module15-syn_seq_view.htm; updated June 16, 2012; Some figures shown in these notes are from Oracle document D11321GC11.

Module 15 – Synonyms, Sequences and Views

Objectives

Create and drop synonyms and public synonyms.

Create sequences.

Generate values from sequences and use them to store rows to data tables.

Create, alter, and drop views.

Create JOIN views.

Learn the advantages associated with views.

Synonyms

Facts about Synonyms

Synonyms are a form of database shorthand.

Synonyms allow the specification of a short reference name for a long, complex object name, e.g. **sys.dba_tablespaces** may be shortened to **tabspaces** by creating a synonym named **tabspaces**.

Synonyms allow access to other databases on other nodes of a distributed database network that is transparent to the system user.

Normally only **private synonyms** are created by system users – **public synonyms** are created by database administrators.

CREATE SYNONYM Command

The syntax for the CREATE SYNONYM command is:

- **select * from** `SYS.Dba_Tablespaces;`
- `CREATE [PUBLIC] SYNONYM Tabspaces FOR SYS.Dba_Tablespaces;`
- Select * from `Tabspaces;`

In order to create a private synonym you must have the **CREATE SYNONYM** privilege.

Normally, only the owner of an object (such as a table) can create a synonym.

Individuals with the **CREATE PUBLIC SYNONYM** privilege can also create synonyms for objects in other schemas. Example:

```
CREATE SYNONYM Eq FOR Dbock.Equipment;
```

```
Synonym created.
```

Public synonyms must be unique names – for this reason DBAs tend to discourage the creation of public synonyms. The **CREATE PUBLIC SYNONYM** privilege will usually not be granted to regular system users.

Altering and Dropping Synonyms

Synonyms cannot be altered

Use the **DROP** command to drop a synonym. Example:

```
DROP [PUBLIC] SYNONYM Eq;
```

Sequences

Facts About Sequences

Sequences are special database objects used to generate numbers in sequential order, typically to be stored as values in data rows for a data table.

Primary use: To generate **unique key values** for tables that can be used to link to other tables or that will serve as primary keys (sequence generated primary keys are termed surrogate keys).

Example: Use a **Order_Number_Sequence** for a **TestOrders** table where the value of the **Order_Number_Sequence** must be unique and in numerical sequence.

CREATE SEQUENCE Command

This example shows the syntax for the **CREATE SEQUENCE** command.

```
CREATE SEQUENCE Order_Number_Sequence
    INCREMENT BY 1
    START WITH 1
    MAXVALUE 99999999 (or NOMAXVALUE)
    MINVALUE 1 (or NOMINVALUE)
    CYCLE (or NOCYCLE - the default)
    CACHE n (or NOCACHE)
    ORDER (or NOORDER);
```

INCREMENT BY defaults to a value of 1.

- Specifying a positive number increment will generate ascending sequence numbers with an interval equal to the value you select.

- Specifying a negative number will generate descending sequence numbers.

START WITH is the starting number for the sequence.

- The default for **START WITH** is **MAXVALUE** for a descending sequence and **MINVALUE** for an ascending sequence.
- **START WITH** overrides the default value.

MINVALUE is the **lowest** sequence number generated. **MIN VALUE** and **START WITH** default to 1 for ascending sequences.

MAXVALUE is the **largest** sequence number generated.

- For descending sequences, it defaults to -1.
- To allow **unlimited sequence number generation** only use **MINVALUE** for ascending and **MAXVALUE** for descending sequences.
- Specifying a limit with **MINVALUE** or **MAXVALUE** will force an error when an attempt is made to generate a number exceeding the limit and when you have specified **NOCYCLE**.

CYCLE causes automatic cycling to start a sequence over once the **MAXVALUE** for an ascending sequence or **MINVALUE** for a descending sequence is reached. The default MAXVALUE is **10e27 - 1** (a very large number).

CACHE is an option to cache the specified number of sequence values into buffers in the SGA.

