10 Best Practices for Deploying Data Vault on Azure SQL Data Warehouse

Table of Contents

Table of Contents....2

Introduction....3

The 10 Best Practices....4

Best Practice #1: Follow the Rules....5

Best Practice #2: Use Views to Express Interface Contracts....6

Best Practice #3: Forward Only Migration, No Changes to Existing Schema Objects....8

Best Practice #4: Version Control....10

Best Practice #5: Distribution....12

- So how does this work for a Data Vault?....12
- So which distribution strategy should you use: Replication or Hash?....13
- But what do you use as the distribution key?....13
- Let's look at an example.....14
- But what to use for distribution?....15

Best Practice #6: Automation....16

Best Practice #7: Use a Persistent Staging Layer (aka Data Lake)....19

Best Practice #8: Inserting Records into the Data Vault....22

Best Practice #9: Resource Classes....23

Best Practice #10: Rebuild Statistics after Each Major Load....25

Conclusion....25

Get started today....26

Introduction

Data Vault 2.0 is an entire system for building a modern Data Warehouse for use with enterprise-scale analytics. It is designed specifically for today's data warehousing needs – it is Agile, structured, extremely scalable, and uses patterns that support ETL code generation.

This white paper presents a number of good practices for deploying a Data Vault solution on an Azure SQL Data Warehouse platform. It assumes the reader has some knowledge of the Data Vault 2.0 system.

Getting Started with Data Vault on Azure SQL Data Warehouse

There are several good sources to learn about the Data Vault 2.0 system.

- roelentvos.com/blog for in-depth discussions on the Data Vault method
- www.data-vault.co.uk/what-is-data-vault for a selection of articles on Data Vault
- Buy yourself a copy of the comprehensive reference book 'Building a Scalable Data Warehouse with Data Vault 2.0' by Daniel Linstedt and Michael Olschimke

We recommend building a small but meaningful prototype to solidify your understanding of the Data Vault 2.0 approach and to act as a demonstrator that can be used to sell the approach to business management for what otherwise can be an abstract concept.

We recommend building a small but meaningful prototype to solidify your understanding of the Data Vault 2.0 approach

When you start to build a Data Vault 2.0 solution on an Azure SQL Data Warehouse platform you may find yourself looking for specific guidance and good practice you can apply to your project. This white paper discusses a number of these good practices, and we hope that you find them useful.

The 10 Best Practices

This paper discusses ten best practices:

1. **Follow the Rules**
   They are there for a reason, don't modify the method's rules until at least you've had a chance to use, appreciate and understand them

2. **Use Views to Express Interface Contracts**
   These construct a decoupled system and improve flexibility

3. **Forward Only Migration**
   There is only one way for a Data Warehouse schema to travel and that is forwards

4. **Version Control**
   Use Azure Repos (Git) just like any other project

5. **Distribution**
   How should Data Vault tables be distributed across nodes

6. **Automation**
   How to improve productivity and quality through using patterns and automation tools

7. **Use a Persistent Staging Layer**
   To hold a history of data feeds, improve flexibility and provide a way to recover from errors

8. **Inserting records into Azure SQL Data Warehouse**
   Hint: don't use SQL Merge

9. **Optimize Resource Class**
   Don't use SMALLRC

10. **Update statistics after each major update**

Best Practice #1

Follow the Rules

Dan Linstedt designed the Data Vault 2.0™ Data Warehouse method to deliver on goals valued by today's enterprise customers. These include Agile working, parallel load, scalability up to petabyte volumes and beyond, being technology agnostic, and enabling code generation.

The resulting architecture is both very familiar and very different.

It is familiar in its use of data layers, ETL, and star schemas as one option for data consumption.

It is different in its use of standards, re-playability of data loads, flexibility for refactoring, principles for managing data at scale (both the range and number of entities and data size), and automation.

If you build a Data Vault system you must maintain the integrity of the architecture in order to deliver on the design goals. Dan Linstedt is very clear on this. Break the Data Vault design principles at your own peril. We have seen cases where an inexperienced designer flexed or ignored a design principle and ended up with a data warehouse that couldn't scale or was more complex or costly than it needed to be to maintain and extend.

