Robust ETL Design and Implementation for Amazon Redshift

Max Ganz II @ Redshift Observatory

5th December 2022

Abstract

Usually ETL systems are provided with only minimal information and so are capable only of minimal actions. This leads to simple ETL systems, which for example check for new data files in some location, load them to a table, then move those files out of the way, and have no other capabilities. Such ETL systems are not robust, as they require arbitrary, unplanned human intervention to fix and to recover from bugs or data errors. The key to robustness in ETL systems is the provision of additional information such that the ETL system can undertake a wider range of actions, reducing the need for human intervention. A simple but profound enabler is the provision to the ETL system of the information of the set of available data files, and the set of data files which have been loaded into the database. This allows the automation of a wide range of behaviour, including the recovery process once human intervention has fixed bugs or data errors, and so provides a robust ETL system.

Contents

Principles of ETL Design

Robustness in software engineering is the extent to which a software system can gracefully and correctly handle unexpected situations.

A critical type of unexpected situation present in all software systems is that of faults. A fault will require a certain set of circumstances fulfilled to manifest, at which point it becomes a bug.

In the general sense, faults and bugs are not unexpected; we know they will come. In the specific sense, the exact actual fault which manifests as an exact actual bug, bugs are unexpected.

As with software, data also contains faults, which is to say incorrect data, which manifest as bugs when the incorrect data is used and so produces incorrect results.

Data is challenging to produce correctly because it is silent; whether it is correct or incorrect, it just sits there. It doesn't crash. There are no stack dumps.

To be robust, an ETL system must gracefully and correctly handle bugs in itself, in the data systems it operates, and in the data itself, detected either when they first occur or, as can be the case, after they have been occurring for a long time.

It is of course impossible for the ETL system itself to fix bugs; this work necessarily must be performed by humans, but once the humans have done this work, we should look for the ETL system, rather than humans, because humans are unreliable, to gracefully and correctly handle the unexpected situation we find ourselves in, which is to say the situation where we now have no known bugs, but have incorrect data loaded into the database as a result of a now-fixed bug.

This white paper presents a general design for a robust ETL system, and an implementation specifically for Amazon Redshift, taking into account the properties and behaviours of that database.

ETL Design

Humans as we all inherently understand, being them ourselves, are when it comes to performing any single, most simple task - such as picking up a wine glass - are unreliable. Sooner or later, we fumble, and the wine glass goes crashing to the ground, which is why we all buy wine glasses throughout our lives.

With a wine glass, failure is often catastrophic - the glass falls, and breaks - but with the large majority of simple tasks, failure is not catastrophic and we simply try again. If we are working in a call center and ask a caller for their ID and mis-hear, once we realise, we ask them again.

As such, it is possible to build reliable systems which invole humans when humans have the opportunity to retry, but it is not possible to build reliable sysstems which involve humans when humans *must* and *always* perform a task exactly correctly *on their first attempt.*

The only source of reliability for such systems that I know of are automated systems where those systems have been through a period of debugging, then performed the same task many times, and are then used to perform that same task one more time.

Automated systems are limited in the tasks they can undertake; but to attain robustness, it *must* be that humans are limited to tasks which they can repeat as often as necessary *until* they get it right, and so do not have to always perform the task correctly on the first attempt. Only automated systems can be used for tasks which must always be performed correctly on the first attempt.

The automated systems for ETL are built from software.

All software contains faults, which can then manifest as bugs.

Naturally, we expect our own ETL code to contain bugs, but I think we tend to assume the database systems will be reliable, or reliable enough we do not need to worry about it.

Two examples from Redshift make it clear we do need to take database reliability into account.

First, in Redshift, the sub-string `'nolock'` is elided from any `varchar` or `char` values obtained via `SELECT`. The SQL command `select 'anolockb';` returns `'ab'`. One developer who ran into this was having problem with company names, and I have found also “Nolock” is a rare but extant Western surname. This bug exists to this day; try it on your own server.

