **FOREIGN KEY** clause for the **TestOrderDetails** table, you will find the specification of **CASCADE DELETE**. The **CASCADE DELETE** option specifies that if a row in **TestOrders** is deleted, then associated rows for the **TestOrderDetails** table will also be deleted.

This **DELETE** command example shows the results of deleting a row in **TestOrders** and then selecting rows from **TestOrderDetails**. Note that the **CASCADE DELETE** option took effect.

```
DELETE FROM TestOrders

    WHERE Orderid = '111';

Oracle responds with:  1 row deleted.
```

Now we query the database.

```
SQL> SELECT * FROM TestOrderDetails;

Oracle responds with:  no rows selected
```

Cluster Information in the Data Dictionary

Querying the data dictionary **User_Clusters** view will provide information about the cluster that was created in these notes.

```
SELECT Cluster_Name, TableSpace_Name
    FROM User_Clusters;

CLUSTER_NAME    TABLESPACE_NAME
------------    ---------------
ORDERCLUSTER    Data01
```

Hashed Clusters

General Concepts

Hash clusters can improve data retrieval performance.

- Indexed tables or indexed clusters have rows located by using Key Value-RowID pairs through indexes.
- Hashed clusters store and retrieve rows through a **hash function**.
- The hash function distributes hash values (based on cluster key values) evenly throughout the storage area.
- The hash value corresponds to a data block in the cluster.
- **NO Input/Output** is required to locate a row in a hash cluster.

When to Use/Not Use Hash Clusters

Use hashing when:

- Queries are **equality queries** – the row with the key is found in one I/O compared to indexed approaches that require one or more reads of the index and another I/O read of the data block containing the row.

  Example:

  ```
  SELECT column1, column2, . . . FROM table_name WHERE cluster_key = ' . . . '
  ```

- The tables in the hash cluster are static – extending a hash cluster beyond the initial disk space allocation can cause degradation of performance due to the use of overflow storage space.

Do NOT use hashing when:

- Queries retrieve rows over a range of values. A hash function cannot determine the row locations and a full table scan often results.

  Example:

  ```
  SELECT column1, column2, … FROM table_name WHERE cluster_key < ' . . . '
  ```

- Tables continue to grow over the life of the database.

- Applications tend to require full-table scans as part of normal data processing – hashing performs extremely poorly in this situation.

- There is not enough available disk space to allocate the space needed for the hash cluster in advance.

Creating a Hash Cluster

A hash cluster is created through use of the **HASHKEYS** clause.

```
CREATE CLUSTER HashOrderCluster
    ( OrderId NUMBER(3) )
    PCTUSED 60
    PCTFREE 40
    SIZE 1200
    Tablespace usrers
  HASH IS OrderID HASHKEYS 150;

CREATE TABLE HashTestOrders (
    OrderId           NUMBER(3)
      CONSTRAINT HashTestOrders_PK Primary Key,
    OrderDate         DATE
      CONSTRAINT HashOrderDate_NN NOT NULL,
    Order_Amount       NUMBER(10,2) )
  CLUSTER HashOrderCluster (OrderId) ;
```

```
CREATE TABLE HashTestOrderDetails (
    ProductId          NUMBER(5),
    OrderId            NUMBER(3),
    Quantity_Ordered   NUMBER(3)
      CONSTRAINT HashQuantity_Ordered_CK
        CHECK (Quantity_Ordered >= 0),
    ItemPrice          NUMBER(10,2),
  CONSTRAINT HashTestOrderDetails_FK
      FOREIGN KEY (OrderId)
      REFERENCES TestOrders
      ON DELETE CASCADE,
  CONSTRAINT HashTestOrderDetails_PK
      PRIMARY KEY (ProductId, OrderId) )
  CLUSTER HashOrderCluster (OrderId) ;
```

- The primary key (**hash key**) can be one or more columns – the above example has a single column named **OrderID**.
- The **HASHKEYS** clause value specifies and limits how many unique hash values can be generated – the actual value is rounded to the nearest prime number by Oracle.
- The **HASH IS** clause is optional.

- It can be used to specify the cluster key; however, this is only necessary if the intent is **NOT** to hash on the primary key or if the primary key is of data type **NUMBER** and is unique already so that the internal hash function will be sure to distribute the rows uniformly among the disk space allocated to the cluster.

- Specify the **HASH IS** parameter only if the cluster key is a single column of the **NUMBER** data type.

- This parameter can also be used to specify a user-defined hash function to use instead of the function provided by default by Oracle.

