

WHY EQUIJOINS MATTER!

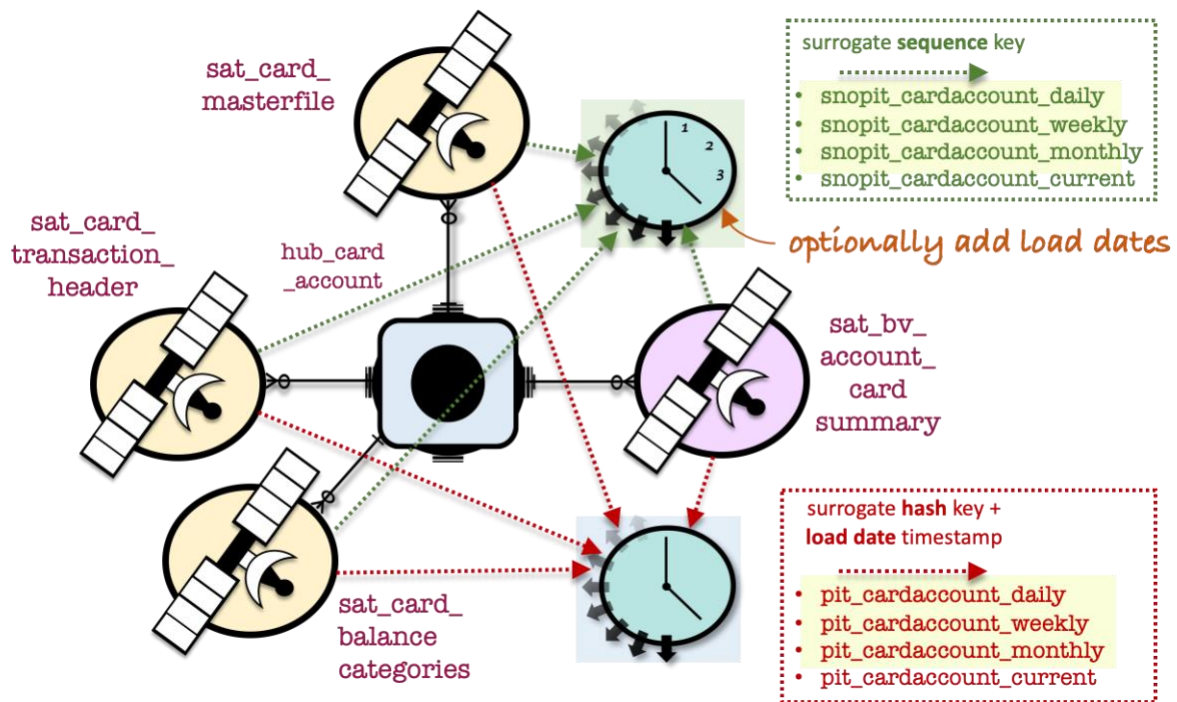


The Scene

In March 2021 I proposed a slight **modification** to the **standard** satellite table structure and point-in-time (PIT) table **snapshot process** that would take advantage of [Snowflake's](#) interpretation of a massively parallel platform ([MPP](#)) architecture, today I will provide evidence of what these modifications can do.

*Please familiarise yourself with **both** articles before proceeding with the rest of **this** article.*

- Link to March 2021 article: bit.ly/3dn83n8
- Link to why equijoins matter in every analytics platform: bit.ly/3vjTXdg



Simulate **daily load** to four satellites and a common hub table for a **six month period** (181 days).

Starting 1 Jan 2021, randomise account number creation and change attributes in satellite tables meaning that some accounts will reference the ghost record in a satellite table to start with while others will start with a reference to actual data. After all a hub table in this model cannot have a business key without at least **one** of the surrounding satellites supplying that business key!

```
set my_rec_count = 150000;
```

Account_ID function: lpad(uniform(1, \$my_rec_count, random()), 10, '0') as account_id

Table generator function: table(generator(rowcount => \$my_rec_count))

Tablename	Record count	Column count	Size
hub_account	150,000	9	4MB
sat_card_masterfile	17,253,768	613	3GB
sat_card_balancecategories	17,259,059	419	2GB
sat_card_transaction_header	17,257,668	12	875.3MB
sat_bv_account_card_summary	150,001	10	4.4MB

Build **logarithmic** PIT tables taking a snapshot of the surrogate hash keys and load dates and logarithmic SnoPIT tables taking a snapshot of the surrogate sequence ids at the following intervals:

- Daily, only benefit of this PIT is providing an **equijoin** between the PIT satellite and surrounding satellite tables. Realistically we wouldn't build a PIT like this!
- Weekly, simulating the business case for having **Monday morning** data ready for quick access
- Monthly, simulating an **end-of-month** state of the **business objects**.