- This speeds access, but loses the cached numbers if the database is shut down.
- The default value for cached numbers is 20 if you do not specify **NOCACHE**.

ORDER forces sequence numbers to be output in order, and is usually used where the sequence number is used for time stamping.

Example Create Sequence:

```
CREATE SEQUENCE Order_Number_Sequence
    INCREMENT BY 1
    START WITH 1
    MAXVALUE 999
    MINVALUE 1
    CYCLE
    CACHE 5
    ORDER;
```

```
Sequence created.
```

Using Sequences

In order to use a sequence, you must first generate the initial sequence number by using the **NEXTVAL** option.

An example using NEXTVAL:

- Insert one row into **TestOrders**.

```
INSERT INTO TestOrders (OrderID, OrderDate,
Order_Amount)
    VALUES (Order_Number_Sequence.NEXTVAL, SYSDATE,
100.42 );
```

```
1 row created.
```

- Display the data rows from **TestOrders** – note the first data row was added during an earlier lab assignment. The second data row added today by using the sequence.

```
SELECT * FROM TestOrders;
```

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

```
         1 15-JUL-09       100.42
```

The **CURRVAL** option returns the **current sequence number**, but **will not execute** unless the sequence has been called at least one time using the **NEXTVAL** option.

CURRVAL is used instead of **NEXTVAL** to use the same sequence number more than once, for example, when you are inserting rows into a related table where referential integrity must be enforced. You can use CURRVAL as often as desired within a statement and the same number is returned.

An example using CURRVAL:

- Insert one row into **TestOrders**.

```
INSERT INTO TestOrderDetails (ProductID, OrderID,
Quantity_Ordered, ItemPrice)

    VALUES(14985, Order_Number_Sequence.CURRVAL, 2,
50.21);
```

```
1 row created.
```

- Display the data rows from **TestOrderDetails** – note the first two data rows were added during an earlier lab assignment. The third data row was added today by using the sequence.

```
SELECT * FROM TestOrderDetails;
```

```
PRODUCTID    ORDERID QUANTITY_ORDERED  ITEMPRICE
---------- ---------- ---------------- ----------
     55555        111                1         50
     66666        111                1      25.95
     14985          1                2      50.21
```

If you use **NEXTVAL** and **CURRVAL** in the same SQL statement, both of these values will be the value retrieved by **NEXTVAL**.

You cannot use **NEXTVAL** or **CURRVAL** in subqueries as columns in a SELECT clause where you use DISTINCT, UNION, INTERSECT, or MINUS or in ORDER BY, GROUP BY, or HAVING clauses.

How does the DBMS keep sequence numbers under control?:

- A sequence number generated is only available to the session that generated the number.
- Other users referencing the **Order_Number_Sequence.NEXTVAL** will obtain unique values.

- Concurrent access to a sequence will result in each user receiving different sequence numbers.
- Gaps may occur in sequence numbers received for use since other users may access the same sequence.

Within a single statement, a reference to **NEXTVAL** will first generate the next number from the sequence, and then all other references within the statement return the same sequence number.

Restrictions on using CURRVAL and NEXTVAL

CURRVAL and **NEXTVAL** **can be** used in the following places:

- VALUES clause of INSERT statements
- The SELECT list of a SELECT statement
- The SET clause of an UPDATE statement

CURRVAL and **NEXTVAL** **cannot be** used in these places:

- A subquery
- A view query or materialized view query
- A SELECT statement with the DISTINCT operator

- A SELECT statement with a GROUP BY or ORDER BY clause
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator
- The WHERE clause of a SELECT statement
- DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement
- The condition of a CHECK constraint

Caching Sequence Numbers

When sequence values are "read" they are stored to the sequence cache in the SGA. You want these values to be accessible quickly.

This example creates a sequence with a cache of 40 values

```
CREATE SEQUENCE Product_Number_Seq
    CACHE 40;
```

After these values are used, value #41 through #80 are next generated and stored.

Choose a high value for CACHE to result in fewer reads from disk to the sequence cache, recognizing that in an instance failure some sequence values are likely to be lost.

