Curs 5: Views and Materialized Views

Views. Views are stored in the database as queries definition. They do not store actual data but function as an alternative presentation of information stored in tables. Different displays of the same table may be accessed by different users. Whenever data are requested from a view, the underling query is executed. For instance, for the second query in figure 1, a full scan on table ***employees*** is performed as indicated in the from clause of the query defining the view.

Views **simplify complex queries**, hiding complex joins, aggregations or other complex calculations frequently used by the business logic of the application, so that the interaction with the persistent layer is made plain, but they do not impact the performance of queries.

```
create or replace view info_emp as
select employee_id, last_name, sysdate - hire_date nb_days
from employees;


select * from info_emp;
```

Figure 1: Creating and querying a view.

Any DML (select, insert, update, delete) operations we perform on a view will be performed on a table that is found in the query that defines the view. For example, the results of the update in figure 2 will be seen in ***employees*** table.

```
update info_emp
set last_name = 'Woody'
where employee_id = 19;


select * from employees
where employee_id = 19;
```

Figure 2: DML operations on views.

Views **secure data** as they provide means to restrict users' access to sets of rows, columns, or tables. Figure 3 lists the statements to grant or privileges to perform DML operations on a view.

```
grant all on prod_cat to proiect2;


revoke update on prod_cat from proiect2;
```

Figure 3: Users' privileges on views.

Unwanted updates are prevented by *READ ONLY* or *WITH CHECK OPTION* clauses.

In the example in figure 4, the product with the *product_id* 22 is associated with the category with *category_name* 'Storage' and *category_id* = 5. If the category id would be changed to 'CPU', the product 22 would disappear from the view. The second update is prevented by the *WITH CHECK OPTION* clause.

```
create view prod_cat as
select     p.product_id, p.product_name, p.standard_cost,
           p.category_id, pc.category_name
from products p join product_categories pc
     on (p.category_id = pc.category_id)
where pc.category_name in ('Video Card', 'Storage')
with check option;


update prod_cat
set category_id = 3
where product_id = 22;
```

Figure 4: WITH CHECK OPTION CLAUSE.

As mentioned in Codd's **rule 4**, for relational database management systems, views metadata are to be interrogated in the same way we interrogate other tables (relations). In the data dictionary we may find the complete list of columns presented in all views, and also, we may check if a column is updatable or not. The query in figure 5 shows that columns *CATEGORY_ID* and *CATEGORY_NAME* are not updatable.

```
select table_name, column_name, updatable
from user_updatable_columns
where table_name = 'PROD_CAT';
```

```
TABLE_NAME    COLUMN_NAME         UPDATABLE
----------------------------------------
PROD_CAT      PRODUCT_ID          YES
PROD_CAT      PRODUCT_NAME        YES
PROD_CAT      STANDARD_COST       YES
PROD_CAT      CATEGORY_ID         NO
PROD_CAT      CATEGORY_NAME       NO
```

Figure 5: Using the data dictionary for views.

If we remove *WITH CHECK OPTION* clause from the definition of *PROD_CAT* view, the column *CATEGORY_NAME* will still be un-updateable. Notice in figure 6 that the definition of a view is easily replaced by using *OR REPLACE* clause.

```
create or replace view prod_cat as
select     p.product_id, p.product_name, p.standard_cost,
           p.category_id, pc.category_name
from products p join product_categories pc
     on (p.category_id = pc.category_id)
where pc.category_name in ('Video Card', 'Storage');

select table_name, column_name, updatable
from user_updatable_columns
where table_name = 'PROD_CAT';
```

```
TABLE_NAME    COLUMN_NAME         UPDATABLE
----------------------------------------
PROD_CAT      PRODUCT_ID          YES
PROD_CAT      PRODUCT_NAME        YES
PROD_CAT      STANDARD_COST       YES
PROD_CAT      CATEGORY_ID         YES
PROD_CAT      CATEGORY_NAME       NO
```

Figure 6: Key preserved tables.

Codd's **rule 6** states that "all views that are theoretically updatable are also updatable by the system". Yet in most of the RDBMS updatable views are only those that map distinct rows in a single base table.

In Oracle DB, if the query defining the view performs joins, then only the columns that are included in a ***key-preserved table*** are updatable. A table is a key-preserved table if the table key participates in the view as a key. In other words, a key-preserved table has its key columns preserved through a SQL join. In the example above, in figure 5, table *PRODUCTS* is key preserved because its key *product_id* is also a key in the view. Table *PRODUCT_CATEGORIES* is not key preserved as its key *category_id* is not a key in the view. There are many products associated with the same category, hence a value of *category_id* key will appear on multiple rows in the view.