We know that architects love to tinker with and tweak systems to fit them into a wider solution. However, for your initial Data Vault system, we recommend following the design principles as closely as possible. Get some experience in the method and understand why the standards are set out the way they are and how a Data Vault works before considering any changes. What we've found is that in most cases you can construct your Data Vault system to deliver what you need while remaining within the original design principles.

In most cases you can construct your Data Vault system to deliver what you need while remaining within the original design principles.

For a primer on the principles we suggest you start with Dan Linstedt and Michael Olschimke's book: "Building a Scalable Data Warehouse with Data Vault 2.0".

Note that the Data Vault principles are not permanently fixed, they can and do change slowly over time to keep up with advances in technology, storage capacity and to cater for improvements in the Data Vault method contributed by the community. You can keep abreast of the latest developments by joining the Data Vault Alliance (www.datavaultalliance.com), a local user group (e.g. see www.ukdatavaultusergroup.co.uk), or subscribing to thought leader blogs (see www.roelentvos.com/blog and www.data-vault.co.uk/what-is-data-vault).

Best Practice #2

Use Views to Express Interface Contracts

Your Data Vault Data Warehouse is made up of layers. You have a staging layer, raw vault, business vault, and marts.

If you look at the data pipe end-to-end, you'll see a sequence of:

Figure 1: The chain of tables and SQL ETL involved in a Data Vault data pipeline

The whole structure is tightly coupled. For example, if the staging table were to change (say the source system renames a column) then this could 'break' the SQL used down the rest of the pipe. You'd need to amend, test and deploy a whole set of changes.

Tightly coupled systems are generally less flexible, more difficult to change and more brittle. As tightly coupled systems are considered bad practice in software engineering, you need to find a way to build a loosely coupled version of the same pipe.

Tightly coupled systems are generally less flexible, more difficult to change and more brittle.

One common practice is to build a view on each table and to write ETL against the view rather than the underlying table. You end up with a pipe something like this:

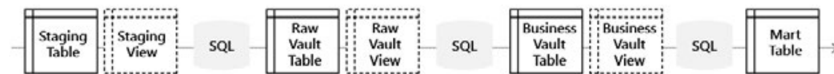

Figure 2: Adding views to the chain to buffer changes to underlying tables and to add new columns without update the original table

Why would you do this?

- It is not often explicitly called out, but each layer in the pipe offers a data interface contract to the next layer or to external systems that access that data
- Along the pipe each layer depends on the contract being honoured by the previous layer
- If you code a view to deliver the interface contract, you can decouple it from the physical table underneath, that physical table can be stored in a way that is totally different from the interface contract and the view performs the transformation needed to meet that layer's obligations
- If a physical table changes outside your control you can rescue the situation by refactoring just the one view to ensure the next layer sees what it expects to see
- If you need to support two versions of an interface (say you have a legacy application that needs a particular version) you can write two views, one per version
- It is possible to build business rules purely as views over the raw vault table, avoiding the need to physicalise the business vault table and improving system flexibility further

There is a second and more important reason for using views. Big data systems work best when data is written once and never updated. However, in a Data Vault architecture, layers can add columns to schemas to meet the interface contract for the following layer. You can easily achieve that by using a view to add

the extra columns, meaning the underlying table never needs to be updated.

For example, the staging layer needs to add a number of hash columns (for hashdiffs) to support the ETL that populates the raw Data Vault.

If you want to add views, remember:

- Don't build a view on a view, this degrades performance
- Don't overdo it, use views where they add value
- As you are working in an Agile fashion, you can start with a table and add a view later, when needed, in a future iteration

Best Practice #3

Forward Only Migration, No Changes to Existing Schema Objects

With a Data Vault you develop your solution incrementally in an Agile fashion. Each sprint you migrate the database schema forwards only by adding new tables and views to the schema, not changing what you have. You may also drop tables containing data you no longer need.

With a Data Vault you develop your solution incrementally in an Agile fashion.

So, if you need to capture more columns of data in a satellite, you would deliver this by creating a new satellite table with those additional columns, leaving the original satellite table alone. You'd end up with two satellite tables which you can join using a business view. This makes sprint-based database increments easy to code: simply write a script to add these new table elements and drop the elements you no longer need.