Altering and Dropping Sequences

Use the **ALTER SEQUENCE** command to alter a sequence.

- The syntax is like that shown for the **CREATE SEQUENCE** command.
- Most parameters may be altered, but only future sequence numbers are affected.
- You cannot alter the **START WITH** clause without dropping and recreating the sequence.

Use the **DROP SEQUENCE Sequence_Name** command to drop a sequence. Example:

```
DROP SEQUENCE Order_Number_Sequence;
```

```
Sequence dropped.
```

Caution: If you have created a trigger or procedure that references a sequence, the trigger/procedure will fail if the sequence is dropped.

Views

Facts about Views

Views are **virtual tables** – that is, they are a logical representation of a collection of columns from one or more (related) base tables or other views. Views are defined in a way such that they make it easier for a user to access needed information from a database.

For example, a salesperson may be accustomed to viewing sales orders according to the traditional sales order form that the firm uses.

- In most firms, the data required to produce a complete sales order requires data from the **CUSTOMERS**, **ORDERS**, **ORDER_DETAILS**, and **PRODUCTS** tables.

- Creating a view of the data from these tables provides the customer a single "object" with which to process information about a sales order.

- Views are selected (queried) just like a table.

As another example, consider the "tables" you access that comprise the data dictionary. Most of these are actually views of the data.

Advantages of Views

- Views don't exist until they are queried except as specification statements stored in the database's data dictionary. Views do NOT actually store any data.
- Views are efficient unless views are **stacked**, i.e. one view references another view - then performance problems can occur.
- Views allow the designer to **restrict access** to specific columns within a table or tables.
- Views can provide **derived column values** (**calculated fields**) that are not normally stored as part of tables, and display the derived column data along with actual data.
- Views can filter data.
- Depending on how views are constructed, inserts and deletions can be accomplished by using views, or data manipulation for views may be restricted to simply selecting data.

CREATE VIEW Command

The general CREATE VIEW syntax is shown here

```
CREATE VIEW ViewName
        (Column#1-Alias, Column#2-Alias, . . . )
```

```
AS
SELECT Table1.Column1, Table1.Column2,
       Table2.Column1, Table2.Column2, . . .
  FROM Table1, Table2, . . .
 WHERE Condition1, Condition2, . . . ;
```

The specification of **Column Alias Names** is optional. This is usually done in order to provide the system user with column names that are meaningful to them according to the terminology they use when performing their jobs.

View privileges:

- To create a view in your schema you need the **CREATE VIEW** system privilege.
- To create a view in another schema you need the **CREATE ANY VIEW** system privilege.
- Both of these privileges may be acquired through roles or can be granted to you explicitly.
- A view owner must have been granted explicit privileges to access all objects referenced in a view definition – these privileges cannot be obtained through roles.
- For a view owner to grant view access to others the owner must have received object privileges on the base objects with the **GRANT OPTION** or the system privileges with the **ADMIN OPTION**.

Example:

- **User1** has the **SELECT** and **INSERT** privileges for the **Employee** table owned by **User2**.
- Any view on the **Employee** table created by **User1** can only support **SELECT** and **INSERT** operations.
- **DELETE** and **UPDATE** operations are not supported.

Example Create View:

This example uses the **Product** table in the **DBOCK** schema. Here is a description of the table.

```
DESC Product;
```

```
Name                 Null?    Type
-------------------- -------- --------------
PRODUCT_NUMBER       NOT NULL CHAR(10)
DESCRIPTION          NOT NULL VARCHAR2(50)
RETAIL_PRICE                  NUMBER(8,2)
WHOLESALE_PRICE               NUMBER(8,2)
SALE_PRICE                    NUMBER(8,2)
```

A **SELECT** statement displays three of the columns from the table.

```
COLUMN Product_Number FORMAT A14;

COLUMN Description FORMAT A25;

COLUMN Sale_Price FORMAT 99,999.99;

SELECT Product_Number, Description, Sale_Price

FROM Product;
```

```
PRODUCT_NUMBER DESCRIPTION               SALE_PRICE

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

1              Large Desk                    276.60

2              Small Desk                    138.00

3              Tiny Desk                      16.96
```