- Current, equijoin to the **current state** of the business objects

Tablename	Record count	Column count	Size
pit_cardaccount_daily	27,297,309	11	2.7GB
pit_cardaccount_weekly	3,900,000	11	400MB
pit_cardaccount_monthly	900,000	11	92.3MB
pit_cardaccount_current	150,000	11	15.4MB
snopit_cardaccount_daily	27,297,309	7	929MB
snopit_cardaccount_weekly	3,900,000	7	132.3MB
snopit_cardaccount_monthly	900,000	7	30.5MB
snopit_cardaccount_current	150,000	7	5.1MB

Mod #1: adding autoincrement / identity column to satellite tables

Upon satellite table creation add a new **autoincrement/identity** column called **DV_SID** (Data Vault Surrogate ID) with a start number value of zero and a step number increment of 1 (see: bit.ly/3h2VbF5). Remember, this column carries **no relation** to any other data vault table, it is however **temporal** because a new value is assigned for every record and **not** every new business key (or unique relationship in the case of a link-satellite table). Every satellite will have their own **independent** DV_SID.

```
create table rawvault.sat_card_masterfile
(dv_tenantid varchar(20) not null
, dv_hashkey_hub_account binary(20) not null
, dv_loaddate datetime not null
, dv_applieddate datetime not null
, dv_recsource varchar(100) not null
, dv_hashdiff binary(20) not null
, dv_sid int autoincrement(0, 1)
, card_type varchar(1)
, card_balance decimal
, card_status varchar(1)
, credit_limit decimal
, ...
```

Insert the **Ghost** record immediately after table creation; every satellite table must have one. And since it is the first record the DV_SID value will be 0.

```
insert into rawvault.sat_card_masterfile (dv_tenantid, dv_hashkey_hub_account,
dv_loaddate, dv_applieddate, dv_recsource, dv_hashdiff)
select "
, to_binary(repeat(0, 20))
, to_timestamp('1900-01-01 00:00:00')
, to_timestamp('1900-01-01 00:00:00')
```

```

, 'GHOST'
, to_binary(repeat(0, 20))
, 'GHOST'
, 'GHOST'
;

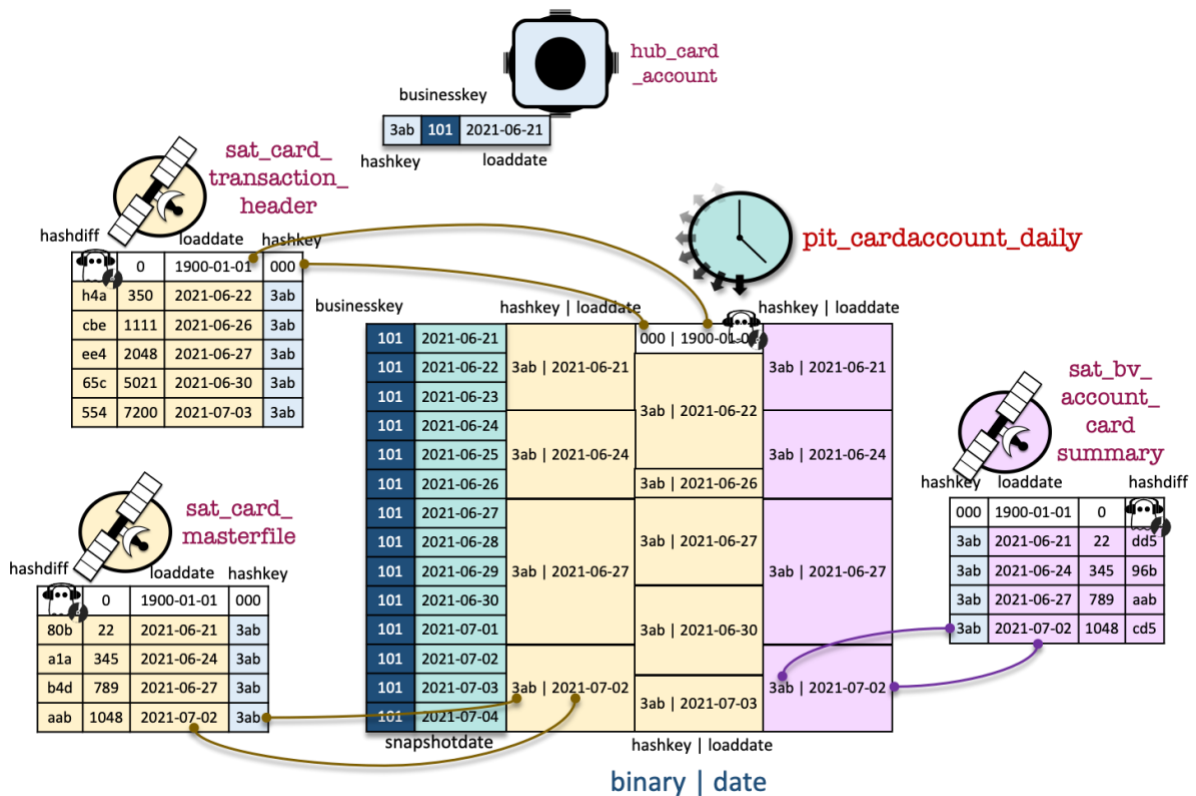
```

