Views - The 'other' database object

Erick D. Slazinski
Department of Computer Technology, Purdue University
West Lafayette, IN 47907-1421, USA

Abstract

Database views are a powerful and versatile construct that, if used creatively, can solve several commonly occurring business problems. These problems include integration issues and backwards compatibility, location transparency, managing overly complex Structured Query Language (SQL) queries and overcoming some limitations of the SQL language. This paper documents some tips and techniques that the author has found in his many years (12+) in industry. Database (DB) views are basic objects defined in the ANSI SQL-92 standard. As an integral part of the SQL language, which builds upon the select statement, views are easy to teach and generally well understood by students. With the ease of development of views coupled with the expressive power that is contained in the select statement, it is the author's recommendation that views should be included in any intermediate to advanced database course.

Keywords: Views, RDBMS, SQL, RAD

1. Introduction

Database views are often underestimated and therefore one of the most underused objects in today's Relational Database Management Systems (RDBMSs). Besides offering traditional functionality of information hiding (often as a form of security enforcement) and pre-generated reports - database views can also offer a mechanism to insulate software developers from a changing database schema (as often occurs in today's Rapid Application Development (RAD) environment), provide backwards-compatibility when a system evolves, provide location transparency for a distributed database environment, reduce query complexity, solve integration problems and overcome certain limitations of SQL.

2. Traditional Uses of Views

"According to the SQL-92 standard, views are virtual tables that act as if they are materialized when their name appears" (Celko, 1999, p.55). The term virtual is used because the only permanent storage that a view uses is the Data Dictionary (DD) entries that the RDBMS defines. When the view is accessed (by name) in a SQL from clause, the view is materialized. This materialization is simply the naming, and storing in the RDBMS's temporary space, the result set of the view's select statement. When the RDBMS evaluates the statement that used the view's name in its from clause, the named result set is then referenced in the same fashion as a table object. Once the statement is complete the view is released from the temporary space. This guarantees read consistency of the view. A more permanent form of materialized views is now being offered by RDBMS vendors as a form of replication, and is not germane to this discussion.

The syntax for creating a database view is:

```
SQL> create view <name> [<column list>] as
        <select statement>
```

The power of a view is that the select statement is almost **any** select statement no matter how simple or complex – various vendors may disallow certain functions to be used in the select statement portion of a view's definition.

The classic use of views is to implement an organization's security policy. One such policy could be to restrict the data that a user is entitled to see. In the following example, an employee would be allowed to see his/her own personal information, but no one else's.

```
SQL> create view EMP_DATA as
        select * from EMPLOYEE
        where ENAME = User;[1]
```

[1] User is an Oracle pseudo-column, which contains the username that executed the query (Loney and Koch, 2000, p.68).

This view is a simple and elegant way of enforcing security with a minimal of effort on the database developer's part. We now have a mechanism that provides a consistent approach to security. By implementing security policies as a layer of database views, we can remove all access rights to the underlying tables and manage access rights, since the views are actually performing the security checks. .

The second classic use of a view is to provide "canned queries" to users. These queries are often used to support the day-to-day operations and are stored in a view in order to save time from re-entering the query and also reduces the level of SQL required on the part of the user who needs the query. Advanced queries, such as those used in generating reports can be stored as a view. These reports frequently contain tedious formatting such as splitting apart a customer's first and last name; converting the name into mixed case; performing data translation such as converting an employee's department id into a department name; complex data transformations such as converting the time stored in that database into the current time zone or converting temperature reading into degrees Fahrenheit or degrees Celsius depending on the user's location. Additional, more advanced features include integrating data possibly stored on multiple database servers, providing backwards compatibility to a previous schema revision level shall be discussed later in this paper. An example of a canned query follows:

```
SQL> create view SALARY_REPORT as
        select initcap(ENAME) "Name",
        to_char (SAL, '$99,999') "Salary",
        to_char (HIREDATE,
                'fmddth "of" Month, yyyy')
                                "Start Date"
        from EMP;[2]
```

Views of Views

Another benefit of having the select statement is that views can be built in a hierarchical fashion, that is one view based on another view, which may continue on until either a RDBMS imposed limit is reached or performance is severely impacted.. This allows for an incremental build approach to developing SQL solutions. The reader must be warned, however, that based on the author's experience, layering views more than 3 levels deep can incur a serious performance penalty. This penalty makes sense when one thinks about what is actually occurring inside the database server itself – each layer of views must be materialized in temporary storage, **in sequence**.