A **CREATE VIEW** statement creates a view named **Products_Over100** to display only data for products where the **Sale_Price** is greater than $100.00. The suffix **VW** clearly identifies this object as a view, although not all IS shops use this naming convention.

```
CREATE VIEW Products_Over100_VW

    AS

SELECT Product_Number, Description, Sale_Price

FROM Product WHERE Sale_Price > 100
```

```
WITH CHECK OPTION CONSTRAINT Product_Sale_Price_CK;

View created.
```

You can retrieve data rows from a view using the standard **SELECT** statement. Note that only more expensive products are listed.

```
SELECT * FROM Products_Over100_VW;

PRODUCT_NUMBER DESCRIPTION                SALE_PRICE
-------------- ------------------------- ----------
1              Large Desk                     276.60
2              Small Desk                     138.00
```

A check constraint limits **INSERT** and **UPDATE** operations against a view such that only rows that the view can select can be inserted or updated. This **INSERT** command updates the underlying **Product** table.

```
INSERT INTO Products_Over100_VW VALUES ('4', 'Executive
Desk', 495.95);

1 row created.
```

The rows from the underlying **Product** table include the new product. Note that the new row has no **Retail_Price** or **Wholesale_Price** values – they are **NULL** – values for these columns were not inserted through the view and there is no default value for those columns.

```
SELECT * FROM Product;
```

```
PRODUCT_NUMBER DESCRIPTION     RETAIL_PRICE
WHOLESALE_PRICE SALE_PRICE

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

1              Large Desk              599.95
240.52      276.60

2              Small Desk              429.95
120      138.00

3              Tiny Desk               100.50
14.75       16.96

4              Executive Desk
495.95
```

This **INSERT** command fails to update the underlying **Product** table because the command tries to insert a product data row with a **Sale_Price** value that is less than **$100**. You can try to insert the data row directly into the Product table – the row will insert. This demonstrates that a view can be used as a means of enforcing a very specific data integrity check option constraint.

```
INSERT INTO Products_Over100_VW VALUES ('5', 'Cheap
Desk', 95.95);
```

```
ERROR at line 1:

ORA-01402: view WITH CHECK OPTION where-clause
violation
```

JOIN Views

A **JOIN VIEW** includes column data from more than one base table. This example joins the Course and Section tables created during an earlier lab assignment.

- The **Course_Number** and **Description** columns are in the **Course** table.
- The other columns are in the **Section** table.
- Since **Course_Number** is found in both tables, it is qualified in the **SELECT** clause with an alias for the **Course** table.

```
CREATE VIEW Course_Section_VW

    AS

SELECT C.Course_Number, Description, Section_Number,
```

```
    Section_Term, Section_Year
FROM Course C JOIN Section S
    ON C.Course_Number = S.Course_Number
WHERE Section_Year >= 2007;
```

```
View created.
```

This selects rows from both the Course, Section, and Course_Section_VW tables/views. Can you determine why the view only has two rows?

```
SELECT * FROM Course_Section_VW;
```

```
COURSE_ DESCRIPTION       SECTION_NUMBER SECTIO
SECTION_YEAR

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

CMIS565 Oracle DBA                 111111 Summer
2007

CMIS565 Oracle DBA                 222222 Fall
2007
```

```
SELECT * FROM Course;
```

```
COURSE_ DESCRIPTION                      HOURS
```

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

CMIS565 Oracle DBA                          3


SELECT * FROM Section;