- In this example a cluster is created that has a **CHAR** data type primary key and no **HASH IS** clause is specified so the cluster will be hashed on the **Prescription_Drug_Code** column value.

```
CREATE CLUSTER Prescription_Drug_Cluster
     (Prescription_Drug_Code CHAR(5))
     SIZE 512 SINGLE TABLE HASHKEYS 500;
```

- Hash clusters cannot be indexed.

- Hash clusters should have a hash key that reflects how **SELECT** statement **WHERE** clauses will query the database, but most often the cluster is hashed on the primary key.

Sorted Hash Cluster

Hash clusters with related rows (same hash key or part of hash key) that need to be returned in sorted order can be sorted.

- Example (from Oracle Database Administrator's Guide) – this cluster has three columns that form the **hash key** (**telephone_number**, **call_timestamp**, and **call_duration**). The entries are sorted by **call_timestamp** and **call_duration**.

```
CREATE CLUSTER call_detail_cluster (
   telephone_number NUMBER,
   call_timestamp NUMBER SORT,
   call_duration NUMBER SORT )
  HASHKEYS 10000 HASH IS telephone_number
  SIZE 256;

CREATE TABLE call_detail (
   telephone_number     NUMBER,
   call_timestamp       NUMBER   SORT,
   call_duration        NUMBER   SORT,
   other_info           VARCHAR2(30) )
  CLUSTER call_detail_cluster (
   telephone_number, call_timestamp, call_duration );
```

- The query shown here returns call records for the hash key by oldest record first.

```
SELECT * FROM call_detail

    WHERE telephone_number = 6505551212;
```

Single Table Hash Cluster

A single-table hash cluster provides very fast access to individual table rows.

- The hash cluster here can contain only one table.
- There is a 1-to-1 mapping between hash keys and data rows.
- Note use of the **SINGLE TABLE** clause.

- The **HASHKEYS** value here rounds up to the nearest prime number (**503**) that specifies the maximum number of possible hash key values, each of size **512** bytes.

Example:

```
CREATE CLUSTER Physician_Specialty_Cluster (Specialty_Code
NUMBER)

    SIZE 512 SINGLE TABLE HASHKEYS 500;
```

```
CREATE TABLE Physician_Specialty (
   Specialty_Code       NUMBER,
   Specialty_Description VARCHAR(75) )
  CLUSTER Physician_Specialty_Cluster (Specialty_Code);
```

Controlling Hash Cluster Disk Space Use

Choose the cluster key based on the most common type of query to be issued against the cluster tables.

- Example: An **EMPLOYEE** table may have the **EmployeeID** column as the cluster key.

- Example: A **DEPARTMENT** table may have the **DepartmentNumber** column as the cluster key.

The **HASH IS** clause parameter should **only be used** if the cluster key is a single column of **NUMBER** data type and if the column values are **uniformly distributed integers** – this facilitates allocating rows within the cluster without having collisions (two cluster key values that hash to the hash value).

The **SIZE** parameter should be set to the average disk space needed to store all rows for a given hash key (need to calculate average row size for all related rows in a cluster).

Additional Topics

ALTER TABLE Options

The **ALTER TABLE** command can be used to modify the structure of a table, to change storage and block utilization parameters, to add a referential integrity constraint, to remove a referential integrity constraint, and for several other modifications.

Adding a Column to a Table: This is an example of an **ALTER TABLE** command that adds two columns to the **DEPARTMENT** table.

```
ALTER TABLE Department ADD

    (Dept_Location   VARCHAR2(25),

     Dept_Division   VARCHAR2(50) DEFAULT 'Business' );
```

New columns initially store **NULL** values unless a **DEFAULT** clause is specified as was done for the **Dept_Division** column shown above (the default is 'Business').

New columns can only be specified as NOT NULL if the table does not contain any rows or if a DEFAULT value is specified for the column.

Changing Storage/Block Utilization Parameters: The ALTER TABLE command below demonstrates changing PCTFREE, PCTUSED, and various STORAGE clauses.

```
ALTER TABLE Department
    PCTFREE 30
    PCTUSED 50
    STORAGE (NEXT 256K PCTINCREASE 20
        MINEXTENTS 2 MAXEXTENTS 50);
```

- The **NEXT** parameter change takes effect the next time the Oracle Server allocates an extent for the **DEPARTMENT** table.

- The **PCTINCREASE** change will be used to recalculate NEXT when the next extent is allocated. If both NEXT and PCTINCREASE are modified, then the PCTINCREASE change goes into effect after the next extent is allocated.

