Late-Binding Views

Max Ganz II @ Redshift Research Project

15th December 2023 (updated 1st May 2024)

Abstract

There are in Redshift two types of view, normal and late-binding. When the leader node creates a normal view, it checks at the time of creation `pg_class` for the tables and views being used by the view, and will not create the view if it refers to non-existent tables or views. Accordingly, external tables cannot be used with normal views because external tables are not present in `pg_class` and so are perceived as being non-existent. To work around this, late-binding views were created, which perform no dependency checking - they do not inspect `pg_class`. This approach has two major and three minor issues; the lack of dependency checking means blunders when creating or modifying views lead to faults which can become apparent only at a very different time and in a very different place to a very different person who has no idea what caused the fault, there are as the number and complexity of late-binding views increase critical performance problems when enumerating the columns in late-binding views such that it is possible for it to become impossible to enumerate late-binding view columns, a system table function is used to enumerates late-binding view columns and access privileges for this function are incorrectly implemented such that `syslog unrestricted` and `access system table` do not work, the lack of dependency information means no records in `pg_depend` so it is impossible to perform data lineage or inspect dependencies, and finally, late-binding views pollute `pg_class` as they look like normal views (except for a single undocumented difference). Late-binding views should due to their flaws be used for and only for their one essential purpose, when an external table must be in a view.

Contents

Introduction

Redshift originally stored all data on disk in its own nodes.

Then in July 2017 AWS introduced "Redshift Spectrum", which allowed data to be stored in S3, in what are termed *external tables*.

External tables could not be used in views, because external tables are not normal tables, and so are not present in `pg_class`, and so the leader-node (which is an enhanced Postgres 8) cannot create views because from its point of view, you're referring to a non-existent table.

Not longer after release, in September 2017, a new form of view, which could use external tables, was introduced : *late-binding views*.

This document investigates late-binding views.

Introductory Discussion

Now, with regard to views using external tables, I think the devs should have modified the leader node to understand external tables (`pg_class` has a `relkind` column, and stores data for both normal tables and normal views), and then normal views could have been used.

What was done instead was to create this new type of view which performs no dependency checking at the time of creation. As such, there are no records in `pg_attribute` (which lists columns) and no records in `pg_depend` (which lists dependencies).

(Note that the view is still listed in `pg_class`, and looks like a normal view in every respect except that `relnatts` (number of attributes, which is to say, columns) is set to 0. This is not documented.)

This also mandates then the creation of further system tables (as happened) to perform exactly the same work as the existing system tables - there are now two sets of system tables, one for normal tables and normal views, and one for external tables and another for late-binding views. This is not good..

So, where there is no dependency checking at the time of creation, what happens then with late-binding views is that the view can refer to anything, whether it exists or not; there's simply to checking when the view is created. Rather, when the view is *used*, then Redshift goes about checking for the views, tables and external tables referenced by the view, and finds out if they exist or not.

Now, with normal views, if you try to modify a table which is used by views, you cannot. The leader node won't let you - it knows there is a dependency. Unfortunately, this knowledge is not a per-column level, so you cannot modify or remove any columns at all, whether or not they're used by views.

This is awkward, because it means to modify tables which are used by views, you need to peel back the views, make the change, then re-make the views.

So what happens now, with late-binding views, is that less experienced engineers think *"ah ha! I can avoid this by using a late-binding view!"*, which they then do, and in fact they stop using normal views and use late-binding views for everything.

There are a *number* of problems with this, two of which are killers.

1. The first killer problem is that because there is now no dependency checking, it is now possible to modify the tables and views a late-binding view

depends upon, without producing any problems *at the time of the modification*.

What happens is that some other poor bastard, at a completely different time, quite possibly in a completely different team, and with no knowledge of those changes, finds *his* code stops working, because his views no longer work when they're used.

2. The second killer problem is querying information about late-binding view columns from the system tables.

   Postgres stores system information in tables. The Redshift devs, for a long time now, when they introduce new functionality, have often provided system information via a *function*. It's a row-generating function, with a veneer view on top of it, where the view simply calls the function.

   For late-binding views, we can enumerate the views themselves via `pg_class`, so that's fine, but the and the only way to obtain information about columns is via the function `pg_get_late_binding_view_cols()`.