While this is the ideal, sometimes you might want to tidy things up a bit through a refactoring. You might want to merge two satellites into one or replace link tables to record different levels of granularity. Why would you do this? Perhaps to make the schema easier to understand for developers, to adopt a better, more meaningful name, or to make data retrieval fractionally faster.

You still want to work with a migrate forwards approach even with refactoring. For example, if you wanted to refactor to merge two satellites into one, you would achieve this by:

- Creating the new Satellite, mapping staged data to the satellite and generating SQL to load the satellite.
- Stopping feeds to the old satellites and starting the feed to the new satellite.
- Amending the SQL View to join the old and new satellites together to form a new business view of the combined tables.
- Note, if you want to you can run both the old and new Satellite table loads in parallel for a while and reconcile them daily to verify that everything is working fine. The SQL View would facilitate the cut over date.
- Note also, you could write further SQL Views to reconstruct the old Satellites from the new Satellite if you have an external dependency that expects those Satellites to exist.

Best Practice #4

Version Control

Data warehouse code is no different from other code. You should be using a version control application such as Git (see Azure Repos at azure.microsoft.com/en-gb/services/devops/repos).

When you work on a data warehouse project, the objects you might store in version control might include:

- **DDL code,** used to migrate the database from one release to the next. With a Data Vault this should be a forward only migration requiring the addition of new tables and occasionally dropping tables no longer required.
- **Metadata.** Mapping data linking tables from one layer to the next.
- **ETL code.** Both generated and hand-written.
- **Tests and Test Data.**
- **Infrastructure scripts** (if you are using infrastructure as code practices) that can construct a Data Warehouse environment on demand, used to create dev, test and production systems.
- **Deployment scripts** used to migrate the entire solution from one version to the next (which should happen at the end of each sprint). This will invoke the DDL script (above).
- **Any operating system code or utilities,** such as PowerShell or Bash scripts.
- **Documentation.**

Your Version Control repository is at the core of your deployment pipeline and development process. It is at the heart of deploying incremental functionality delivered in each sprint into production in a controlled manner – as illustrated below.

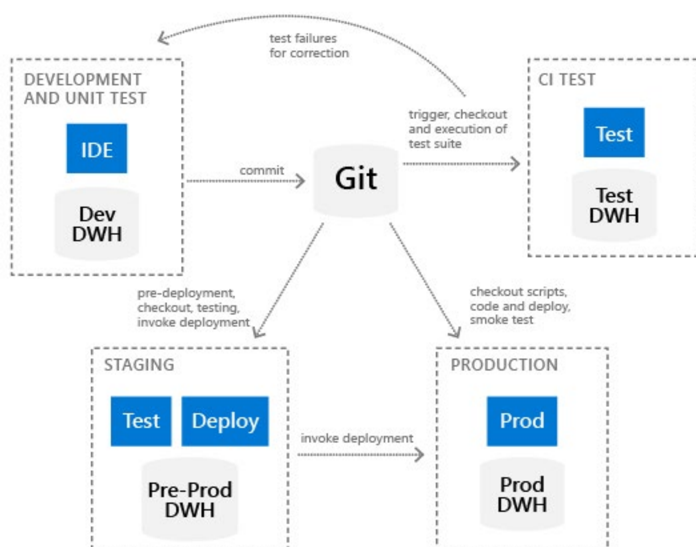

Figure 3: Using Git as the Heart of the Deployment Process

In basic terms, the process is as follows:

1. Developers work in a development environment isolated from production.
2. When they complete code and local tests they check code into Git.
3. A continuous integration service detects a change in code and launches a regression test – running a full set of tests against the new version of the code base. Failures are identified quickly for a fix from the Developer.
4. At the end of each sprint a deployment engineer takes the code base from Git into a staging environment. The engineer conducts final tests of the deployment scripts and, assuming all is well, launches production deployment.
5. The Production environment downloads deployment scripts and executes the update to bring new code and data structures online. Smoke tests are executed to confirm the upgrade.