Populate with six months' worth of test data.

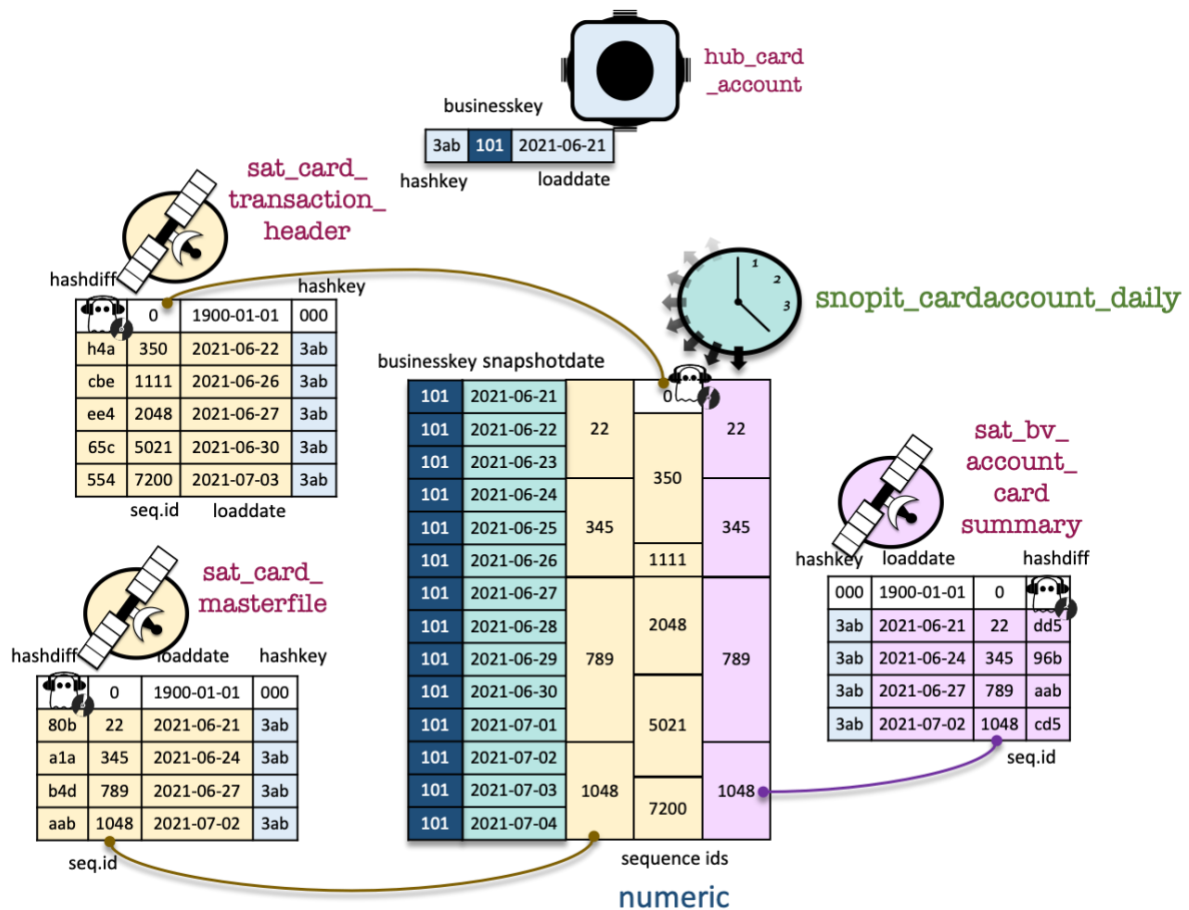
Mod #2: building PIT table to return the sequence id

PIT tables are designed to take advantage of **physical index-joins** between the PIT table itself and its surrounding satellite tables, it is as so as described in Dan Linstedt's "Building a Scalable Data Warehouse with Data Vault 2.0".