   Problem is that this function validates late-binding views before returning information about them (and returns no information about invalid views), and this validation work has performance problems. The more views, the more complex they are, the slower it becomes, and this is non-linear; it's fine with smaller numbers, it's problematic with greater.

   You can end up in a position where this function simply does not return in a timely manner (I've had it run for over a day) and at that point you can no longer enumerate the columns of late-binding views.

   The test programme which comes with this document explores this behaviour.

3. Access privileges for the late-binding view system table function (and I think for all similar row-producing functions in the system table) have not been correctly implemented. An administrator will see all late-binding views, all other users will see their own rows only - but this is regardless of `syslog unrestricted` and the new role privilege `access system table`, both of which cause normal users to see in system tables all rows, rather than only their own rows.

4. Where late-binding views have no dependency information, there is no information about them in `pg_depend`. This makes it impossible to compute dependency information (I use this information to produce data lineage graphs). You can't parse the SQL to figure this out, because AWS do not publish a BNF for Redshift (which prevents any number of vitally useful third party tools from coming into existence). The only solution is make all the views as normal views, so that the leader node populates `pg_depend`.

5. Late-binding views are stored in `pg_class`, and look like normal views, only of course, they're different : no dependency information, no column information in `pg_attribute`. Essentially they pollute `pg_class`. It's undocumented, but you can tell them apart from normal views, because

`relnatts` for late-binding views is set to 0 (this being the count of attributes, which is to say, columns).

Test Method

A two node `dc2.large` cluster is created.

The following tests are performed;

1. A late-binding view is created, which uses a single table, where that table does not exist.

   I record the count of the number of rows returned by `pg_catalog.pg_get_late_binding_view_cols()`.

   The missing table is then created, and the count of the number of rows is recorded again.

2. A single test table, with a single column, is created.

   The test then loops, creating one late-binding view per iteration of the loop, where the view is simply a `select *` from the single test table (so the views are identical except for their names). After every 100 views, the time taken for the function `pg_catalog.pg_get_late_binding_view_cols()` to return is recorded.

   This is repeated up to 2000 views.

3. A single test table, with ten columns, is created.

   The test then loops, creating one late-binding view per iteration of the loop, where the view is simply a `select *` from the single test table (so the views are identical except for their names). After every 100 views, the time taken for the function `pg_catalog.pg_get_late_binding_view_cols()` to return is recorded.

   This is repeated up to 2000 views.

4. The test loops, creating one test table with one column and one late-binding view per iteration, where the late-binding view is a `select *` from the table which has been created in that iteration.

   After every 100 views, the time taken for the function `pg_catalog.pg_get_late_binding_view_cols()` to return is recorded.

   This is repeated up to 2000 views.

5. The test loops, creating two test tables with one column each and one late-binding view per iteration, where the late-binding view is a `select *` from the tables which have been created in that iteration (the tables being cross joined).

After every 100 views, the time taken for the function `pg_catalog.pg_get_late_binding_view_cols()` to return is recorded.

This is repeated up to 2000 views.

6. The test loops, creating three test tables with one column each and one late-binding view per iteration, where the late-binding view is a `select *` from the tables which have been created in that iteration (the tables being cross joined).

   After every 100 views, the time taken for the function `pg_catalog.pg_get_late_binding_view_cols()` to return is recorded.

   This is repeated up to 2000 views.

7. Now the first killer. The test initializes by creating a single test table with a single column, and a late-binding view which is a `select *` from that table.

   The test then loops, creating a late-binding view per iteration, where that view is a `select *` from the *previous late-binding view*.

   After every 50 views, the time taken for the function `pg_catalog.pg_get_late_binding_view_cols()` to return is recorded.

   This is repeated up to 350 views.

8. Finally, the real killer. The test initializes by creating a single test table with a single column, and a late-binding view which is a `select *` from that table.

   The test then loops, creating one table and one late-binding view per iteration, where that view is a `select *` from a cross-join of the table created in the iteration and the previous late-binding view.

   After every 50 views, the time taken for the function `pg_catalog.pg_get_late_binding_view_cols()` to return is recorded.

   This is repeated up to 350 views.

In all cases, the time taken for the function `pg_catalog.pg_get_late_binding_view_cols()` is obtained from the system table `stl_scan`, and the time is for and only for the single `scan` step which is reading from that function.

