Views

9 min. read · View original

The relational database provides programmers with a high degree of abstraction from the physical world of the computer. You can't tell where on the disk the RDBMS is putting each row of a table. For all you know, information in a single row might be split up and spread out across multiple disk drives. The RDBMS lets you add a column to a billion-row table. Is the new information for each row going to be placed next to the pre-existing columns or will a big new block of disk space be allocated to hold the new column value for all billion rows? You can't know and shouldn't really care.

A view is a way of building even greater abstraction.

Suppose that Jane in marketing says that she wants to see a table containing the following information:

- user_id
- email address

- number of static pages viewed
- number of bboard postings made
- number of comments made

This information is spread out among four tables. However, having read the preceding chapters of this book, you're perfectly equipped to serve Jane's needs with the following query:

```
select u.user_id,
       u.email,
       count(ucm.page_id) as n_pages,
       count(bb.msg_id) as n_msgs,
       count(c.comment_id) as n_comments
from users u, user_content_map ucm, bboard bb,
comments c
where u.user_id = ucm.user_id(+)
and u.user_id = bb.user_id(+)
and u.user_id = c.user_id(+)
group by u.user_id, u.email
order by upper(email)
```

Then Jane adds "I want to see this every day, updated with the latest information. I want to have a programmer write me some desktop software that connects directly to the database and looks at this information; I don't want my desktop software breaking if you reorganize the data model."

```
create or replace view janes_marketing_view
as
select u.user_id,
       u.email,
       count(ucm.page_id) as n_pages,
       count(bb.msg_id) as n_msgs,
       count(c.comment_id) as n_comments
from users u, user_content_map ucm, bboard bb,
comments c
where u.user_id = ucm.user_id(+)
and u.user_id = bb.user_id(+)
and u.user_id = c.user_id(+)
group by u.user_id, u.email
order by upper(u.email)
```

To Jane, this will look and act just like a table when she queries it:

```
select * from janes_marketing_view;
```

Why should she need to be aware that information is coming from four tables? Or that you've reorganized the RDBMS so that the information subsequently comes from six tables?

Protecting Privacy with Views

A common use of views is protecting confidential data. For example, suppose that all the people who work in a hospital collaborate by using a relational database. Here is the data model:

```
create table patients (
       patient_id     integer primary key,
       patient_name   varchar(100),
       hiv_positive_p char(1),
       insurance_p    char(1),
       ...
);
```

If a bunch of hippie idealists are running the hospital, they'll think that the medical doctors shouldn't be aware of a patient's insurance status. So when a doc is looking up a patient's medical record, the looking is done through

```
create view patients_clinical
as
select patient_id, patient_name, hiv_positive_p
from patients;
```

The folks over in accounting shouldn't get access to the patients' medical records just because they're trying to squeeze money out of them:

```
create view patients_accounting
as
select patient_id, patient_name, insurance_p from
patients;
```

Relational databases have elaborate permission systems similar to those on time-shared computer systems. Each person in a hospital has a unique database user ID.

Permission will be granted to view or modify certain tables on a per-user or per-group-of-users basis. Generally the RDBMS permissions facilities aren't very useful for Web applications. It is the Web server that is talking to the database, not a user's desktop computer. So the Web server is responsible for figuring out who is requesting a page and how much to show in response.

Protecting Your Own Source Code

The ArsDigita Shoppe system, described in http://philip.greenspun.com/panda/ecommerce, represents all orders in one table, whether they were denied by the credit card processor, returned by the user, or voided by the merchant. This is fine for transaction processing but you don't want your accounting or tax reports corrupted by the inclusion of failed orders. You can make a decision in one place as to what constitutes a reportable order and then have all of your report programs query the view:

```
create or replace view sh_orders_reportable
as
select * from sh_orders
where order_state not in
('confirmed','failed_authorization','void');
```

Note that in the privacy example (above) we were using the view to leave unwanted columns behind whereas here we are using the view to leave behind unwanted rows.

If we add some order states or otherwise change the data model, the reporting programs need not be touched; we only have to keep this view definition up to date. Note that you can define every view with "create or replace view" rather than "create view"; this saves a bit of typing when you have to edit the definition later.

If you've used `select *` to define a view and subsequently alter any of the underlying tables, you have to redefine the view. Otherwise, your view won't contain any of the new columns. You might consider this a bug but Oracle has documented it, thus turning the behavior into a feature.