A typical PIT table would look like this



Since we do not have indexes in Snowflake but instead have a **zone-map** for every **min/max** value for **every** column for **every** micro-partition we then model the PIT to take advantage of this change, *ala SnoPIT*.



If you noticed, SnoPIT now looks more like a Fact table, more on this later!

The Evidence

Of what use is this new structure if we do not provide evidence of its effectiveness...



Exhibit A: Clustering overlap

Snowflake provides the tools to analyse clustering information for your tables

```
select system$clustering_information('<table_name>', '(<column1>, <column2>...');
```

See: bit.ly/3h2RvTx

Average overlaps: overlapping micro partitions, high number is **bad**

Average depth: micro partitions depth, high number is **bad**

Tablename: sat_card_masterfile, total partitions: 1453, *wide and deep table*

Column(s)	Average overlaps	Average depth
dv_hashkey_hub_account	1452.0	1453.0
dv_hashkey_hub_account, dv_loaddate	1452.0	1453.0
dv_loaddate	0	1.0
dv_sid	6.9897	7.9897

Thanks to “randomness” of the hash key value the values appear scattered across micro-partitions. If DV_LOADDATE is considered alone the total constant partition count would be a whopping 1452! But that is expected for a very wide and deep table!

Tablename: sat_card_transaction_header, total partitions: 404, *thin and deep table*

Column(s)	Average overlaps	Average depth
dv_hashkey_hub_account	643.0	644.0
dv_hashkey_hub_account, dv_loaddate	643.0	644.0
dv_loaddate	9.4876	7.1149
dv_sid	9.5062	7.1335

In a much thinner table DV_SID has very similar statistics to DV_LOADDATE column!

Tablename: sat_bv_card_account_summary, total partitions: 3, *thin and shallow table*

Column(s)	Average overlaps	Average depth
dv_hashkey_hub_account	2.0	3.0
dv_hashkey_hub_account, dv_loaddate	2.0	3.0
dv_loaddate	0.6667	1.6667
dv_sid	0.0	1.0

If all your tables are this small SNoPIT and even the traditional PIT is not for you!

Tablename: pit_cardaccount_daily, total partitions: 1446

Column(s)	Average overlaps	Average depth
sat_card_masterfile_dv_hashkey_hub_account, sat_card_masterfile_dv_loaddate	1443.0014	1444.0014
sat_card_transaction_header_dv_hashkey_hub_account, sat_card_transaction_header_dv_loaddate	1445.0	1446.0
sat_bv_account_card_summary_dv_hashkey_hub_account, sat_bv_account_card_summary_dv_loaddate	1445.0	1446.0

Tablename: snopit_cardaccount_daily, total partitions: **406**

Column(s)	Average overlaps	Average depth
sat_card_masterfile_dv_sid	54.064	32.5296
sat_card_transaction_header_dv_sid	53.1872	32.1429
sat_bv_account_card_summary_dv_sid	405.0	406.0

DV_SID will have high cardinality, but it is **not** scattered, it is linear; therefore, when selecting the columns to join on Snowflake will know which micro-partitions fall into the min/max (zone map) range on **both** sides of the join query. It also means that **no** clustering columns should be selected for data vault tables otherwise you lose the natural clustering in load order!

Tablename: pit_cardaccount_weekly, total partitions: 204

Column(s)	Average overlaps	Average depth
sat_card_masterfile_dv_hashkey_hub_account, sat_card_masterfile_dv_loaddate	207.0	208.0
sat_card_transaction_header_dv_hashkey_hub_account, sat_card_transaction_header_dv_loaddate	207.0	208.0
sat_bv_account_card_summary_dv_hashkey_hub_account, sat_bv_account_card_summary_dv_loaddate	207.0	208.0

Tablename: snopit_cardaccount_daily, total partitions: **65**

Column(s)	Average overlaps	Average depth
sat_card_masterfile_dv_sid	8.7692	6.4923
sat_card_transaction_header_dv_sid	8.5231	6.3231
sat_bv_account_card_summary_dv_sid	64.0	65.0

The BV column has the same value per snapshot in each partition!