Best Practice #5

Distribution

You have a choice of distribution methods for your Azure SQL Data Warehouse tables: 'Round Robin', 'Hash' and 'Replicated'. While Round Robin is the default distribution it is not necessarily the best option for your Data Vault.

Microsoft's guidance is to use Replicated distribution for small tables, which for SQL Data Warehouse is less than 2GB and Hash distribution for larger tables (for a discussion of these options see docs.microsoft.com/en-us/azure/sql-data-warehouse/sql-data-warehouse-tables-distribute).

So how does this work for a Data Vault?

Data Vaults have their own unique patterns and structures: they contain three main types of table – Hubs, Links and Satellites.

Common query patterns join Satellites to their parent Hub or Link and use Link tables to navigate from Hub to Hub through joins on the Link table's foreign keys.

For performance reasons you should locate data that may be retrieved together in a query on the same node. This avoids expensive data movement across nodes when the query is executed. This means that you should store Satellites on the same node as the parent Hub or Link record.

For performance reasons you should locate data that may be retrieved together in a query on the same node.

Link tables however contain references to two or more Hubs, which may be located on different nodes. To minimise cross-node traffic, you should locate the Link table with the hub with the greatest cardinality, and that Link table's satellite records as well.

So which distribution strategy should you use: Replication or Hash?

If you look at the size distribution across a Data Warehouse's data sets, you will find only a few large tables. These tend to be records from the heart of source systems, holding transaction-type or log information. Other data sets may be richer (with a sizeable number of columns) but with fewer rows, such as customer records. And still other data will rarely change, acting as reference data with a very small number of rows (such as the list of countries, channel types or customer segments). We recommend profiling your data to understand the size of each table before deciding on your replication strategy.

If you look at the size distribution across a Data Warehouse's data sets, you will find only a few large tables.

Replication should only be used for small, relatively static, tables such as reference data sets. If you've implemented these as Hubs and Satellites, then replicate both types of table across all nodes.

The remaining option is Hash distribution. You nominate a column to act as the distribution key, Azure SQL Data Warehouse hashes the column and uses the hash to decide which node to use to store the data.

But what do you use as the distribution key?

In order to ensure placement is consistent, the distribution column must be invariant, and you need to make sure that data typically queried together is located on the same node to avoid expensive cross-node traffic.

It is a common (but not mandatory) practice for Data Vault to use a hash surrogate primary key for Hubs and Links and a compound key for Satellites made up of the parent Hub's or Link's surrogate key and load datetime. There are all kinds of advantages in doing this with traditional Data Warehouse technology as it reduces the Data Vault tables to a pattern which in turn supports automation and, as the key is used for joins, they run more efficiently as a Binary(16) rather than in other data formats.

This isn't the case for Azure SQL Data Warehouse. It performs better if you avoid hashing and using a surrogate key: instead you should use the natural key, even if it is a compound key made up of more than one column (implemented as a new column in the table concatenating the two columns to be used for hashing). Benchmark performance measures support this strategy and offer a significant performance advantage over hashing a surrogate primary key.

Benchmark performance measures support this strategy and offer a significant performance advantage over hashing a surrogate primary key.

Let's look at an example.

You have a Hub called Taxpayer (HUB_TAXPAYER). It has a compound natural key – the tax jurisdiction and the Taxpayer Identifier for that Jurisdiction. Combined, they uniquely identify the Taxpayer.

```
CREATE TABLE [vlt].[HUB_TAXPAYER]
(
 [JURISDICTION_TIN]        nvarchar(360)    NOT NULL
,[JURISDICTION]            nvarchar(240)    NOT NULL
,[TIN]                     nvarchar(120)    NOT NULL
,[SOURCE]                  int              NOT NULL
,[LOAD_DATE]               date             NOT NULL)
WITH
(   CLUSTERED COLUMNSTORE INDEX
,   DISTRIBUTION = HASH([JURISDICTION_TIN])
);
```

Figure 4: example create table for a Data Vault Hub with a compound key

LINK tables are a little more challenging. They contain a FOREIGN KEY column for each HUB they link to, plus metadata for the SOURCE and LOAD_DATE, and, traditionally a calculated primary key which is the hash of the concatenation of the set of natural keys from the related HUBs.