DML and Views

In addition to proving a mechanism for storing database queries, views can be treated as an insulating layer (a feature that we shall exploit later) when executing Data Manipulation Language (DML) statements. It is often desirable to execute these basic operations: **insert, update** and **delete** against a view, especially if users are used to using the view during select operations. However, when DML operations are performed against a view, the user is sometimes reminded that they are no longer dealing with a database table. Rules regulating the conditions that these operations may be performed are vendor specific. Though the regulations can be annoying, the integrity of the underlying database objects must be maintained. An example of this follows:

Given a customer table and associated view to support verifying personal information.

```
SQL> create table CUSTOMER (
    CUSTID NUMBER (6) not null,
    NAME    VARCHAR2(45),
    ADDRESS VARCHAR2(40),
    CITY    VARCHAR2(30),
    STATE   VARCHAR2(2),
    ZIP     VARCHAR2(9),
    AREA    NUMBER (3),
    PHONE   VARCHAR2(9),
    REPID   NUMBER (4) not null,
    CREDITLIMIT NUMBER (9,2));


SQL> create view WEB_CUSTOMER as
        select NAME, ADDRESS, CITY,
               STATE,  ZIP, AREA, PHONE
        from CUSTOMER;
```

To entice new customers to purchase from our company over the web, our developers add a "create new customer" form where the user enters the required information. However, entering data through our view now causes the following error:

```
SQL> INSERT INTO WEB_CUSTOMER
        VALUES ('John Doe',
                '123 Any street', 'Any City',
                'IN', '47905', 123,
                '456-7890');

ORA-01400:    cannot    insert    NULL    into
("ERICK"."CUSTOMER"."CUSTID")
```

It is easy to see why a user can get an error when trying to insert data 'through' the (above) view - if the view does not have the non-null attributes that its underlying objects have, the insert will fail because the user is unaware **and** unable to populate the attributes that are required. Updates are usually another problem area - again with an eye towards data integrity the rules and restrictions that views have do make annoying sense.

Database vendors such as Oracle have given the database developer a mechanism to help the end-user - **instead-of triggers**. These triggers are actually defined on the view. This may seem a bit odd to some readers since

[2] The functions `initcap` and `to_char` are string-formatting functions.

the trigger is being placed on a virtual structure, however, remember that the view is materialized and does exist for duration of the query being executed. Like all other triggers, **instead-of** trigger can be defined to fire on any DML statement. Thus the **instead-of** trigger will intercept the action and apply the DML statement on the correct base object(s). The following **instead-of** trigger, when added to the above view will correct the insert error.

```
SQL> create or replace trigger
    WEB_CUST_IBR
    instead of INSERT on WEB_CUSTOMER
    for each row
    begin
      insert into CUSTOMER values
          (CUST_ID.NEXTVAL, :new.NAME,
           :new.ADDRESS, :new.CITY,
           :new.STATE, :new.ZIP,
           :new.AREA, :new.PHONE,
          999, 1000);
     end WEB_CUSTOMER_IBR;
```
[3]

Now, when the insert statement is run, we have a successful transaction.

```
SQL> INSERT INTO WEB_CUSTOMER
    VALUES ('John Doe', '123 Any street',
                'Any City',  'IN', '47905', 123,
                '456-7890');
```

1 row created.

If **instead-of** trigger functionality is not available from a specific vendor – DML-specific stored procedures could provide the similar functionality, though not as transparent. For example, you could create an `INSERT_CUSTOMER` stored procedure that correctly stored the data in the database.

3. Solving Integration Issues or Providing Backwards Compatibility

Integration and backwards compatibility are really the same problem that a developer must face at different points in a system's lifecycle. Integration issues can arise when bringing together systems that were not created to be compatible. While backwards compatibility is a strong motivation when enhancing an existing or developing any replacement system, the cost of meeting this requirement may be prohibitive. In either situation, the goal is the same; we need to get some existing code to work with a new schema.

In the following example, we have a table that maintains information about courses. A new requirement stating the need to store an additional description attribute is required to support the condensed summer version of the course. The database architect has decided to normalize the descriptions out into a separate table.

[3] CUST_ID.NEXTVAL is an Oracle sequence for auto-generating a unique set of integer values.