    - Example: The **DEPARTMENT** table has two extents of size **128K**. The 3rd extent will be **256K** (as specified in the ALTER TABLE command shown above). The 4th extent will be approximately **307.2** (20% larger), but an extent must be a multiple of the block size (assuming a block size of **8K**) so this would be rounded up to **312K**.

- The **MINEXTENTS** clause can be modified to any value equal to or less than the current number of allocated extents and will only have an effect if the table is truncated.

- The **MAXEXTENTS** clause can be modified to any value equal to or greater than the current number of extents for the table. It can also be set to **UNLIMITED**.

- You'll notice that the **INITIAL** storage parameter is not listed – it cannot be modified for a table that already exists.

Add a Referential Integrity Constraint: The example ALTER TABLE command shown here adds a referential integrity constraint. You might not want the constraint added when the table is initially created, or you might be adding a constraint for a new table.

First a new table named **TEACHER** is created and a new row is inserted into the table.

```
/* Create Owner Table */

CREATE TABLE Teacher (

    Teacher_ID      CHAR(4)

        CONSTRAINT Teacher_PK PRIMARY KEY

            USING INDEX Tablespace Index01

            PCTFREE 5,

    Teacher_Name    VARCHAR(40)

        CONSTRAINT Teacher_Name_NN NOT NULL

    )

PCTFREE 5 PCTUSED 60

TABLESPACE DATA01;


INSERT INTO Teacher VALUES ('1234','Bock, Douglas');
```

Next the **SECTION** table has a column added, and then a foreign key constraint is added to the **SECTION** table to enforce referential integrity to the **TEACHER** table's **TEACHER_ID** column.

```
ALTER TABLE Section ADD

    (Teacher_ID CHAR(4) DEFAULT '1234');


ALTER TABLE Section
```

```
    ADD CONSTRAINT Section_To_Teacher_FK

    FOREIGN KEY (Teacher_ID) REFERENCES Teacher;
```

Now test the new foreign key constraint by trying to insert a new **SECTION** row with an invalid **TEACHER_ID** value.

```
INSERT INTO SECTION VALUES

    ('55555','Summer',2012,'CMIS460','FH-3208','2222')

ERROR at line 1:

ORA-02291: integrity constraint (DBOCK.SECTION_TO_TEACHER_FK)
violated - parent
```

Drop a Referential Integrity Constraint: The example **ALTER TABLE** command shown here drops the **SECTION_TO_TEACHER_FK** foreign key constraint for the **SECTION** table.

```
ALTER TABLE Section

    DROP CONSTRAINT Section_To_Teacher_FK;


Table altered.
```

Manually Allocating Extents: The example ALTER TABLE command shown here manually allocates an extent to the DEPARTMENT table in order to control the distribution of the table across files that comprise the tablespace where the table is stored.

```
ALTER TABLE department ALLOCATE EXTENT (SIZE 512K

  DATAFILE
  '/u01/student/dbockstd/oradata/USER350users01.dbf');
```

The manual allocation of extents does not affect the computation of the size of the NEXT extent where the PCTINCREASE clause is not zero.

Move a Table: A non-partitioned table can be moved from one tablespace to another in order to support a shift in application processing of some sort. This is also useful to eliminate row migration that may exist in a table.

```
ALTER TABLE department

    MOVE TABLESPACE Index01;
```

Moving a table reorganizes the extents and eliminates row migration. After moving a table, the DBA must rebuild the indexes for the table manually; otherwise, Oracle will return the **ORA-01502 index 'Index_name" or partition of such index is in unusable state** error message. Rebuilding Indexes is covered in a later module in this course.

Rename a Column: Sometimes a DBA will need to rename an existing column in order to correct some earlier error in following an organization's design guidelines. For example, a column named **Dept_Spvsr** might need to be renamed to a more meaningful column name, **Dept_Supervisor**.

```
ALTER TABLE department

    RENAME COLUMN Dept_Spvsr TO Dept_Supervisor;
```

The new name must obviously not duplicate the name of an existing column in the table. When a column is renamed, the Oracle Server updated associated data dictionary tables so that function-based indexes and CHECK constraints remain valid for the column.

<u>Drop a Column</u>: You can drop columns from table rows. This is done to rid the database of unused columns without having to export/import data and recreate indexes and constraints.

```
ALTER TABLE department

    DROP COLUMN dept_resources

    CASCADE CONSTRAINTS CHECKPOINT 500;
```

Dropping a column can be time-consuming as all data in the column is deleted from the table. As such, it is often useful to specify checkpoints to minimize the size of the undo space that is generated by the DROP procedure as was done with the CHECKPOINT clause in the ALTER TABLE command given above.