Tablename: pit_cardaccount_monthly, total partitions: 48

Column(s)	Average overlaps	Average depth
sat_card_masterfile_dv_hashkey_hub_account, sat_card_masterfile_dv_loaddate	47.0	48.0
sat_card_transaction_header_dv_hashkey_hub_account, sat_card_transaction_header_dv_loaddate	47.0	48.0
sat_bv_account_card_summary_dv_hashkey_hub_account, sat_bv_account_card_summary_dv_loaddate	47.0	48.0

Tablename: snopit_cardaccount_monthly, total partitions: **17**

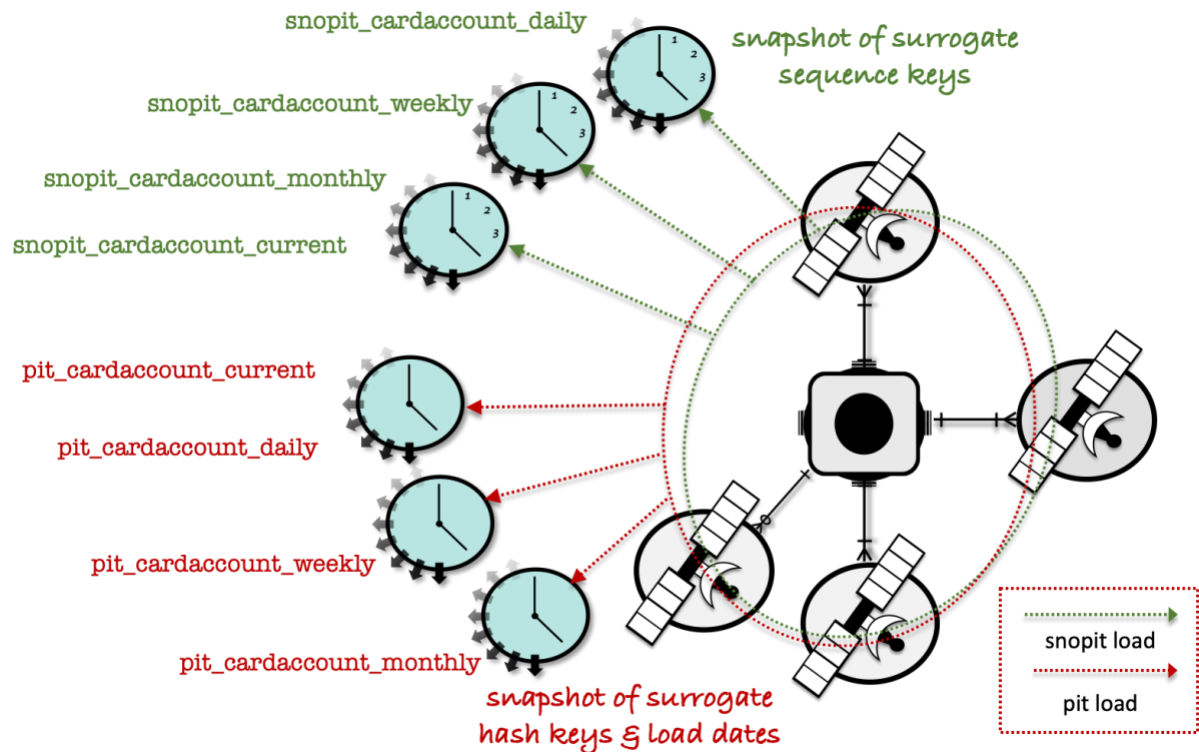
Column(s)	Average overlaps	Average depth
sat_card_masterfile_dv_sid	2.9412	3.4706
sat_card_transaction_header_dv_sid	2.9412,	3.4706
sat_bv_account_card_summary_dv_sid	16.0	17.0

The snapshots have gotten small enough that it doesn't even matter how the tables are clustered! *Recall the table sizes in the sample information above!*

More on clustering depth: bit.ly/2SzCAH3

Exhibit B: Join performance

PIT table construction involves the use of complicated join criteria, especially in the **traditional** method (cross-join). PIT tables are built for this reason, to eliminate the use of LEFT JOIN conditions in data vault queries for the end-user and information marts. If a satellite does not yet have a record for a business key but another satellite table does then an equijoin will not return any records (refer to Ghost record article above). It does however mean that this work is left to an automation framework to manage these PIT tables. An alternate pattern would be to run a **conditional multi-table inserts** (see: bit.ly/3h97Ypz) to related logarithmic PIT table structures, this ensures that there is a single source for all PIT tables and if the condition is met then the target table is populated.



Sample code:

```
insert all
when (dayname($my_loaddate) = 'Mon')
and (select count(1) from queryassistance.pit_cardaccount_weekly tgt where
tgt.snapshotdate=src_snapshotdate) = 0 then
into queryassistance.pit_cardaccount_weekly
...
... insert code ...
...
when ($my_loaddate = last_day(to_date($my_loaddate)))
and (select count(1) from queryassistance.pit_cardaccount_monthly tgt where
tgt.snapshotdate=src_snapshotdate) = 0 then
into queryassistance.pit_cardaccount_monthly
...
... insert code ...
...
... etc... join for PIT & SnoPIT
...
```

This will reduce the PIT build time but what of querying the PIT table in place of the hub or link table, let's observe some usage statistics. Remember that the intention in Data Vault 2.0 is to deliver Information Marts over PIT tables as **views**.



