DAMG 6210
Database Design

Nik Bear Brown
@NikBearBrown

SQL Views, Synonyms, Sequences & Null

Topics

- SQL Views
- Synonyms
- Sequences

View Definition

- A relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.
- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

Example Queries

- A view consisting of branches and their customers

```
create view all_customer as
        (select branch_name, customer_name
         from depositor, account
        where depositor.account_number =
              account.account_number )
        union
        (select branch_name, customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number )
```

- Find all customers of the Perryridge branch

```
select customer_name
      from all_customer
      where branch_name = 'Perryridge'
```

View Stability

- A view does not actually store any data. The data needed to support queries of a view are retrieved from the underlying database tables and displayed to a result table whenever a view is queried. The result table is only stored temporarily.
- If a table that underlies a view is dropped, then the view is no longer valid. Attempting to query an invalid view will produce an ORA-04063: view "VIEW_NAME" has errors error message.

Uses of Views

- Hiding some information from some users
  - Consider a user who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount.
  - Define a view

    (**create view** *cust_loan_data* **as**
    select *customer_name, borrower.loan_number, branch_name*
    from *borrower, loan*
    where *borrower.loan_number = loan.loan_number*)
  - Grant the user permission to read *cust_loan_data,* but not *borrower or loan*
- Predefined queries to make writing of other queries easier
  - Common example: Aggregate queries used for statistical analysis of data

Processing of Views

- When a view is created
  - the query expression is stored in the database along with the view name
  - the expression is substituted into any query using the view
- Views definitions containing views
  - One view may be used in the expression defining another view
  - A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
  - A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
  - A view relation v is said to be *recursive* if it depends on itself.

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all accounts with the maximum balance

```
with max_balance (value) as
    select max (balance)
    from account
select account_number
from account, max_balance
where account.balance =
max_balance.value
```

Defining Views

Views are relations, except that they are not physically stored.

They are used mostly in order to simplify complex queries and to define conceptually different views of the database to different classes of users.

```
CREATE VIEW  telephony-purchases AS
  SELECT product, buyer, seller, store
  FROM  Purchase, Product
  WHERE  Purchase.product = Product.name
         AND  Product.category = “telephony”
```

Updating Views

How can I insert a tuple into a table that doesn't exist?

```
CREATE VIEW  bon-purchase  AS
 SELECT   store, seller, product
 FROM      Purchase
 WHERE    store = "The Bon Marche"
```

If we make the following insertion:

```
INSERT INTO  bon-purchase
 VALUES  ("the Bon Marche", Joe, "Denby Mug")
```

We can simply add a tuple

("the Bon Marche", Joe, NULL, "Denby Mug")

to relation Purchase.

Non-Updatable Views

```
CREATE VIEW  Seattle-view  AS

    SELECT  seller, product, store
    FROM    Person, Purchase
    WHERE   Person.city = "Seattle"    AND
            Person.name = Purchase.buyer
```

How can we add the following tuple to the view?

(Joe, "Shoe Model 12345", "Nine West")

Updateable Views

- You can insert a row if the view in use is one that is updateable (not read-only).
- A view is updateable if the INSERT command does not violate any constraints on the underlying tables.
- This rule concerning constraint violations also applies to UPDATE and DELETE commands.

Drop View

- A DBA or view owner can drop a view with the DROP VIEW command. The following command drops a view named *dept_view*.

```
DROP VIEW dept_view;
View dropped.
```

Synonyms

- A *synonym* is an *alias*, that is, a form of shorthand used to simplify the task of referencing a database object.

- **<u>Creating Synonyms</u>**
- The general form of the CREATE SYNONYM command is:

```
CREATE [PUBLIC] SYNONYM synonym_name
FOR object_name;
```

Synonyms

- There are two categories of synonyms, *public* and *private*.
- A public synonym can be accessed by any system user.
- The individual creating a public synonym does not own the synonym – rather, it will belong to the PUBLIC user group that exists within Oracle.
- Private synonyms, on the other hand, belong to the system user that creates them and reside in that user's schema.

Synonyms

- A system user can grant the privilege to use private synonyms that they own to other system users.
- In order to create synonyms, you will need to have the CREATE SYNONYM privilege.
- This privilege will be granted to you by the DBA.
- You must have the CREATE PUBLIC SYNONYM privilege in order to create public synonyms.

Synonyms

- The three advantages to synonym usage.

  · First, a synonym provides what is termed *location transparency* because the synonym name hides the actual object name and object owner from the user of the synonym.

  · Second, you can create a synonym for a database object and then refer to the synonym in application code. The underlying object can be moved or renamed, and a redefinition of the synonym will allow the application code to continue to execute without errors.

  · Third, a public synonym can be used to allow easy access to an object for all system users.