Following on from the recommendation above not to use hashing for the primary key you simply declare a compound primary key as the combination of the foreign key links to the linked hubs.

But what to use for distribution?

Queries use LINKs when building a data set with data from multiple Satellites.

Such queries use the foreign key columns as the join column to link to HUB records to get access to the HUB's SATELLITES.

In order to speed things up and reduce data traffic the LINK should be located with its related HUB record.

But the LINK is connected to two (or more) HUBS and you can't store the LINK locally to both. If you distribute the LINK using the same distribution key as one of its related HUBs, then each time you make a join you will find that HUB record locally to the LINK and the other HUB record could be on any other node. To minimise traffic, it is better to distribute the LINK using the key from the HUB with the greatest number of records (greatest cardinality).

In the example below we have identified a link TAXPAYER_ACCOUNT between two hubs – the HUB_TAXPAYER discussed above and a HUB_ACCOUNT. This could model that a given taxpayer has one or more accounts, so HUB_ACCOUNT is larger than HUB_TAXPAYER. Therefore, we distribute the LINK with HUB_ACCOUNT.

```
CREATE TABLE [vlt].[LINK_TAXPAYER_ACCOUNT]
(
 [JURISDICTION]          nvarchar(240)    NOT NULL
,[ACCOUNT_NUMBER]nvarchar(50)     NOT NULL
,[TIN]                   nvarchar(120)    NOT NULL
,[SOURCE]                int              NOT NULL
,[LOAD_DATE]             date             NOT NULL)
WITH
(   CLUSTERED COLUMNSTORE INDEX
,  DISTRIBUTION = HASH([ACCOUNT_NUMBER])
);
```

Figure 5: example create table for a Data Vault Link, the distribution uses the foreign key to the Hub with the greatest cardinality

Finally, SATELLITES connect to HUB or LINK tables. SATELLITES have a compound key, the parent HUB or LINK's primary key plus a load_datetime. It makes sense to distribute these in the same way as their parent record – by the HUB's primary key, or the LINK's foreign key used for the distribution.

Finally, if the LINK_TAXPAYER_ACCOUNT has Satellites then distribute these using the same hash key as the parent LINK table. To do that you must include the distribution column (or compound column) in the Satellites as an additional metadata item.

Best Practice #6

Automation

Automation gives your project a real productivity boost and is considered a good practice in any modern data warehouse service.

But what is automation? We define it as the ability to generate as much of the code as possible from metadata that describes the operation of the data warehouse.

Let's take an example.

We've shown that the Data Vault architecture reduces your data model to a small number of components – mainly Hubs, Links, and Satellites.

And the SQL used to load the data warehouse for each of these component types follows a pattern. The SQL for every Hub will look the same. So, you can reduce the SQL you need to write to a few templates and insert metadata to generate the actual SQL needed for the ELT processes.

Automation gives your project a real productivity boost and is considered a good practice in any modern data warehouse service.

Across the top of the diagram below we show the data flow we want to build. A source system has a FLIGHT table and recent changes to this table have been loaded to a staging layer. As part of the staging load process we derive additional values, for example pre-joining two columns we may need to use as a compound hash distribution key when we load your HUB, LINK or SATELLITE tables.

We want to load the HUB table "HUB_OPERATOR" which holds a master list of Airline Operators. The staged Flight table contains one column of interest that map across into the HUB – the business key for Operator, the OPERATOR_CODE.

To prepare the SQL to load the HUB we can assemble some metadata – the name of the source table and columns of interest and the target HUB table name and columns of interest. We apply these to the template, replacing fields demarcated by double curly brackets: {{}} and end up with the generated SQL on the right.

The template works for all HUB tables.