<u>The UNUSED Option</u>: A table can be marked as unused and then later dropped when a database's activity lessens. This is also useful if you want to drop two or three columns. If you drop two columns, the table is updated twice, once for each column. If you mark the two columns as UNUSED, then drop the columns, the rows are updated only one time.

```
#Mark column unused

ALTER TABLE department

    SET UNUSED COLUMN dept_resources

    CASCADE CONSTRAINTS;


#Drop unused columns

ALTER TABLE department

    DROP UNUSED COLUMNS CHECKPOINT 500;
```

Truncate a Table

When a DBA truncates a table, this action deletes all rows in a table and releases any allocated disk space. This also truncates indexes for the table.

```
TRUNCATE TABLE department;
```

Other effects of truncation include:

- No undo data is generated – the TRUNCATE command commits implicitly because TRUNCATE TABLE is a DDL command, not DML.
- A table referenced by a foreign key from another table cannot be truncated.
- Any triggers for deletion do not fire when a table is truncated.

Drop a Table

Tables are dropped when they are not needed any longer or when they are being reorganized. In the latter case, the contents of the table are typically exported, the table is dropped, then recreated, and the contents are imported to the new table structure.

```
DROP TABLE department
    CASCADE CONSTRAINTS;
```

The **CASCADE CONSTRAINTS** option is necessary if this is a parent table in a foreign key relationship.

Use the **ALTER TABLE. . .DROP COLUMN** command to drop a column(s) that is no longer needed.

Flashback Drop and the Recycle Bin

A table that is dropped does not have its space immediately recovered. Instead the table is placed in a recycle bin and the FLASHBACK TABLE command can restore the table.

The recycle bin is a data dictionary table with information on dropped objects (tables, indexes, constraints, etc).

Each user essentially has their own recycle bin because the only objects the user can access in the recycle bin are those the user owns (except a user with SYSDBA privilege can access any object).

Objects are named using the convention:

```
BIN$unique_id$version
```

- **Unique_ID** is a 26-character globally unique identifier.
- **Version** is the version number assigned by the database.

To view the recycle bin use a **SELECT** statement.

```
COLUMN Original_Name FORMAT A14;

SELECT Object_Name, Original_Name, DropTime

FROM RECYCLEBIN;



OBJECT_NAME                    ORIGINAL_NAME  DROPTIME

------------------------------ -------------- ------------------

BIN$whZwQFKyp0LgQKOSZvwM3g==$0 TEST_PARALLEL
2012-06-09:22:14:05

BIN$whZwQFKwp0LgQKOSZvwM3g==$0 STUDENT
2012-06-09:22:00:40

BIN$qHjqiFu2x4zgQKOSZvxkIw==$0 INVOICE
2011-07-19:22:34:56

...more rows display


12 rows selected.
```

- Dropping a tablespace does not result in the objects going in the recycle bin.
- Dropping a user - objects belonging to the user are not placed in the recycle bin and the bin is purged.
- Dropping a cluster - member tables are not placed in the recycle bin.

To disable the recycle bin, use one of these statements.

```
ALTER SESSION SET Recyclebin = OFF;

ALTER SYSTEM SET Recyclebin = OFF SCOPE = SPFILE;
```

To enable the recycle bin, use one of these statements.

```
ALTER SESSION SET Recyclebin = ON;

ALTER SYSTEM SET Recyclebin = ON SCOPE = SPFILE;
```

In both cases above, using ALTER SYSTEM requires a database restart.

To restore a table use the **FLASHBACK TABLE. . .TO BEFORE DROP** command. This sequence demonstrates using the command.

```
CREATE TABLE Test (
    Column1  VARCHAR2(15));
INSERT INTO Test VALUES ('ABC');
INSERT INTO TEST VALUES ('XYZ');
COMMIT;

Commit complete.
```

```
SELECT * FROM Test;

More...


COLUMN1

---------------

ABC

XYZ


DROP TABLE Test;

Table dropped.


SELECT * FROM Test;


ERROR at line 1:

ORA-00942: table or view does not exist


FLASHBACK TABLE Test TO BEFORE DROP RENAME TO Test2;

Flashback complete.


SELECT * FROM Test2;

More...
```

```
COLUMN1

---------------

ABC

XYZ
```

Place a Table in Read-Only Mode

A table can be read-only mode. An example is a table that contains data that you do not want to be modified--very static data.

```
ALTER TABLE States READ ONLY;
```

To return the table to read/write mode:

```
ALTER TABLE States READ WRITE;
```

Data Integrity

The DBA is always concerned with maintaining **Data Integrity**. This simply means that the data stored in the database meets the requirements of established **business rules** for the organization. Data integrity can be maintained by:

- **Application programs** – the code written can have integrity rules built-in that keep invalid data from being added to a database.