Original table:

```
SQL> CREATE TABLE COURSE (
      CRS_NO          varchar2(5) not null,
      DESC            varchar2(2000) not null,
      CREDITS          number,
      EQUIV_CRS_NO  varchar2(5));
```

Becomes:

```
SQL> create table COURSE_REV2 (
      CRS_NO    varchar2(5) not null,
      CREDITS   number,
      EQUIV_CRS_NO  varchar2(5));

SQL> create table COURSE_DESCRIPTION
        (crs_no        varchar2 (5)  not null,
         semster_desc varchar2 (2000)  not null,
         summer_desc varchar2 (2000));
```

This has the adverse effect of causing all of the existing code that accesses the course table to stop functioning. To get the legacy code to access the new database structure, we created a view that had the name of the original course table. To ensure backwards compatibility, the following was added:

```
SQL> CREATE VIEW COURSE
        (CRS_NO, DESC, CREDITS,
          EQUIV_CRS_NO) as
         select CB.CRS_NO,
                  CD.SEMSTER_DESC,
                  CREDITS, EQUIV_CRS_NO
         from COURSE_REV2 CB,
                COURSE_DESC CD
        where CB.CRS_NO = CD.CRS_NO;
```

With the view in place, all of the original queries should work as originally intended. To complete the insulating layer (by handling DML statements), **instead-of** triggers or stored procedures are used to hide the fact that a view is actually being referenced, instead of an actual table.

All new code would be written against the new database structure (either the views or actual tables) and any time-critical portions of the legacy code may also need to be rewritten.

4. Location Transparency

In the modern world of Global corporations, acquisitions and mergers, IT professionals often find themselves in a situation where they must produce an integrated set of corporate databases. Normally, this is a large undertaking that upper management is not concerned about. In order to satisfy the immediate need of producing *what looks like* an integrated set of corporate databases, while a thorough analysis of what an integrated set of corporate databases looks like, database developers can again use database views to solve the immediate need. Once again we use advanced features in our select statements, this time to provide location transparency – that is to provide a single unified set of tables that can be queried and accessed, even though the data is not co-located.

There are several ways to achieve location transparency. Most RDBMSs allow access to a remote server's data, by prefixing the traditional `owner.tablename` with a remote database name (of course, access protocols must be enforced by the appropriate database administrators (DBAs)). Access mechanisms can vary depending on the vendor. For instance, Oracle provides remote access through database links and synonyms.

The following example is from a class project, which entailed the bringing together of 3 related, but different database schema for the purposes of providing the end user (the instructor) with a single report, regardless of the data source. (This particular report was a log of all patient check-in / out activity at three different medical clinics.) Read access was granted to each member of the team, by each member, therefore no additional mechanisms were required.

```
create or replace view PATIENT_IN_OUT_LOG as

SELECT 'C' AS TYPE, TO_DATE(LOG_TIME, 'MM
DD YYYY HH24:MI:SS') AS DATETIME, Pa-
tient_Name AS NAME, In_Or_Out AS IO
FROM STUDENT1.PATIENT_LOG

UNION

SELECT 'E' AS TYPE, TO_DATE(LOG_TIME, 'DD
MM YYYY HH24:MI:SS') AS DATETIME,
PATIENT_NAME AS NAME, IN_OR_OUT AS IO
FROM STUDENT2.PATIENT_LOG

UNION

SELECT 'F' AS TYPE, TO_DATE(PATIENT_IN,
'Month DD, YYYY HH:MISS AM') AS DATETIME,
PATIENT_ID AS NAME, 'I' AS IO
FROM STUDENT3.PATIENT_LOG
WHERE PATIENT_OUT IS NULL

UNION

SELECT 'F' AS TYPE, TO_DATE(PATIENT_OUT,
'Month DD, YYYY HH:MISS AM') AS DATETIME,
PATIENT_ID AS NAME, 'O' AS IO
FROM STUDENT3.PATIENT_LOG
WHERE PATIENT_IN IS NULL;
```

In this example, a fairly involved select statement is used for the view. The statement uses the SQL Union operator for combining the result sets from several individual queries into a single result set. In addition to combining the result sets, the individual SQL statements are executing some transformation logic in order to have a consistent set of output. Some items of note: to identify the data's original source an appropriate string constant (`'C', 'E', 'F'`) was assigned to each data row, not only does this identify the data, but provides the end user a mechanism to group, order and sort the data when retrieved from the view. The `TO_DATE` function, transforms a date time string into an Oracle date object. In this particular group, one individual (STUDENT3) didn't store the action (of checking in or out) explicitly, but used a separate date field to record each action's occurrence. This caused the group to have to create an additional select statement to map the data into the final form. The resulting dataset could then be manipulated into any of the required reports required by the project.