Views-on-the-fly and OUTER JOIN

Let's return to our first OUTER JOIN example, from the simple queries chapter:

```
select users.user_id, users.email,
classified_ads.posted
from users, classified_ads
where users.user_id = classified_ads.user_id(+)
order by users.email, posted;


...
   USER_ID EMAIL
POSTED
---------- ----------------------------------- --
--------
     52790 dbrager@mindspring.com
     37461 dbraun@scdt.intel.com
     52791 dbrenner@flash.net
     47177 dbronz@free.polbox.pl
     37296 dbrouse@enter.net
     47178 dbrown@cyberhighway.net
     36985 dbrown@uniden.com
1998-03-05
     36985 dbrown@uniden.com
1998-03-10
     34283 dbs117@amaze.net
     52792 dbsikorski@yahoo.com
...
```

The plus sign after `classified_ads.user_id` is our instruction to Oracle to "add NULL rows if you can't meet this JOIN constraint".

Suppose that this report has gotten very long and we're only interested in users whose email addresses start with "db". We can add a WHERE clause constraint on the `email` column of the `users` table:

```
select users.user_id, users.email,
classified_ads.posted
from users, classified_ads
where users.user_id = classified_ads.user_id(+)
and users.email like 'db%'
order by users.email, posted;

   USER_ID EMAIL                             POSTED
---------- ------------------------------ -------
---
     71668 db-designs@emeraldnet.net
    112295 db1@sisna.com
    137640 db25@umail.umd.edu
     35102 db44@aol.com                      1999-
12-23
     59279 db4rs@aol.com
     95190 db@astro.com.au
     17474 db@hotmail.com
    248220 db@indianhospitality.com
     40134 db@spindelvision.com              1999-
02-04
    144420 db_chang@yahoo.com
     15020 dbaaru@mindspring.com
...
```

Suppose that we decide we're only interested in classified ads since January 1, 1999. Let's try the naive approach, adding another WHERE clause constraint, this time on a column from the `classified_ads` table:

```
select users.user_id, users.email,
classified_ads.posted
from users, classified_ads
```

```
where users.user_id = classified_ads.user_id(+)
and users.email like 'db%'
and classified_ads.posted > '1999-01-01'
order by users.email, posted;

   USER_ID EMAIL                             POSTED
---------- ------------------------------ -------
---
     35102 db44@aol.com                      1999-
12-23
     40134 db@spindelvision.com              1999-
02-04
     16979 dbdors@ev1.net                    2000-
10-03
     16979 dbdors@ev1.net                    2000-
10-26
    235920 dbendo@mindspring.com             2000-
08-03
    258161 dbouchar@bell.mma.edu             2000-
10-26
     39921 dbp@agora.rdrop.com               1999-
06-03
     39921 dbp@agora.rdrop.com               1999-
11-05

8 rows selected.
```

Hey! This completely wrecked our outer join! All of the rows where the user had not posted any ads have now disappeared. Why? They didn't meet the `and classified_ads.posted > '1999-01-01'` constraint. The outer join added NULLs to every column in the report where there was no corresponding row in the `classified_ads` table. The new constraint is comparing NULL to January 1, 1999 and the answer is... NULL. That's three-valued logic for you. Any computation involving a NULL turns out NULL. Each WHERE clause constraint must evaluate to true for a row to be kept in the result set of the SELECT. What's the solution? A "view on the fly". Instead of OUTER JOINing the `users` table to the `classified_ads`, we will OUTER JOIN `users` to a *view* of `classified_ads` that contains only those ads posted since January 1, 1999:

```

select users.user_id, users.email, ad_view.posted
```

```
from
  users,
  (select *
   from classified_ads
   where posted > '1999-01-01') ad_view
where users.user_id = ad_view.user_id(+)
and users.email like 'db%'
order by users.email, ad_view.posted;

   USER_ID EMAIL                           POSTED
---------- ------------------------------ -------
---
     71668 db-designs@emeraldnet.net
    112295 db1@sisna.com
    137640 db25@umail.umd.edu
     35102 db44@aol.com                   1999-
12-23
     59279 db4rs@aol.com
     95190 db@astro.com.au
     17474 db@hotmail.com
    248220 db@indianhospitality.com
     40134 db@spindelvision.com           1999-
02-04
    144420 db_chang@yahoo.com
     15020 dbaaru@mindspring.com
...
174 rows selected.
```

Note that we've named our "view on the fly" `ad_view` for the duration of this query.

How Views Work

Programmers aren't supposed to have to think about how views work. However, it is worth noting that the RDBMS merely stores the view definition and not any of the data in a view. Querying against a view versus the underlying tables does not change the way that data are retrieved or cached. Standard RDBMS views exist to make programming more convenient or to address security concerns, not to make data access more efficient.

How *Materialized* Views Work

Starting with Oracle 8.1.5, introduced in March 1999, you can have a *materialized view*, also known as a *summary*. Like a regular view, a materialized view can be used to build a black-box abstraction for the programmer. In other words, the view might be created with a complicated JOIN, or an expensive GROUP BY with sums and averages. With a regular view, this expensive operation would be done every time you issued a query. With a materialized view, the expensive operation is done when the view is created and thus an individual query need not involve substantial computation.