- **Database triggers** – these are PL/SQL programs that execute whenever an event, such as the insertion of a new table row or the modification of a value stored in a column occurs. Triggers are usually used to enforced complex business rules that are not easily enforced through defined integrity constraints

- **Integrity constraints** – you've already seen examples of integrity constraints in the CREATE TABLE commands shown earlier in these notes. This is the preferred way to enforce data integrity.

Constraint State: An integrity constraint can be enabled or disabled. An enabled constraint enforces data integrity. If data does not conform to the constraint, then new rows are not inserted and modified rows are rolled back.

An Integrity constraint can have the following states:

- **DISABLE NOVALIDATE**: Such a constraint is not checked so new data can be entered that does not conform to the constraint's rules.

- **DISABLE VALIDATE**: Modification of constrained columns is not allowed. The index on the constraint is dropped and the constraint is disabled.

- **ENABLE NOVALIDATE**: New data violating the constraint cannot be entered, but the table can contain invalid data that has already been entered – useful for uploading data warehouses from valid OLTP data.

- **ENABLE VALIDATE**: This is the enabled state for a constraint where data must conform to the constraint. A table moved to this state is locked and all data in the table is checked for validity.

A DBA may temporarily disable integrity constraints for a table in order to improve the performance of these operations, especially for large data warehouse configurations:

- Load large amounts of data.
- Perform batch operations that change a majority of the rows in a table, for example, updating each employee's salary as a result of a 3% raise.
- Import or export one table at a time.

Deferring Constraints: Checking constraints can be deferred until the end of a transaction – this is used most often for data loads of rows belonging to both parent and child tables such as would be the case for **SALES_ORDER** and **ORDER_DETAILS** tables.

Non-deferred constraints are enforced at the end of each DML statement. Violations cause rows to be rolled back.

Deferred constraints are checked when a transaction commits. Any violations detected when the transaction commits cause the entire transaction to roll back.

Example commands:

```
CREATE TABLE department (
    Dept_Number  NUMBER(5) PRIMARY KEY DISABLE, . . . ;

ALTER TABLE department
    ADD PRIMARY KEY (Dept_Number) DISABLE;

ALTER TABLE department
    ENABLE NOVALIDATE CONSTRAINT PK_Dept_Number;

ALTER TABLE employee
    ENABLE VALIDATE CONSTRAINT FK_Dept_Employee_No;

ALTER SESSION
```

```
     SET CONSTRAINTS DEFERRED;

ALTER SESSION
     SET CONSTRAINTS IMMEDIATE;
```

Additional Data Dictionary Information

The views used most often by a DBA to query table and constraint information are:

- DBA_TABLES
- DBA_OBJECTS
- DBA_CONSTRAINTS
- DBA_CONS_COLUMNS

```
COLUMN table_name FORMAT A17;

COLUMN tablespace_name FORMAT A15;

SELECT table_name, tablespace_name, pct_free, pct_used

FROM dba_tables

WHERE owner = 'DBOCK'

ORDER BY table_name;


TABLE_NAME        TABLESPACE_NAME   PCT_FREE   PCT_USED

----------------- --------------- ---------- ----------
```

```
BED                Data01                  10
BEDCLASSIFICATION  Data01                  10
DEPARTMENT         Data01                  10
DEPENDENT          Data01                  10
EMPLOYEE           Data01                  10
<more tables are listed>
```

```
COLUMN object_name FORMAT A15;
COLUMN object_type FORMAT A15;
SELECT object_name, object_type, created, last_ddl_time
FROM dba_objects
WHERE owner = 'DBOCK'
ORDER BY object_name;
```

```
OBJECT_NAME     OBJECT_TYPE     CREATED   LAST_DDL_
--------------- --------------- --------- ---------
<more objects are listed>
SPECIALTY       TABLE           14-JUN-09 14-JUN-09
TREATMENT       TABLE           15-JUN-09 15-JUN-09
UN_EMPLOYEEPARK INDEX           14-JUN-09 14-JUN-09
INGSPACE
```

END OF NOTES