To make it a fair comparison between the two methods the following statements were run in a Snowflake session:

```
alter session set USE_CACHED_RESULT = FALSE;
alter warehouse vault suspend;
```

- The first statement disables resultset cache for the session (see: bit.ly/3w4W58e)
- The second statement flushes out the virtual warehouse cache, it is run after every statement

Each statement will now utilize remote storage, an apples-to-apples comparison



Select using Daily PIT, both return 27.3m rows

Profile Overview Finished		Profile Overview Finished	
Total Execution Time (19.636s)		Total Execution Time (15.847s)	
 (100%)		 (100%)	
Processing	49 %	Processing	53 %
Local Disk IO	0 %	Local Disk IO	2 %
Remote Disk IO	28 %	Remote Disk IO	20 %
Network Communication	6 %	Network Communication	5 %
Synchronization	4 %	Synchronization	6 %
Initialization	12 %	Initialization	13 %
Total Statistics		Total Statistics	
IO		IO	
Scan progress	100.00 %	Scan progress	100.00 %
Bytes scanned	5.47 GB	Bytes scanned	2.72 GB
Percentage scanned from cache	0.00 %	Percentage scanned from cache	0.00 %
Bytes written to result	1.54 GB	Bytes written to result	1.44 GB
Network		Network	
Bytes sent over the network	11.04 GB	Bytes sent over the network	6.37 GB
Pruning		Pruning	
Partitions scanned	3,950	Partitions scanned	2,910
Partitions total	3,950	Partitions total	2,910

PIT

SnoPIT



Select using Weekly PITs, both return 3.9m rows

Profile Overview Finished		Profile Overview Finished	
Total Execution Time (9.729s)		Total Execution Time (9.205s)	
 (100%)		 (100%)	
Processing	48 %	Processing	64 %
Local Disk IO	0 %	Local Disk IO	0 %
Remote Disk IO	32 %	Remote Disk IO	16 %
Network Communication	7 %	Network Communication	7 %
Synchronization	5 %	Synchronization	4 %
Initialization	8 %	Initialization	9 %
Total Statistics		Total Statistics	
IO		IO	
Scan progress	99.71 %	Scan progress	98.75 %
Bytes scanned	3.13 GB	Bytes scanned	1.92 GB
Percentage scanned from cache	0.00 %	Percentage scanned from cache	0.00 %
Bytes written to result	222.85 MB	Bytes written to result	211.05 MB
Network		Network	
Bytes sent over the network	6.94 GB	Bytes sent over the network	5.88 GB
Pruning		Pruning	
Partitions scanned	2,704	Partitions scanned	2,537
Partitions total	2,712	Partitions total	2,569