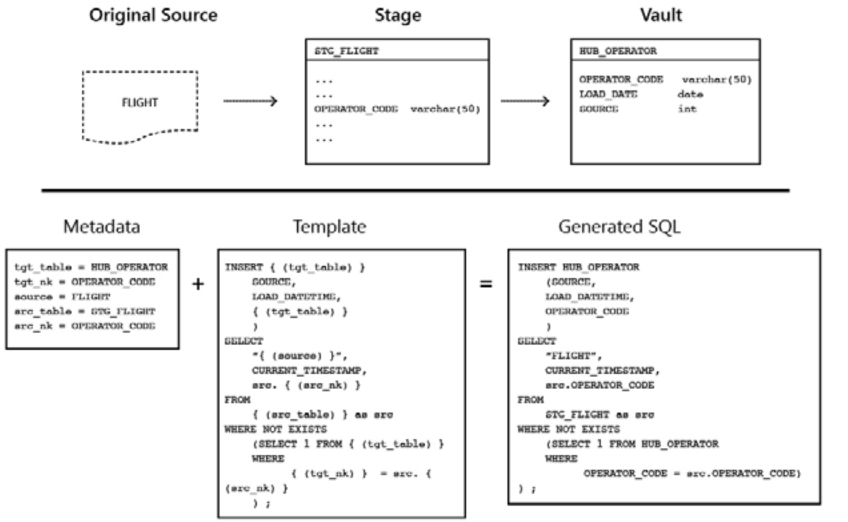

Figure 6: using a template and metadata to generate SQL

If you can prepare an array of HUB metadata, you could loop through the array and generate a full set of SQL code to load HUBs.

The same principles apply for LINK, SATELLITE and other types of table: assemble relevant metadata, apply a template and generate the SQL.

You may be tempted to develop your own code generation utility for your project. This is possible, but you may find that it takes a while before this code beds down, that it needs to be integrated with other tools to work effectively, it is difficult to refactor code, and it becomes a drain on your project resources. If this is the case, then you may want to evaluate a Data Vault automation tool available from the market.

There are tools available that support code generation plus a whole lot more of the development process – including version control, data exploration, data modelling, mapping, customisation, generation of documentation, testing and support for multi-layer Data Warehouse technologies. Two notable tools that automate for Azure SQL Data Warehouse target architectures are Wherescape and BimlFlex.

Automation can lead to quite dramatic improvement in productivity, agility and quality. It especially pays back when you need to refactor or rework your design. We have seen examples where user requirements discussed in a workshop - for example, extracting an extra field from a Data Lake data set, loading it to the Data Vault and onward into a Star Schema - have been implemented in 10-15 minutes. The change was in place and running almost before users had returned to their desks.

Best Practice #7

Use a Persistent Staging Layer (aka Data Lake)

A Persistent Staging Layer (or PSL) is an optional architectural layer in a Data Vault implementation.

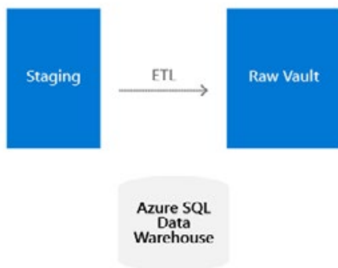

Figure 7: Regular Data Vault Architecture, Transient Staging and Raw Vault

In a traditional approach to Data Vaults, a Staging Layer sits at the very front and is used to marshal and assemble data feeds in Azure SQL Data Warehouse before they are fed into the raw Data Vault. Staged data is truncated every cycle after load into the raw vault.

A Persistent Staging Layer is more akin to a Data Lake. The PSL sits at the front of the Data Vault and holds copies of all the data fed to the system.

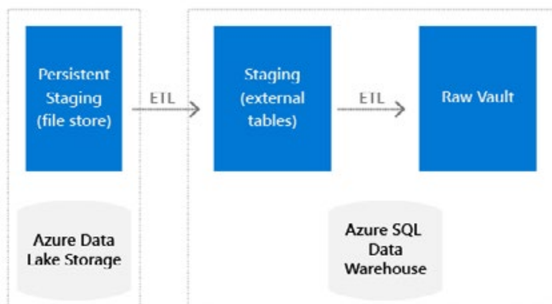

Figure 8: Adding a Persistent Staging Layer

This data may be processed to optimize storage, e.g. converting it into Parquet or Avro format and zipping.

This makes the PSL an ideal candidate for Azure Data Lake Storage. The advantage of this is that you can access or manipulate this data in a number of ways: you can use Data Bricks, Polybase, attaching the data to Azure SQL Data Warehouse as an external table.