Second, the Redshift `JSON` loader was found to contain a bug which allowed it to load `varchar` strings containing the `NULL` character (binary zero). Redshift mandates that `varchar` data cannot contain the `NULL` character, as it is used as the end-of-string marker; all strings appear to be only the characters prior to the `NULL`, which makes remediation problematic. The developer who ran into this problem had to wipe and re-create his database from scratch.

The consequences of a bug range from the very smallest to the very largest. It may be we end up with a single value in a single row being incorrect; or, as we saw above, it may be the entire database must be reloaded from scratch.

This implies that the ETL system must be able to reload the entire database in a timely manner, which is to say, the ETL system must be scalable.

If the ETL system is not scalable, and we come to need to reload the entire database, we will find ourselves in the situation where the business needs the database back up now or sooner, but it cannot be done.

We must next consider there exists a class of bugs which are worse than those which require a complete database reload.

We can imagine a scenario where a bug which leads during the ETL process to data corruption, which is spotted only after a considerable time, such that the data corruption is now present in all backups, as old backups are usually eventually deleted.

Even if we manage to detect such a bug before all backups are corrupted, it may well be we must go back a long way in time to find an unaffected backups and then we have the painful task of bringing an old backup up to date with all the data which has been generated since then (assuming that this is in fact even possible).

(In fact, I think what would happen is it would be declared that this data could not be recovered, because it would be too much work for what it’s worth : in other words, the ETL system would have failed, as it was not able to cope with the situation.)

What this type of problem means is that we cannot rely on backups of the database as a complete and sure safeguard against failure.

To cope with this problem, what we must do is store *all original data*, completely unprocessed, so that in the event we discover our ETL has been flawed for a considerable period of time, we still have the original data to return to, so we can then correct the bug and now produce valid data not only on an ongoing basis, but for all data to date.

This in fact means backups are unnecessary; we can always simply rebuild the database from scratch. The original data *is* the backup, and where it is wholly unprocessed, it is immune to bugs in the data processing system; in the event of a bug being found, we correct the bug and re-run the ETL process.

In the event that data as a normal part of system operation is deleted or updated, or, bugs are found in the original data which can only be fixed by creating new original data, the data files which will need to be modified are *not* deleted. Rather, they are *superseded.* Each file for example contains a version number in its name, and when a file needs to change, the version number is bumped and so a later version of that file is created.

A policy of retaining all version of all original data files provides a generalized robustness to the ETL system, because no matter what goes wrong, we always have the original data to fall back upon; we can never have gone wrong by inadvertently damaging or deleting original data we in fact later discover we needed to keep. In short, we do not want to be in the businesses of needing to modify original data files. It's too risky, where the consequences are too serious, as these are the data which we need to recover from bugs.

This approach also means the code which handles original data quite literally does not contain the command to delete data, and so we profoundly reduce the possibility of a bug in that software which would delete data.

The next two considerations are Redshift specific.

The first Redshift specific issue is that it will often be the case, sooner or later, that a Redshift cluster needs to be re-sized.

There are two issues with this.

The first is that classic re-size takes the cluster into read-only mode for an extended period of time - easily days - which unsurprisingly often is not viable. Elastic re-size has too many limitations to be a full solution to this problem and so must be set aside.

The second and more forcible issue is that cluster re-size is simply *not* a wholly reliable operation. Sooner or later, you will issue a re-size, and it will not complete, and your cluster will be stuck, and your cortisol levels will go through the roof.

This is not what we expect, nor not how things should be, but it is the truth and reality of the matter, and so if we are looking for reliable systems based on Redshift, we have to find a way to deal with it, which means finding a way to avoid the need to re-size a cluster.