Materialized views consume space because Oracle is keeping a copy of the data or at least a copy of information derivable from the data. More importantly, a materialized view does not contain up-to-the-minute information. When you query a regular view, your results includes changes made up to the last committed transaction before your SELECT. When you query a materialized view, you're getting results as of the time that the view was created or refreshed. Note that Oracle lets you specify a refresh interval at which the materialized view will automatically be refreshed.

At this point, you'd expect an experienced Oracle user to say "Hey, these aren't new. This is the old CREATE SNAPSHOT facility that we used to keep semi-up-to-date copies of tables on machines across the network!" What is new with materialized views is that you can create them with the

ENABLE QUERY REWRITE option. This authorizes the SQL parser to look at a query involving aggregates or JOINs and go to the materialized view instead. Consider the following query, from the ArsDigita Community System's /admin/users/registration-history.tcl page:

```
select
  to_char(registration_date,'YYYYMM') as
sort_key,
  rtrim(to_char(registration_date,'Month')) as
pretty_month,
  to_char(registration_date,'YYYY') as
pretty_year,
  count(*) as n_new
from users
group by
  to_char(registration_date,'YYYYMM'),
  to_char(registration_date,'Month'),
  to_char(registration_date,'YYYY')
order by 1;


SORT_K PRETTY_MO PRET      N_NEW
------ --------- ---- ----------
199805 May       1998        898
199806 June      1998        806
199807 July      1998        972
199808 August    1998        849
199809 September 1998       1023
199810 October   1998       1089
199811 November  1998       1005
199812 December  1998       1059
199901 January   1999       1488
199902 February  1999       2148
```

For each month, we have a count of how many users registered at photo.net. To execute the query, Oracle must sequentially scan the `users` table. If the users table grew large and you wanted the query to be instant, you'd sacrifice some timeliness in the stats with

```
create materialized view users_by_month
   enable query rewrite
   refresh complete
   start with 1999-03-28
   next sysdate + 1
   as
   select
     to_char(registration_date,'YYYYMM') as
sort_key,
     rtrim(to_char(registration_date,'Month')) as
pretty_month,
     to_char(registration_date,'YYYY') as
pretty_year,
     count(*) as n_new
   from users
   group by
     to_char(registration_date,'YYYYMM'),
     to_char(registration_date,'Month'),
     to_char(registration_date,'YYYY')
   order by 1
```

Oracle will build this view just after midnight on March 28, 1999. The view will be refreshed every 24 hours after that. Because of the `enable query rewrite` clause, Oracle will feel free to grab data from the view even when a user's query does not mention the view. For example, given the query

```
select count(*)
from users
where rtrim(to_char(registration_date,'Month')) =
'January'
and to_char(registration_date,'YYYY') = '1999'
```

Oracle would ignore the `users` table altogether and pull information from `users_by_month`. This would give the same result with much less work. Suppose that the current month is March 1999, though. The query

```
select count(*)
```

> ```
> from users
> where rtrim(to_char(registration_date,'Month')) =
> 'March'
> and to_char(registration_date,'YYYY') = '1999'
> ```

will also hit the materialized view rather than the `users` table and hence will miss anyone who has registered since midnight (i.e., the query rewriting will cause a different result to be returned).

More:

Reference

Next: style

---

philg@mit.edu

Reader's Comments

> In this on views you state that condition "classified_ads.posted > '1999-01-01' " will not give the desired results because the column 'posted' is nullable hence this condition will compute to NULL whenever 'posted' column is NULL. Hence the query will never return rows where 'posted' value is NULL.
>
> And in order to solve this issue you go onto to create a view with the following query: (select * from classified_ads where posted > '1999-01-01')
>
> Wont this view suffer from the same issue? Why will this view contain columns where 'posted' column is NULL.
>
> Please explain.
>
> -- sanjay raj, August 25, 2005

Re Sanjay Raj's comment: Well it has been almost 9 years since you asked the question, so probably by now you've either found the answer elsewhere or lost interest in databases altogether. But since others might be confused and no one else has responded I figured I would. This view-on-the-fly (ad_view) is a list of all ads with dates after 1999-01-01. This is then outer joined to the users table ("where users.user_id = ad_view.user_id(+)") so that users whose ids are included in ad_view will have their ads next to them, but all other users (including both those with no ads and those whose most recent ads were before 1999-01-01 and thus didn't make the cut for ad_view) will be listed with null/white space.

The difference is that rather than outer joining and then filtering for date, which breaks it because you're filtering out the nulls, you're instead filtering for date first by creating the view, and then outer joining, so you don't touch the nulls after they're created.

-- Dan Cusher, June 10, 2014

Add a comment