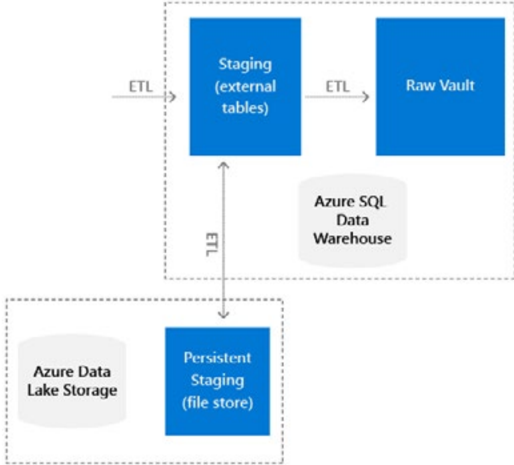

Figure 9: Optional Layout, PSL Branches from the Staging Layer

If you are uncomfortable that you've added another processing stage before data gets into the Data Warehouse, there is a variant of this where the staging layer writes to the PSL after it loads data to the Data Vault.

Architects who add a PSL find the following benefits:

- You can load data to the PSL even if you aren't processing it yet in the Data Vault. Ingesting data to a PSL is relatively easy, so you can press ahead and start queuing data sets in the PSL for eventual addition to the Data Vault. You benefit by having a ready-made set of historical data when you start loading that data, and the PSL can act as a kind of Operational Data Store allowing users to report directly against individual raw tables for simpler operational reporting use cases.

- You can process parts of tables, ignoring columns you don't yet need or want to bring into the Data Vault. You aren't forced to process and understand all the columns of a 100-column table – you can concentrate on what you need to add value in each sprint. When you are ready to process the additional columns, you have the historical data sets available to replay the load of these columns, and the Data Vault models allow us to break out these additional columns into separate Satellites.

- If you make a mistake, you can delete data loaded in error, correct the process and replay the data feeds from the PSA. This makes a huge difference. You have a safety valve that takes the stress out of modelling the Data Vault and developing and deploying ETL. It means that you can experiment with variants of the Data Vault schema and select what works for the project.

- It becomes a central store of the truth - a place to store copies of near raw data exactly as the data exists in source systems. And if you discontinue a system, while it may disappear from the network, its data will live on in the PSL.

Best Practice #8

Inserting Records into the Data Vault

The Data Vault pattern calls for data to be inserted into a Hub or Link only if a new record has been sourced. Satellites create new records only if the Satellite's payload has changed (or it is the first time the record is created). Any existing Hub, Link and Satellite records are never updated.

```
INSERT HUB_BOOKING (BOOKING_PK, BOOKING_REF, SOURCE, LOAD_DATE)
SELECT stg.BOOKING_PK, stg.BOOKING_REF, stg.SOURCE, stg.LOAD_DATE
FROM STAGE_BOOKING as STG
WHERE NOT EXISTS
(
SELECT 1 FROM HUB_BOOKING WHERE BOOKING_PK = stg.BOOKING_PK
)
```

Figure 10: example SQL to Insert Records into a Hub, for use with an Azure SQL Data Warehouse Implementation

Azure SQL Data Warehouse doesn't support the SQL Merge statement. You need to construct an equivalent SQL statement to work in its place.

The example code alongside shows how this can be done. It sets out how to insert records into a HUB_BOOKING table from the staged data table STAGE_BOOKING.

The SQL is present in two clauses – the bottom clause selects data from the staging layer table that does not exist in the Hub table and the top clause inserts that data into the Hub table.

SATELLITE inserts are made if any of the SATELLITE payload changes. If the SATELLITE has a number of columns you can end up with a lengthy comparison clause – each column has to be compared in turn to see if has changed, and the comparison also needs to check for changes to/from null as well as nulls don't behave well in regular comparison statements.

To simplify the SQL a common Data Vault practice to use a HASHDIFF. You concatenate all of the data in the payload (sorting columns into alphabetic order) into one string and hash the result. To avoid constructing similar strings with odd column combinations insert a separate between each column (e.g. a pipe '|') and replace null values with a placeholder (e.g. "^^"). The HASHDIFF is one more metadata column added to the satellite record. With Azure SQL Data Warehouse you should use the SHA256 algorithm (as MD5 is deprecated). Simply compare the hashdiffs values of the existing SATELLITE with the incoming data in the staging layer to see if the payload has changed.