The solution to this problem lies in the ETL system. The ETL system must be able to concurrently maintain multiple Redshift clusters. By this, when re-sizing is needed, a new cluster, of the required size, is brought up and populated. When it is loaded, there are then two clusters concurrently being loaded with data by the ETL system, and then all users are moved off the original cluster to the new cluster, and once this is complete, the original cluster is shut down.

The capability to run concurrently maintain multiple Redshift clusters obviously has other benefits - being able to spin up clusters for testing or development, others for heavy-weight internal data analysis (keeping that load load from the normal production cluster), and so on.

The second Redshift specific issue is that all sorted relational databases exist in a producer-consumer scenario, where the loading of new data or modification of existing data produces unsorted data, and the database operation to sort unsorted data (`VACUUM` on Redshift) consumes that unsorted data, converting it to sorted data.

Remember here that sorting is the method by which it is possible to have timely SQL on Big Data, but this is possible when and only when sorting is correctly operated, and one of the many requirements for correct operation is that data is sorted.

If the data is not sorted, sorting cannot be used correctly, and it is no longer possible to obtain timely SQL on Big Data - in other words, your cluster just crashed and burned - and when a database reaches a point that unsorted data is being produced more quickly than consumed, that's exactly what happened, as the cluster then degenerates into a fully unsorted state.

With Redshift, the data sorting operation, `VACUUM`, operates on a single table at a time, and only one `VACUUM` can run at a time per cluster. That's per *cluster*, not per database, or per schema.

As such, `VACUUM` is a scarce resource. There are 24 hours of `VACUUM` time per day, and that's it. If you have say an unsorted few-terabyte table on say a busy 20 large node cluster, you're looking at a day of `VACUUM` for that one table - during which time, all the other data loaded or updated has been producing unsorted data.

There is then a second need to be able to concurrent maintain multiple Redshift clusters, but here so we can select which data sources are loaded to which clusters. This way, when a cluster cannot keep up with the rate at which unsorted blocks are being produced, we can bring up a second cluster, doubling our `VACUUM` budget, and move sets of data, and their users, over to that new cluster.

We now have already some features necessary for an ETL system to be robust, without yet addressing the core problem of having correct data or data processing available, having just fixed a bug, but incorrect data loaded into the database.

Obviously, we have to remove the incorrect data from the database, and replace it with correct data.

The first step then once a bug has been spotted is to fix the bug. A human must do this work. This is why we cannot kill all humans.

Once the fix has been made, it will either be we have valid original data which has been processed incorrectly (but which can now be processed correctly), or we had actual incorrect original data, which has now either been superseded with a later version of itself or is now having its problems masked during the ETL process, by code we have now added for this purpose.

As such, the ETL system can now produce correct data.

For the ETL system to perform the task of removing incorrect data from the database and replacing it with correct data, it must know what data is available for loading, what data is present in the database, and what loaded data is incorrect.

A simple method to achieve the first of these requirements, knowing what data is available for loading, is to list the files in the S3 buckets for the data sources.

A simple method to achieve the second of these requirements, knowing what data is present in the database, is to have an additional column in each table in the database which stores the filename of the file the row came from. The ETL system can construct the list of loaded files by issuing a `SELECT DISTINCT` on the filename column.

(This is the simplest case - the general case is that tables will be loaded by rows generated from multiple data sources and so there is one filename column per data source).

This means we use the database itself to store our state, which seems appropriate, given that it is a database :-) this also naturally allows us to support multiple clusters, as state will differ between clusters, and this allows each cluster to store its own state.

The third requirement, what loaded data is incorrect, is in part something the ETL system can handle, but in part is something it cannot handle.

When the bug is such that original data files have been superseded, the ETL system can handle the removal of incorrect data. This is because the ETL system can know which data in the database is incorrect, as it will be all rows which come from the files which were superseded. The ETL system then can and correctly and robustly perform the necessary delete (and should we be worried about this delete, we can remember that all original data are stored in S3 - we can *always* recover).