Note that to be able to do this, a dummy table has to be created, which has a single row inserted, and the output from the function is cross joined to that table; this causes the query to run in WLM, and so be logged in the system tables. Without this, it would be a leader node SQL query (as opposed to DDL/DML), and so would be seen only in `stv_recents`, but this system table shows only the most recent 100 queries, and given how long some of the test queries take, they would no longer be in the table by the time I came to look for them.

In all cases, the function `pg_catalog.pg_get_late_binding_view_cols()` is called five times, with the slowest and fastest times being discarded, and the mean and standard deviation of the remaining three times being given as the result.

Results

See Appendix A : Raw Data Dump for the Python `pprint` dump of the results dictionary.

All durations are in seconds.

All durations are taken from the system table `stl_scan` and are for and only for the single step which is reading from `pg_catalog.pg_get_late_binding_view_cols()`.

All iterations have the reading step performed five times, with the slowest and fastest time elided, and the mean and standard deviation computed from the three remaining times.

Invalid/Valid

Invalid	Valid
0	1

One Table One Column Many Views

Iteration	Mean	StdDev
250	0.027	0.000
500	0.062	0.007
750	0.089	0.000
1000	0.134	0.011
1250	0.167	0.002
1500	0.221	0.014
1750	0.266	0.006
2000	0.413	0.001
2250	0.503	0.016
2500	0.582	0.003
2750	0.707	0.007
3000	0.787	0.014

One Table Ten Column Many Views

Iteration	Mean	StdDev
250	1.364	0.009
500	1.510	0.045
750	1.590	0.011
1000	1.739	0.015
1250	1.853	0.009
1500	2.055	0.033
1750	2.138	0.015
2000	2.298	0.002
2250	2.468	0.026
2500	2.622	0.020
2750	2.803	0.010
3000	2.954	0.007

One Table per View

Iteration	Mean	StdDev
250	1.437	0.029
500	1.624	0.012
750	1.925	0.074
1000	2.107	0.040
1250	2.358	0.009
1500	2.607	0.043
1750	3.092	0.131
2000	3.219	0.034
2250	3.645	0.034
2500	3.972	0.046
2750	4.289	0.011
3000	4.859	0.129

Two Table per View

Iteration	Mean	StdDev
250	3.994	0.072
500	4.622	0.093
750	5.159	0.029
1000	5.818	0.068
1250	6.624	0.046
1500	7.787	0.369
1750	8.474	0.043
2000	9.511	0.315
2250	10.507	0.103

Iteration	Mean	StdDev
2500	11.849	0.075
2750	13.030	0.107
3000	14.497	0.226

Three Table per View

Iteration	Mean	StdDev
250	4.281	0.030
500	5.260	0.086
750	6.398	0.155
1000	7.770	0.113
1250	9.423	0.101
1500	11.302	0.580
1750	12.521	0.067
2000	14.200	0.138
2250	16.540	0.496
2500	18.411	0.058
2750	20.692	0.202
3000	23.543	0.367

One Table, Previous View per View

Iteration	Mean	StdDev
50	4.315	0.007
100	7.602	0.079
150	14.850	0.092
200	27.626	0.349
250	46.746	0.336
300	75.797	0.324
350	113.876	0.199

One Table and Previous View, per View

Iteration	Mean	StdDev
50	5.378	0.045
100	14.567	0.541
150	37.454	0.279
200	91.676	0.376
250	178.720	1.438
300	333.567	2.121

Iteration	Mean	StdDev
350	694.375	12.310

Discussion

Invalid/Valid

The test creates a late-binding view, which use one normal table, where that normal table does not exist and so the view is invalid, and calls `pg_get_late_binding_view_cols()`, which returns no rows.

I then make the view valid, by creating the table the view uses, and then the number of rows returned is 1 (there is a single column in the table, and the view uses only that table).

We see that `pg_get_late_binding_view_cols()` in fact is validating late-binding views, because it returns and only returns valid late-binding views. A view (and its columns) which is using a non-existent table or view, and so is invalid, is *not* listed.

The fact that validation is occurring means the function has to do significant work for every late-binding view; it's not simply listing the views.

Note that this behaviour contradicts the documentation, which states;

> *Returns the column metadata for all late-binding views in the database*

This inaccuracy is wholly in line with my general experience and view of the RS docs.