Dropping Synonyms

- If you own a synonym, you have the right to drop (delete) the synonym. The DROP SYNONYM command is quite simple.

  DROP SYNONYM synonym_name;

- In order to drop a public synonym you must include the PUBLIC keyword in the DROP SYNONYM command.
- In order to drop a public synonym, you must have the DROP PUBLIC SYNONYM privilege.

  DROP PUBLIC SYNONYM synonym_name;

Renaming Synonyms

- Private synonyms can be renamed with the RENAME SYNONYM command.
- All existing references to the synonym are automatically updated.
- Any system user with privileges to use a synonym will retain those privileges if the synonym name is changed.
- The syntax of the RENAME SYNONYM command is like that for the RENAME command for any other database object such as a view or table.

  RENAME old_synonym_name TO new_synonym_name;

Renaming Synonyms

- The RENAME SYNONYM command only works for private synonyms.
- If we attempt to rename a public synonym such as the *tblspaces* synonym, Oracle will return an ORA-04043: *object tblspaces does not exist* error message as is shown here.

```
RENAME tblspaces TO ts;
ORA-04043: object TBLSPACES       does not
exist
```

Sequences

- SQL provides the capability to generate sequences of unique numbers, and they are called **sequences.**
- Just like tables, views, indexes, and synonyms, a sequence is a type of database object.
- Sequences are used to generate unique, sequential integer values that are used as primary key values in database tables.
- The sequence of numbers can be generated in either ascending or descending order.

Creating Sequences

- The syntax of the CREATE SEQUENCE command is fairly complex because it has numerous optional clauses.

```
CREATE SEQUENCE <sequence name>
[INCREMENT BY <number>]
[START WITH <start value number>]
[MAXVALUE <MAXIMUM VLAUE NUMBER>]
[NOMAXVALUE]
[MINVALUE <minimum value number>]
[CYCLE]
[NOCYCLE]
[CACHE <number of sequence value to cache>]
[NOCACHE]
[ORDER]
[NOORDER];
```

Sequences Example

```
CREATE SEQUENCE order_number_sequence
INCREMENT BY 1
START WITH 1
MAXVALUE 100000000
MINVALUE 1
CYCLE
CACHE 10;
Sequence created.
```

Accessing Sequence Values

- Sequence values are generated through the use of two *pseudocolumns* named *currval* and *nextval*.
- A pseudocolumn behaves like a table column, but psuedocolumns are not actually stored in a table.
- We can select values from pseudocolumns but cannot perform manipulations on their values.
- The first time you select the *nextval* pseudocolumn, the initial value in the sequence is returned.
- Subsequent selections of the *nextval* pseudocolumn will cause the sequence to increment as specified by the INCREMENT BY clause and will return the newly generated sequence value.

Accessing Sequence Values

- The *currval* pseudocolumn returns the current value of the sequence, which is the value returned by the last reference to nextval.

- **Example**

```
CREATE TABLE sales_order (
order_number NUMBER(9)
  CONSTRAINT pk_sales_order PRIMARY KEY,
  order_amount NUMBER(9,2));
```

Accessing Sequence Values

- The INSERT commands shown below insert three rows into the *sales_order* table. The INSERT commands reference the *order_number_sequence.nextval* pseudocolumn.

```
  INSERT INTO sales_order
    VALUES(order_number_sequence.nextval,
155.59 );
  INSERT INTO sales_order
    VALUES(order_number_sequence.nextval,
450.00 );
  INSERT INTO sales_order
  VALUES(order_number_sequence.nextval,
16.95);
```

Accessing Sequence Values

- Use of *currval.*

```
CREATE TABLE order_details (
      order_number  NUMBER(9),
      order_row     NUMBER(3),
      product_desc  VARCHAR2(15),
      quantity_ordered NUMBER(3),
      product_price NUMBER(9,2),
CONSTRAINT pk_order_details
      PRIMARY KEY (order_number, order_row),
CONSTRAINT fk_order_number FOREIGN KEY (order_number)
REFERENCES sales_order);
```

Dropping a Sequence

- DROP SEQUENCE command is used to drop sequences that need to be recreated or are no longer needed.
- The general format is shown here along with an example that drops the *order_number_sequence* object.

DROP SEQUENCE <sequence name>;

DROP SEQUENCE order_number_sequence;

Sequence dropped.

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.
  - Example: Find all loan number which appear in the *loan* relation with null values for *amount.*

select *loan_number*
from *loan*
where *amount* **is null**

- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- However, aggregate functions simply ignore nulls
  - More on next slide

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - Example*: 5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: *(true* **and** *unknown) = unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Null Values and Aggregates

- Total all loan amounts

  select sum (*amount*)
  from *loan*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount

- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.