However, when the bug is in software, either in the ETL system, or in a system it operates, or in pre-processing work, there is no way for the ETL to know which data in the database is now incorrect. This is a problem of much the same type as getting the ETL system itself to fix bugs; humans are required.

It must then be that humans are once again is involved; but, as a human is involved, what must now look for is the utmost simplicity, and that whatever is done fails safely.

What is done, then, is that the human must delete the incorrect data in the database.

This is simpler and safer than it sounds, because often all that is needed is a truncate of the affected table or tables. This is less efficient than deleting exactly the incorrect data, but it is simple, fast and easy to reason about; the ETL system will simply see that no files are now present in the table(s) and regenerate all the data for those tables.

If the human wishes to be more precise, then it can by using the filename column delete and only delete those rows which the human knows will now be incorrect; and, again, this fails safely, because any excess delete is harmless - it will, just as the correctly deleted data, be reloaded by the ETL system. We would only lose efficiency.

So, to summarize : the ETL system knows which files are in S3, as it can list files in S3. The ETL system knows which files have been loaded into the database, because it can scan the filename column(s) in the tables in the database. Finally, for the case when original data has been superseded, the ETL system will automatically delete the incorrect data and replace it with correct data; but for any other type of bug, a human must delete the incorrect data in the database. The act of deleting the incorrect data will cause the ETL system to replace that data, and now the bug(s) have been fixed, the data being loaded into the database will be correct.

Some final notes.

We can also see that this design inherently handles cluster bring up; it is not a special case. When a new cluster is started, the ETL system looks to see which files are loaded - there are none - and so it now loads *all* files into the new cluster. System bring up is actually simply part of normal ETL operation.

We can also see now it is straightforward to imagine concurrently maintaining multiple clusters; each cluster having its own tables will have its own state of what has been loaded and what not. The ETL system simply needs a list of cluster end-points, and to iterate over them, performing the same steps on each - looking at what's in S3, looking at what has been loaded in the given cluster, and loading whatever data is missing.

ETL Implementation

The target database is Redshift, with S3 to store original data.

We begin by having new, incoming original data delivered into S3 buckets - one bucket per data source.

The next stage is the sanitization, validation and possibly amalgamation (shortened from now on to *SVA*) of this data in S3, and by this, the production of a new set of files, which are ready for loading by the ETL system into the database they are intended for.

It is I find often the case that incoming data files will cover a fixed time period, and in such cases, the name of each file is usually arranged such that it contains the start and end timestamp of the period of time covered by that file (which is exactly what we want).

However, it could be that we have streaming data, which unless we took steps to make it so (Kinesis, for example), would not be landing in files in S3.

Now, there are a range of scalable streaming data processors, which we could use for SVA - but these streaming data processors would not, of course, handle data which is landing directly in S3 as files; but a mechanism which handles files landed in S3 would handle *all* original data sources, which either naturally or were arranged to have their files landing in S3.

It is then simpler, a single mechanism, to SVA files which have been landed in S3, and not to SVA streaming data.

Moreover, we must also remember the key principle of utmost simplicity in the sub-system which obtains original data, and places it into S3. Original data is often not reproducable; our single goal then is in the simplest possible way, with the least functionality, and so in the most reliable way, get original data into S3 - introducing SVA introduces a complex processing task prior to the landing of the data in S3.

What do we do if the stremaing SVA contains bugs? we'd be dead in the water.

So the first task is and only is, to get original data into S3.

SVA happens afterwards.

Now, it must be understood that SVA must happen *outside* of the database, prior to loading data into the database.

The first reason for this is that it can be file amalgamation is needed for performant data load. If you try to load a thousand small files into Redshift, come back in a few hours; amalgamate to one file per slice, come back in one minute.

The second reason is that the data may be flawed in ways such that it cannot *be* loaded into the database.