An example is given below:

```
Say that a staging table has the following fields that are part of a
target satellite table:
[GIVEN_NAME]             nvarchar(50),
[FAMILY_NAME]            nvarchar(50),
[YEAR_OF_BIRTH]  int
We would calculate the HASHDIFF as:
SELECT
HASHBYTES('SHA2_256',
CONCAT_WS("|",
TRIM(ISNULL(FAMILY_NAME, "^^")),
TRIM(ISNULL(GIVEN_NAME, "^^")),
ISNULL(CAST(YEAR_OF_BIRTH AS VARCHAR(4)), "^^"))
)) AS HASHDIFF, …
```

Figure 11: fragment of SQL to calculate a HASHDIFF column from 3 columns in a staging table that will form the payload of a satellite table in the raw vault

Best Practice #9
Resource Classes

SQL Data Warehouse performance depends on having a good idea of the performance capacity in your Data Warehouse engine and deciding how this is going the be allocated to the various jobs you may run.

Remember also that Data Vault is designed for parallel load. All loads into the Raw Data Vault layer, and to a substantial extent all loads to Staging, the Business Vault and Marts may be run in parallel. Your only constraint is the ability of the data storage mechanism to handle multiple parallel inserts.

Resource classes are used to assign a share or quota of the available processing capacity to a query. Each database user is assigned a resource class to be used whenever that user runs queries.

There are two types of resource class: static or dynamic.

Static RCs allocate a fixed amount of memory, so as you scale out the warehouse you can fit more queries in parallel. Static RCs range from staticrc10 to staticrc80 in size in increments of 10.

Dynamic RCs are allocated a percentage of the available resource. So as the system is scaled out the RC will increase in size.

For example, using EXEC sp_addrolemember 'largerc', 'loaduser' will allocate the LARGE RC dynamic resource class to user LOADUSER. The largerc resource class allocates 22% of memory to that user and all queries you run with that user. It also means you can only run 4 largerc queries at a time – as they take up 88% (4 x 22%) of memory.

For data vault systems static resource classes should prove the most useful as scaling out allows us to run more parallel queries.

By default, all queries are allocated the SMALLRC resource class. This takes up 3% of available memory and enables 32 concurrent queries to run. As memory has a strong correlation to query performance, you may find SMALLRC queries run too slowly for your load patterns and need to extend them.

See docs.microsoft.com/en-us/azure/sql-data-warehouse/resource-classes-for-workload-management for a more detailed discussion of resource classes and their use.

Best Practice #10

Rebuild Statistics after Each Major Load

Up to date statistics are important for Azure SQL Data Warehouse performance.

When you perform each load cycle it is good practice to ensure that statistics are updated so that the Data Vault performs at its best. All Data Vault tables participate in table joins when the Data Vault is queried to build the mart data layer.

When you perform each load cycle it is good practice to ensure that statistics are updated so that the Data Vault performs at its best.

Statistics should be updated for columns used in joins – i.e. natural key columns for hubs, natural key columns for hub satellites, foreign keys in links, and the link primary key for any satellites hanging off a link. These will force an update to the table row count and page count values.

See docs.microsoft.com/en-us/azure/sql-data-warehouse/sql-data-warehouse-tables-statistics for a more detailed discussion.

Conclusion

The Data Vault 2.0 method is a mature approach to develop a Data Warehouse solution as part of your analytics programme.

The set of good practices included in this paper is based on the practical experience of numerous developers working with the Data Vault 2.0 method over many years. They will help ensure your projects are successful. We recommend you apply some of the practices set out in this paper on your next project.

Get started today

- Start building with SQL Data Warehouse with a free Azure account
- Watch the webinar on how to build a Data Vault solution using Azure SQL Data Warehouse
- Learn more about Data Vault and how to get started

© 2019 Microsoft Corporation. All rights reserved. This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it. Some examples are for illustration only and are fictitious. No real association is intended or inferred. This document does not provide you with any legal rights to any intellectual property in any Microsoft product.