PIT

SnoPIT

Select using Monthly PITs, both return 900k rows

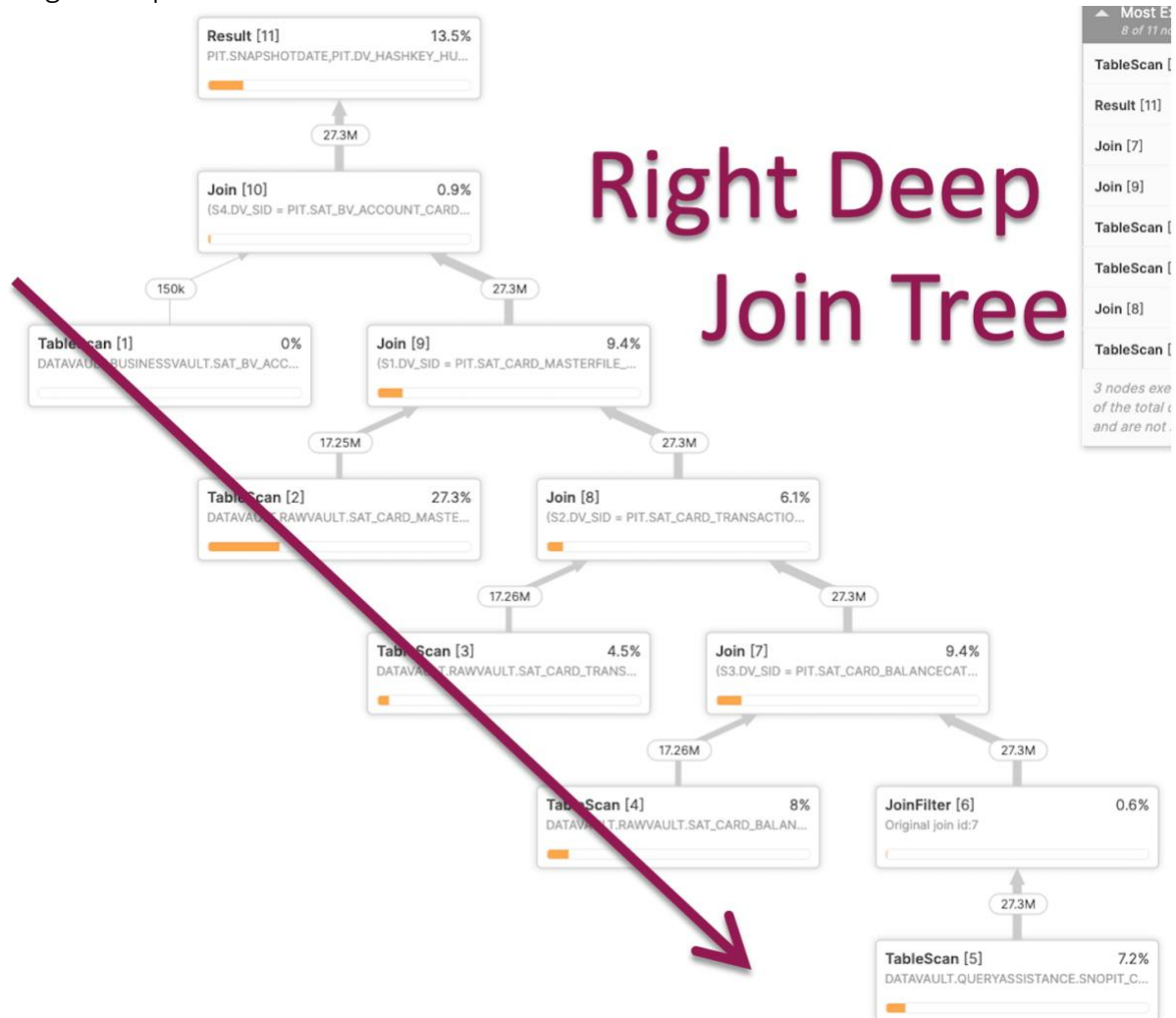
Profile Overview Finished		Profile Overview Finished	
Total Execution Time (8.502s)		Total Execution Time (7.101s)	
 (100%)		 (100%)	
Processing	47 %	Processing	61 %
Local Disk IO	0 %	Local Disk IO	0 %
Remote Disk IO	34 %	Remote Disk IO	16 %
Network Communication	7 %	Network Communication	8 %
Synchronization	5 %	Synchronization	5 %
Initialization	7 %	Initialization	9 %
Total Statistics		Total Statistics	
IO		IO	
Scan progress	78.13 %	Scan progress	57.40 %
Bytes scanned	2.13 GB	Bytes scanned	1.10 GB
Percentage scanned from cache	0.00 %	Percentage scanned from cache	0.00 %
Bytes written to result	48.26 MB	Bytes written to result	48.76 MB
Network		Network	
Bytes sent over the network	5.26 GB	Bytes sent over the network	3.73 GB
Pruning		Pruning	
Partitions scanned	1,994	Partitions scanned	1,447
Partitions total	2,552	Partitions total	2,521

PIT

SnoPIT

Right away you can see SnoPIT achieves better pruning and scans less bytes than the traditional PIT! Every statistic including execution time is better using SnoPIT! What is also of interest, every query planner graphic for the above queries has a very similar pattern...

a Right Deep Join Tree!

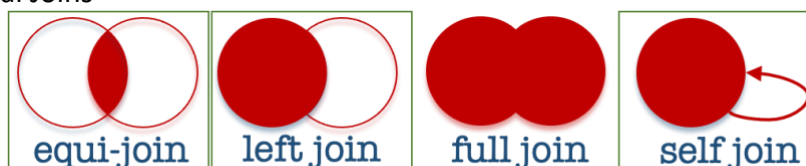


What does this mean?

Simply put, the shape of the join conditions around a centre table resembles a **Star Schema**, a centre fact table surrounded by dimension tables. To understand this further we'll briefly discuss physical joins rather than the logical joins everyone is familiar with.

Observe (see: bit.ly/2F1mrUe)

- Logical Joins



- Physical Joins (see: bit.ly/3AcreK0)

- **Sort-Merge Joins:** sorts one table, stores the sorted intermediate table, sorts the second table, and finally merges the two to form the join result. This results in a **Left Deep Join Tree**.
- **Nested Loop / Index Joins:** looks up each row of the smaller table by querying an index of the large table (if both tables have indexes, performs better than Sort-Merge). This results in a **Left Deep Join Tree**.