One example of this issue which I ran into is that Redshift is case sensitive to the string which represents `NULL` in `CSV` files. I had files coming in, where sometimes the string used ("NULL") was upper-case, and sometimes lower-case - in the same file, varying by column. The only solution was to pre-process, as Redshift could not load these files correctly in their original form, and where the data was coming from a large third party company, there was no possible way for the incoming data to be corrected, either politically or probably technically either, as it would have required correctly changing their code base so it only ever emitted one case, which likely enough would be challenging.

SVA requires the flexibility and power of generalized programming language, and Python is a great choice, but whatever the choice is, it must in the context of being scalable, because as we've seen in the design, it might be there is a bug such that such that *all* data must be pre-processed in a timely manner.

Now, there is in database technology design a trade-off between how much is performed when data is loaded, and how quickly queries run; the more work performed when loading, the more quickly queries run.

One of the methods by which this trade-off is made is to develop during loading the information necessary to allow queries when they run to know enough to access only the rows needed for the query, rather than to read all rows. After sufficient queries have been run, the time saved exceeds the cost of the work done when loading the data.

With sanitization and validation, however, we specifically *do* want to access all rows, *and* we are only going to run one query - the query which is performing sanitization and validation; it will be run only once, and no other queries are being issued on the original source data.

We have then no benefit to performing any work when loading the data, to speed up our queries.

As suchss we look for a scalable database system which performs no work when loading data, and that technology type is Hadoop, which in AWS, is Elastic Map-Reduce, known as EMR.

EMR supports Python. From a coding point of view, the way it works is essentially that every row in the source data is presented to your Python code, which you then process, and whatever you emit to stdout becomes the processed data, the output.

As such, you can sanitize and validate a row based on that row only. The work being done is that which is necessary to ensure the data can and safely be loaded into the database; not more. Sanitization and validation which requires access to the full body of data in the database is performed in the database itself, when processing the staging tables.

The next component is this system is that which monitors the buckets where incoming new data files are placed, where when it sees a new file, it pre-processes that file, and emits the processed data file into a second bucket; so now we have two buckets per data source, one for original data files, one for processed data files.

The simplest and most robust method is to poll the bucket every few minutes. I prefer this, if at all possible; it fails safe. If for some reason a poll goes awry, say the poller crashes, the next poll will occur and all will be well.

(A more efficient but more fragile method would be use S3 notifications (notifications when an object is created). Here we have the problem that sooner or later, a notification will arrive, we will be working on it, and crash - which means we need to take steps to keep track of notifications until they arep processed - it all gets a lot more complicated.)

Now, we are expecting over time bugs to emerge in the pre-processing system, that the fixes to those bugs lead to changes in the pre-processing work being done, and when this occurs, we likely will need to re-run the pre-processing work for whatever data has been affected by the bug.

The pre-processing system monitors for each data source the original data bucket, and the matching processed data bucket, and whenever there are files in the original bucket which are not in the processed bucket, those files are processed.

As such, all that has to be done when pre-processing must be repeated, is deleting the affected files in the processed bucket. This fails safe, as any excess deletes will simply lead to the deleted processed files being generated again from their original data files.

So now we have original data files turning up in the original data buckets, one bucket per source, and there is a matching second bucket for each source, which contains processed - sanitized and validated - versions of each original data file.

Now let's move on to loading data into Redshift.

To begin with, let's imagine a single table, which is loaded from a single data source.

The ETL system begins by scanning the processed file bucket in S3 and forming the list of files there.

Then the table is scanned - recall we have a column in the table which indicates the pathname of the file each row came from - to produce the list of files which are already present in the table. Runlength encoding is used, so this column is very nearly free.

The ETL system subtracts from the set of files in S3 the set of files in the table, and loads what remains, in a single `COPY` command (using a manifest so multiple files can be specified) to a staging table for that data source.

The staging table is created or truncated before the load and experiences a single `COPY` only, so is fully sorted. This saves us the need to use `VACUUM`, which is a scarce resource, and sets us up nicely to be efficient in our further work with the staging table.