Note it is still possible to enumerate invalid late-binding views - you just can't enumerate their columns - as they show up in `pg_class`. They look exactly like normal views, except they have no columns, so `relnatts` is 0. Normal views always have one or more columns, and so for them `relnatts` is 1 or greater.

One Table One Column Many Views

In this test, a single table with a single column is made, and then 3000 late-binding views, each a `select *` from that single table.

With 250 views, `pg_get_late_binding_view_cols()` returns in 0.027s.

With 2500 views, 0.413s, which is non-linear; linear would give us 0.27s.

However, even with the maximum number of views, 3000, the time taken is 0.79s, which presents absolutely no problems at all.

One Table Ten Column Many Views

This test is identical to the previous test, except the table now has ten columns, rather than one.

Increasing the number of columns has a clear impact on performance.

250 views goes from 0.027s to 1.364s

2500 views goes from 0.413s to 2.622s.

We see then the number of columns matters.

This makes sense, as the `pg_get_late_binding_view_cols()` returns every column of every late-binding view - it does *not* return simply one row per view. The more columns, the more rows must be returned, the longest the function takes.

In any event, we're still looking at a couple of seconds, so no problems.

One Table per View

In this test, we're back to one column per table, and now produce one table and then one view, and the view is a `select *` from that table, so there end up being 3000 tables made, and 3000 views.

With 250 views, 1.44s.

With 2500 views, 3.97s.

Now this is interesting. We're back to one column per table, but we see times are much slower than the first test.

Here we see that the function is performing processing on a per-view basis, and that work relates to the tables (and certainly also the views, but that's not been tested here) which are used by a late-binding view, which fits in with the existing idea that the views are being validated.

We see also this work, for a single table being used in a view, is more expensive than going from one to ten columns.

We also see the work being done is definitely non-linear.

With 250 views, 1.44s.

With 500 views, 1.62s.

With 2750 views, 4.289s.

With 3000 views, 4.859s.

Nevertheless, even with 3000 views, we're still looking at less than five seconds, so no problems.

Two Tables per View

This test is the same as the previous test, except now we create two tables each time, and the view uses both.

This has quite a big impact on performance.

250 views goes from 1.44s to 3.99s.

2500 views goes from 3.972s to 11.449s

We're looking roughly at a tripling of the time taken for the function to return, for a doubling of the number of tables. Performance is better than this with a low number of views, but at 1500 we're at about triple and performance then remains about three times slower.

Still, even now, 3000 views is 14.5s - still viable - but we can begin now to think about possible being worried. We can imagine a real-world system with say a thousand views, but where each view uses many tables (and other views, too).

Three Tables per View

This test is the same as the previous test, except now we create three tables each time, and the view uses both.

The results are in line with the two table per view test; except now about four times slower, not three times slower.

250 views goes from 1.44s to 4.28s.

2500 views goes from 3.97s to 18.4s

One Table, Previous View per View

In this test, we start with a single table with one column, and a single view, which is a `select *` from that table.

Now though each time we make a new view, it is a `select *` from the *previous view*.

Remember that we've seen already that `pg_get_late_binding_view_cols()` looks to be performing work on the tables (and likely enough the views also, but that's not yet been tested) in each late-binding view.

If we imagine that the function when it processes one view, must then process every view before that view, we see that for example If we have made 5 views, the 5th view requires processing itself and the four views before that (so five views in total), that the 4th view requires processing itself and the three views before that (so four views in total), and so on.

So the total number of views being processed is the mean of the sum of the integers from 1 to the number of views, inclusive both ends, multiplied by the number of views.

So five views gives us a mean of (5+4+3+2+1) = 15 views being processed.

This then for our test range of 50 to 350 gives us;

Views	Total Views Processed
50	1225
100	4950
150	11175
200	19900
250	31125
300	44850
350	61075

So, 50 views gives 1225 views being processed, which takes 4.32s.

We can compare this quite closely to the 1250 view values we have fro previous tests, which for the one single column table per view test, gives 2.36s

There seems to be some overhead involved in travelling down the views, which is to be expected.

With 350 views, the function takes 113.88s.

In any event, this result shows work is being done per view being listed by the function; for if this were not the case, the time taken for 50 views would be the same regardless of the number of views or tables used by each view, and that obviously is not what we see.