SECTION_NUMBER SECTIO SECTION_YEAR COURSE_ LOCATION

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

        111111 Summer          2007 CMIS565 FH-3208

        222222 Fall            2007 CMIS565 FH-3103

        111112 Fall            2007         FH-3208

        111113 Summer          2007         FH-0301
```

Join views are updateable, but there are a number of restrictions regarding updating this type of view. For example, only one underlying table can be modified by an INSERT, UPDATE, or DELETE operation. Rules governing an updateable join view are covered in detail in the assigned readings and later in this note set.

Altering and Dropping Views

Views are not normally altered unless the underlying base tables or referenced views change – then the view should be explicitly recompiled.

- You can alter any view in your schema.

- To alter a view in another schema you need the **ALTER ANY TABLE** system privilege to execute an **ALTER VIEW** command. Example:

```
ALTER VIEW Course_Section_VW COMPILE;
```

Use the **DROP VIEW** command to drop a view that is no longer needed.

- You can drop any view in your schema.

- To drop a view from another schema you need the **DROP ANY VIEW** system privilege.

```
DROP VIEW Course_Section_VW;
```

There are two ways to replace views:

- Drop and then create the view again.

- Use the **CREATE OR REPLACE VIEW** command. Example:

```
CREATE VIEW OR REPLACE Course_Section_VW
    AS
SELECT C.Course_Number, Description, Section_Number,
```

```
    Section_Term, Section_Year
FROM Course C JOIN Section S
    ON C.Course_Number = S.Course_Number
WHERE Section_Year >= 2007;
```

DML Restrictions on Views

When can **INSERT**, **UPDATE**, and **DELETE** operations be used for base tables defined by a view?

- A view defined by a **SELECT** statement that contains the **SET** or **DISTINCT** operators, a **GROUP BY** clause, or a group function cannot be used for **INSERT**, **UPDATE**, or **DELETE** operations.

- If a view cannot select a row from a base table (due to a **WITH CHECK OPTION**), **INSERT** and **UPDATE** operations cannot be used through the view – see the earlier example for the view named **Products_Over100_VW**.

- If a view omits a **NOT NULL** column (that does not have a **DEFAULT** clause), then **INSERT** operations cannot be used through the view.

Example: Attempt to insert a **NULL** value for the **DESCRIPTION** column fails.

```
INSERT INTO Products_Over100_VW VALUES ('6', NULL,
495.95);

                                                *

ERROR at line 1:

ORA-01400: cannot insert NULL into
("DBOCK"."PRODUCT"."DESCRIPTION")
```

- A view created by using an expression cannot be used to INSERT or UPDATE rows in a base table.

- A join view can be updated, but is limited to updating only one base table participating in the view. There are numerous rules regarding updating a join view.
    - **UPDATE** – updatable columns must map to columns of a key-preserved table – that is a table where the key can also be a key of the virtual table resulting from the join.
    - **DELETE** – rows from a join view can be deleted providing there is only one key-preserved table in the join – a view created with the **WITH CHECK OPTION** clause with a key preserved table that is repeated cannot have the rows deleted through the view.
    - **INSERT** – rows inserted cannot reference columns of any non-key-preserved table. **INSERT** operations are not permitted if the join view is defined with the **WITH CHECK OPTION** clause.

The **DBA_UPDATABLE_COLUMNS** shows all columns in all tables and views that are modifiable with DML statements.

Example: The first query shows which columns are updatable from the **Products_Over100_VW** view. Note the use of the **Table_Name** column name from the **DBA_Updatable_Columns** view even though it is a view that is being evaluated. The second query evaluates the **Course_Section_VW** view – it has columns that are not updatable.

```
SELECT Column_Name, Updatable

FROM DBA_Updatable_Columns

WHERE Table_Name = 'PRODUCTS_OVER100_VW';
```

```
COLUMN_NAME                    UPD

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

PRODUCT_NUMBER                 YES

DESCRIPTION                    YES

SALE_PRICE                     YES
```

```
SELECT Column_Name, Updatable

FROM DBA_Updatable_Columns

WHERE Table_Name = 'COURSE_SECTION_VW';
```

```
COLUMN_NAME                    UPD
------------------------------ ---
COURSE_NUMBER                  NO
DESCRIPTION                    NO
SECTION_NUMBER                 YES
SECTION_TERM                   YES
SECTION_YEAR                   YES
```

END OF NOTES