- **Hash Joins:** smaller table is configured in memory as a hash table, performs a row-by-row hash lookup between the larger table and the smaller table. Restricted by memory size. Involves a **build** phase to load the smaller table into memory (can have a predicate) and a **probe** phase to find matches in the larger table. When executed with an equijoin predicate, this will result in a **Right Deep Join Tree**.

Left Deep Join Tree: a join tree in which the left input of every join is the result of a previous join.

Right Deep Join Tree: a join tree in which the right input of every join is the result of a previous join, and the left child of every internal node of a join tree is a table. You still read from left to write but what happens in a right-deep tree the joins are not physically executing until you've reached the last table in the tree, the deep right corner, and note... that is the PIT table! For a deep dive into how this works visit bit.ly/3x7Gk1v.

Closing arguments



We have seen how SnoPIT outperforms traditional PITs and ladies and gentlemen we have also seen evidence of how either PIT structure through its join structure takes advantage of existing data warehouse capabilities. However, constructing PIT tables adds yet another structure for you to manage and thus you only need PIT tables when you are **not** seeing the query performance you need for your business requirements. As always keep PITs short and thin and SnoPITs produce the same output but are even thinner than traditional PIT tables.

In summary

- SnoPITs are the smaller of the right-deep-join-tree tables, like the PIT
- SnoPITs are thinner of the PIT tables and use a single join condition between itself and adjacent satellite table. The traditional PIT needs two column joins for each satellite table.
- SnoPITs use a numeric field which is the thinnest data type to join on and thus the fastest to join on, whereas traditional PITs use a binary and datetime field to join on.

- The addition to the satellite table (DV_SID column) is managed by the platform, no additional coding needed and no lookup to any other table
- SnoPITs' own columns that reference satellite tables can only increment too, unless of course an entity does not change then the zone map within the SnoPIT becomes less effective. The performance benefit is really in the much larger Satellite tables and its zone maps!
- No clustering of satellite table can be applied, doing so can break the advantage you get from utilizing Snowflake's zone-maps to *find* the micro-partitions to solve your star-join query.
- If you are using XTS for timeline correction (see: bit.ly/3y4mUdV), note that a timeline correction event does not cause micro-partition overlap! Although the timeline correction is to the timeline itself, the correction event is still seen as a new record in the satellite table (it's added to the bottom — i.e. logical vs physical timeline correction).

Is this evidence realistic? It's based on sampling of fictitious data, realistic loads to a satellite table can vary in frequency, width, and depth of satellite tables. I attempted to simulate this by creating random columns horizontally for two of the satellite tables to get some more statistics. But keep in mind that Snowflake's micro-partition structure is a hybrid store and achieves excellent compression ratios where an equivalent non-Snowflake table would consume more storage. With this PIT variation as you can see, you will most certainly see join condition improvements.

How does this methodology using PITs compare to an equivalent query without using PITs at all?

The Verdict

The conclusive evidence... without a PIT table the query ran for 8 minutes and 20 seconds... and the query plan was a Left-Deep Join Tree

With and without a Daily PIT, all return 27.3m rows

Profile Overview Finished	Without a PIT	PIT	SnoPIT
Total Execution Time (8m 19.361s)	(19.636s)	(15.847s)	
Processing	64 %	49 %	53 %
Local Disk IO	3 %	0 %	2 %
Remote Disk IO	2 %	28 %	20 %
Network Communication	22 %	6 %	5 %
Synchronization	4 %	4 %	6 %
Initialization	5 %	12 %	13 %
Total Statistics			
IO			
Scan progress	100.00 %	100.00 %	100.00 %
Bytes scanned	5.93 GB	5.47 GB	2.72 GB
Percentage scanned from cache	0.18 %	0.00 %	0.00 %
Bytes written to result	764.14 MB	1.54 GB	1.44 GB
Network			
Bytes sent over the network	121.33 GB	11.04 GB	6.37 GB
Pruning			
Partitions scanned	2,516	3,950	2,910
Partitions total	2,516	3,950	2,910

**All tests were performed using a Medium cluster single node Virtual Warehouse*

SnoPIT is my own variation of the classic PIT table – conceptually designed for Snowflake and the name is an acronym for what it is, Sequence-Number-Only-Point-In-Time table. The screengrabs are from 12 Angry men, the film is noted for referencing each character by number (juror 1 to 12) for almost the entire film, plus it represents what I was trying to achieve in this article, a trial with evidence of using a PIT in Snowflake.

The views expressed in this article are that of my own, you should test implementation performance before committing to this implementation. The author provides no guarantees in this regard.

Could you imagine Streams on Views? Stay tuned...