One Table and Previous View, per View

Now here, the same test as before, except it's no longer a single table being used by the first view and all views using and only using the previous view; rather, we again start with a single table and single view, but each time we make a view, we now also make a table, and each view uses that table and the previous view.

So we still have each view using the previous view, but we mix in a new table for each view. It might be views and tables require different amounts of work to validate, but in any event, we are certainly adding one table per view to the workload in the previous test.

The performance slow down is very non-linear; it's about one second with 50 views, then about 2x with 100 views, then about 6x with 360 views.

With 350 views, we've gone from 113.88s to 694.38s, which is about 11.5 minutes.

Conclusion

Performance of the function `pg_catalog.pg_get_late_binding_view_cols()` depends on two factors, one minor and one major.

The minor factor is the total number of rows being returned. The function returns one row for every column in every late-binding view, so the more views, and the more columns in the views, the more rows must be returned; the more rows must be returned, the more time the function takes to return. There is simply more data to process.

The major factor is the number of tables and views being used in all late-binding views, because the function is validating the late-binding views (and returning rows only for valid views). This work appears to be non-linear in terms of performance; going from say 50 views to 100 is much faster than going from say 1950 to 2000 views.

I hear from a number of reports in the field of clients using this function and from the users point of view, it has frozen up; it simply never returns.

I've had this experience myself; I issued a query calling this function, on a client's cluster with about 600 complex late-binding views and over a day later, the query was still running.

In the tests here, this being a fairly quick investigation, I've not tried to actually replicate such behaviour, because doing so would take quite a lot of work, and because given all the other issues using late-binding views - critically, that changing late-binding views can and so will induces bugs which distant others discover at a different time and place - it's just not necessary; late-binding views should be avoided anyway.

All in all, I am of the view that late-binding views are a trap.

They have a number of serious issues, but are tempting to inexperienced developers, and AWS as ever provide absolutely no meaningful documentation or any advice or warnings.

Use late-binding views for and only for external tables; which is to say, use them only when you actually *must*.

Revision History

v1

- Initial release.

v2

- Added “About the Author”. made site name in title a link, and made each chapter start a new page.

Appendix A : Raw Data Dump


```
{'eu-central-1': {'dc2.large': {'Misc Tests': {'Invalid/Valid': (0, 1)},
                                'Timed Tests': {'One Table One Column Many Views': {250: [

                                                                                    500: [

                                                                                    750: [

                                                                                    1000:

                                                                                    1250:

                                                                                    1500:

                                                                                    1750:
```

2000:

2250:

2500:

2750:

3000:

'One Table Ten Column Many Views': {250: [

500: [

750: [

1000:

1250:

1500:

1750:

2000:

2250:

2500:

2750:

3000:

'One Table and Previous View, per View': {

```
'One Table per View': {250: [1.389881,
                             1.407971,
                             1.42682,
                             1.475685,
                             1.563982],
                       500: [1.593606,
                             1.61216,
                             1.619976,
                             1.640055,
                             1.776321],
                       750: [1.811689,
                             1.842561,
                             1.911267,
                             2.022408,
                             2.026702],
                       1000: [2.044837,
                              2.065585,
                              2.093549,
                              2.161705,
                              2.213703],
                       1250: [2.311234,
                              2.345591,
                              2.360332,
                              2.366805,
                              2.594447],
                       1500: [2.557307,
                              2.574542,
                              2.57859,
                              2.667481,
```

```
2.806228],
1750: [2.876321,
       2.951699,
       3.057304,
       3.267518,
       5.350069],
2000: [3.162246,
       3.171674,
       3.234388,
       3.250595,
       3.413778],
2250: [3.580311,
       3.605738,
       3.641177,
       3.688019,
       3.744305],
2500: [3.87986,
       3.908394,
       3.99215,
       4.016355,
       4.117444],
2750: [4.268914,
       4.274921,
       4.290679,
       4.302014,
       4.512915],
3000: [4.72491,
       4.734308,
       4.807244,
       5.036611,
       5.285265]},
'One Table, Previous View per View': {50:

100:

150:

200:
```

```
                                                                  250:

                                                                  300:

                                                                  350:

'Three Tables per View': {250: [4.187904,
                                4.240784,
                                4.288567,
                                4.312924,
                                4.730891],
                          500: [5.150703,
                                5.159228,
                                5.250595,
                                5.368963,
                                5.723796],
                          750: [6.23971,
                                6.284882,
                                6.292057,
                                6.617936,
                                6.906559],
                          1000: [7.598545,
                                 7.680021,
                                 7.700044,
                                 7.928654,
                                 8.188016]
                          1250: [9.261514,
                                 9.338888,
                                 9.365387,
                                 9.565782,
                                 9.691725]
                          1500: [10.699206
                                 10.799106
                                 10.993342
                                 12.11421,
                                 14.272023
                          1750: [12.125911
                                 12.42644,
                                 12.560599
                                 12.575457
```

```
                                 13.200647
                           2000: [13.922375
                                 14.09685,
                                 14.107692
                                 14.395242
                                 14.869925
                           2250: [16.027581
                                 16.090543
                                 16.299229
                                 17.231329
                                 17.270747
                           2500: [17.989508
                                 18.363064
                                 18.377968
                                 18.492862
                                 19.119542
                           2750: [20.267543
                                 20.461733
                                 20.660327
                                 20.953699
                                 21.006646
                           3000: [22.906005
                                 23.113274
                                 23.504704
                                 24.010054
                                 24.265129
'Two Tables per View': {250: [3.872256,
                              3.913594,
                              3.980403,
                              4.088029,
                              4.361544],
                        500: [4.407811,
                              4.517345,
                              4.60585,
                              4.74226,
                              4.985494],
                        750: [5.075409,
                              5.118114,
                              5.175447,
                              5.18364,
                              5.654609],
                        1000: [5.695226,
                               5.721563,
                               5.859109,
                               5.872586,
                               6.35847],
                        1250: [6.488244,
                               6.583543,
                               6.601035,
                               6.688612,
```

```
       6.996788],
1500: [7.249985,
       7.507182,
       7.545313,
       8.308091,
       8.731941],
1750: [8.178887,
       8.417119,
       8.482488,
       8.522281,
       8.931544],
2000: [9.208271,
       9.235163,
       9.345106,
       9.951715,
       10.351686],
2250: [10.33257,
       10.365894,
       10.545784,
       10.608162,
       11.249779],
2500: [11.429371,
       11.786412,
       11.805983,
       11.953599,
       12.483635],
2750: [12.834082,
       12.891457,
       13.04483,
       13.153395,
       13.4971],
3000: [14.294209,
       14.320388,
       14.35472,
       14.815526,
       14.871431]}
```

About the Author

I am a C programmer - kernel development, high performance computing, networking, data structures and so on.

I read the C. J. Date book, the classic text on relational database theory, and having learned the principles, wrote a relational database from scratch in C, which purely by chance set me up quite nicely for what came next, moving into data engineering in late 2011, when I joined as the back-end engineer two friends in their startup.

In that startup, I began using Redshift the day it came out, in 2012 (we had been trying to get into the beta programme).

We were early, heavy users for a year and a half, and I ending up having monthly one-to-one meetings with one of the Redshift team managers, where one or two features which are in Redshift today originate from suggestions made in those meetings, such as the distribution style `ALL`.

Once that was done, after a couple of years of non-Redshift data engineering work, I returned to Redshift work, and then in about mid-2018 contracted with a publisher to write a book about Redshift.

The book was largely written but it became apparent I wanted to do a lot of things which couldn't be done with a book - republish on every new Redshift release, for example - and so in the end I stepped back from the contract and developed the web-site, where I publish investigation into, and ongoing monitoring of, Redshift.

So for many years now I've been investigating Redshift sub-systems full-time, one by one, and this site and these investigations are as far as I know the and the only source of this kind of information about Redshift.

Redshift Cluster Cost Reduction Service

I provide consultancy services for Redshift - advice, design, training, getting failing systems back on their feet pronto, the usual gamut - but in particular offer a Redshift cluster cost reduction service, where the fee is and only is one month of the savings made.

Broadly speaking, to give guidance, savings are expected fall into one of two categories; either something like 20%, or something like 80%. The former is

for systems where the business use case is such that Redshift cannot be operated correctly, and this outcome requires no fundamental re-engineering work, the latter is for systems where Redshift can be operated correctly, and usually requires fundamental re-engineering work (which you may or may not wish to engage in, despite the cost savings, in which case we're back to the 20%).

Details and contact information are on the web-site.