Materialized views.

A materialized view physically store content in database segments. The content of a materialized view is computed manually or automatically to reflect the results of the defining stored query. Because data from the base tables is replicated in the materialized view, materialized views have not only the advantages of providing different presentations of the same information and to secure some database object, but **also increase performance of complex queries**. The execution of the second query from figure 7 is performed as a full scan on the data stored in the view *CUST_ORDER* which is faster that performing a join and a group by on the resulting inline view.

```
create materialized view cust_order as
select o.order_id, o.customer_id, sum(quantity * unit_price) value
from orders o join order_items oi on (o.order_id = oi.order_id)
group by o.order_id, o.customer_id;


select * from cust_order;
```

Figure 7: Creating a materialized view with aggregation.

Because materialized views increase query performance, they are extensively used in data warehouse schemas to precompute and store aggregated data. In such schemas, materialized views are referred to as *summaries*.

Just like in the case of indexes, maintaining the view in a consistent state with the changes performed on the base tables may negatively impact the performance of LMD operations. For instance any requests of type insert, update, delete on tables ORDERS or ORDER_ITEMS must trigger an update of the corresponding lines in the view. There are two strategies for materialized views maintenance: **IMMEDIATE** and **DEFFERED**. Immediate refresh means that the update (most of the times ***incremental*** update) will be performed immediately after a DML that should modify the view.

Deferred strategy postpones the refresh to reduce the overhead on update transactions. For instance, summaries may be recalculated at the end of the day taking into account that the aggregated vales may not reflect all the updates until the views are refreshed.

Oracle provides two refresh options ON COMMIT and ON DEMAND corresponding to the strategies presented above. Also, the materialized view may be populated immediately or when the first refresh is requested (see options BUILD IMMEDIAT or BUILD DEFERRE in figure 8). The refresh may be incremental, meaning that only some lines will be affected. For instance, if we add an *order* and *order_items* lines corresponding to it, only a single line will be added in the view *cust_order.* Such fast refresh may be performed only if we create logs on the base tables, see figure 9

```
CREATE MATERIALIZED VIEW view-name
BUILD [IMMEDIATE | DEFERRED]
REFRESH [FAST | COMPLETE | FORCE]
ON [COMMIT | DEMAND]
[[ENABLE | DISABLE] QUERY REWRITE]
```

Figure 8: Creating a materialized, refresh option.

A materialized view log is a table associated with a master table of a materialized view. When DML operations (such as INSERT, UPDATE, or DELETE) are performed on the master table, the database stores rows describing those changes in the materialized view log, see figure 9.

Materialized views can optimize query performance even if they are not explicitly present in the form clause. “The optimizer looks for materialized views that are compatible with the user query, and then uses a cost-based algorithm to select materialized views to rewrite the query. The optimizer does not rewrite the query when the plan generated unless the materialized views has a lower cost than the plan generated with the materialized views”, for more details see [1]. For example, for the first select in the last query in figure 10, the optimizer chooses to scan the view instead of scanning tables *orders* and *order_items*.

```
create materialized view log on order_items
WITH ROWID,  SEQUENCE(order_id, item_id, product_id, quantity,
unit_price) INCLUDING NEW VALUES;

create materialized view log on orders
WITH ROWID,  SEQUENCE(order_id, customer_id, status, order_date)
INCLUDING NEW VALUES;
```

Figure 9: Materialized view logs.

```
create materialized view cust_order
build immediate
refresh fast on commit
enable query rewrite
as
select o.order_id, o.customer_id, sum(quantity * unit_price) value
from orders o join order_items oi on (o.order_id = oi.order_id)
group by o.order_id, o.customer_id;

select o.order_id, o.customer_id, sum(quantity * unit_price) value
from orders o join order_items oi on (o.order_id = oi.order_id)
group by o.order_id, o.customer_id
union
select o.order_id, o.customer_id, count(*) nr
from orders o join order_items oi on (o.order_id = oi.order_id)
group by o.order_id, o.customer_id
order by 1,2;
```

Figure 10: Materialized view for query rewriting.

Bibliography

[1] https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/sql-tuning-guide.pdf.

[2] https://www.dba-oracle.com/t_key_preserved_table.htm

[3] https://docs.oracle.com/en/database/oracle/oracle-database/19/dwhsg/basic-materialized-views.html