If the database administrator (DBA) is providing access to remote data through the use of a database link, then this implementation detail can and should be hidden in a view.

To create the database link, the DBA gives the link a unique name, can connect using a specific user or the connected user and determines the remote server's connect string.

```
SQL> create database link remote_site
        using 'NYC';

SQL> create view NYC_EMP_DATA as
        select * from NYC.EMP;
```

The view provides the DBA with the flexibility of moving the remote data to another database server with minimum impact to the developed code. Also security can be maintained at the view level, instead of having to manage access rights on the links.

5. Reducing Query Complexity

In addition to the real world applications, views can be used as a mechanism to teach incremental development of SQL solutions to business questions. When students first learn SQL, they are typically taught the basics of the querying a RDBMS via the SQL `select` statement. They are then taught to add complexity to the `select` statement, to achieve more robust outputs. With this learning cycle, students will often struggle to translate a business requirement into a single SQL Query, often resulting in (the vernacular) monster queries, which usually have terrible performance and are very difficult to maintain. Just as modular programming led to better applications, so too can modular SQL. Views can and should play an important role in teaching students how to break business problems down into step-wise solutions that are simpler to develop, easier to maintain and often more efficient.

6. Overcoming SQL limitations

In data warehousing, especially when working large sets of sales data, limitations of SQL as a reporting tool start to appear. For example, if you wish to discover daily averages, or averages of averages, most RDBMSs will not even try to parse the query. A developer could resort to coding the daily average logic in procedural extensions that DBMS vendors provide, or use a 3GL application, or they could stay with the SQL environment and create database views. The ability to break the problem down into manageable pieces serves developers well in this situation.

An example of the daily average problem can be found in Kimball's Data Warehouse Toolkit (1996)

If a row in a query result is supposed to contain the combined inventory level for a cluster of 3 products, 4 stores and 7 time periods (i.e. what is the average daily inventory of a brand in a geographic region during a given week), using the SQL AVG function would divide the summed inventory value by 3*4*7=84. The correct answer is to divide the summed inventory by 7, which is the number of time periods. (p.50).

To solve this problem we create a view that contains the average for a given week, grouped by brand and store.

```
create or replace view national_bottle_avg_sales as
select avg(dollar_sales) as
          Average_Dollar_Sales,
        brand, store_state, this_day
from sales_fact, time, product, store
where
  sales_fact.product_key=product.product_key
  and sales_fact.time_key=time.time_key
  and sales_fact.store_key=store.store_key
  and brand = 'National Bottle'
  and store_state = 'TN'
  and this_day between
               '1-Oct-94' and '7-Oct-94'
group by brand, store_state, this_day;
```

Now we are able to query the newly created view and average the weekly data. This allows us to create some interesting analytical reports without having to resort to a third party tool.

```
column brand format a20
column sales_region format a20
column this_day format a20
```

```
select brand, store_state, this_day,
        avg(Average_Dollar_Sales) as Sales
from national_bottle_avg_sales
group by brand, store_state, this_day;
```

4. Conclusion

As we have seen, views are simple constructs that if used creatively can help solve some problems that face today's database developers. Views in their simplest form are stored queries. They can provide a consistent access layer to data, thereby enforcing an organization's data security requirements. With some creative thinking views can lower the cost of system upgrades. With the power of the select statement coupled with vendor-supplied SQL functions, a view can be used to transform incompatible data sets into compatible data sets, thus accelerating and lowering the cost of integration efforts. Remote data can be integrated and viewed as a single data set though the use of remote access mechanisms. Finally, we have seen that certain limitations of the SQL language can be worked around using views.

Anyone who has been exposed to SQL understands the basics of the select statement. Views are merely an extension of this concept and provide an important mechanism to allow individuals to modularize their SQL into simpler, more maintainable code, which results in a lower total cost of ownership for the system.

7. References

Celko, Joe, 1999, Joe Celko's Data & Databases Concepts in Practice. Morgan Kaufmann Publishers, San Francisco.

Kimball, Ralph, 1996, The Data Warehouse Toolkit. John Wiley & Sons, Inc., New York.

Loney, Kevin and George Koch, 2000, ORACEL8*i* The Complete Reference. Osborne/McGray-Hill, New York.