Depending on how much parallelism we choose to build into the ETL system, there may be a single staging table per data source, which is used for all loads, or it might be each individual load creates (and then drops) its own staging tables, so any number of individual load can run concurrently.

The ETL system at this point can perform arbitrary work on the data in staging - for example, there might be say a lookup table of addresses, and any new addresses are inserted into that table.

The ETL system now performs the `INSERT` to load the data into the production table, and that command would convert any values which are held in lookup tables (such as addresses) into ID values to index into the relevant lookup tables.

We now have data loaded into the production table.

When the ETL system runs again, but no new data has arrived, the ETL system checks the processed data bucket, and it will find the list of files in there matches the list of files loaded into the table, and so it will do no work.

If we imagine now that it was discovered a particular original data file in S3 is corrupt, and so a second version is created, where the ETL system is designed such that it is aware of versioning, it will delete from the table all rows from the superseded files, and then load the latest version.

Deleting all rows from file from the table is easy, as rows indicate which file they came from.

As such, fixing broken original data is simply the act of placing the new file, with a bumped version number, into the original data bucket. The pre-processor will process all unprocessed files in the original data bucket, and the ETL system will notice the new file, with its bumped version number, and act as described above.

What we see here is that no special operations are being performed; the ETL system is doing what it does *always*, again and again, just one more time.

Similarly, if we imagine the table was inadvertently truncated, the ETL system would simply reload the table.

This behaviour is valuable, because it is easy to reason about, and allows us to know, as much as can be known, that the table is now in its correct state.

Now, the mechanism described so far works on the basis of filenames.

If *all* the records from a file are deleted, then that file will no longer be seen, and it will be reloaded. If only *some* of the records are deleted, the file will be seen, and the file will not be reloaded; the mechanism fails.

Now we could think to come up with a more capable mechanism, but what I find in practise is when I have reason to doubt a table, it's very often simple, easy, reliable, and enough, to just truncate the table and let the table be reloaded by the ETL system.

Working at the per-file level is appropriate for normal ETL operations, as these operations are all on the basis of files - new files come in, sometimes a file is replaced, we load files into Redshift, and so on. We never in our normal operations work with anything less than a file; it's files all the down.

In any event, if we were to consider improving the mechanism, the next level of fidelity is to detect single rows errors (missing or added), and the level after that is to be able to detect single column errors, where a value is incorrect.

Single rows we could imagine, something along the lines of recording row counts in data files. Single values would require checksum, but then we would not be able to know which row contains the error.

An easier approach can be to make a new table in Redshift, which is identical to the table we want to check, let the ETL system load it, and then compare the two tables; but this is more work than just truncating the original table.

Now let's consider a table which has two data sources.

The first issue we must now address is that each data source covers a given period of time in one of its files, and the period of time may differ between sources; the question is how do we handle deciding which files to load, so when populating the production table, we avoid losing data, or generating duplicate rows?

One approach is to arrange for data sources to cover periods of time such that there will be exact integer ratios of files between all data sources. So if we had say a 15 minute source, and a 60 minute source, we would load every 60 minutes, using four of the 15 minute files, and one of the 60 minute files.

Where Redshift is batch-like, I've actually found this always sufficient. It's simple, easy to reason about, and usually add no or little latency; most data sources are hourly or daily.

We can however without much thought design a fully capable mechanism, which examines the time periods available for each data source, by converting the start and end times for each file into seconds, and then iterating over that list, merging any contiguous times, leaving us if all times are contiguous with a single row, or multiple rows if not, where the gaps between each row are the intervals which have no data, where we then load all processed files which overlap the missing data intervals, and have start and end time conditions in the `WHERE` clause of the `INSERT` such that we which emit only rows into the production table which fall into the gaps of missing data; but making this mechanism is work, and I've never needed it - but it can be done, and without excessive difficulty, should it be needed.

A simple solution to determining the files to load when all data sources can exactly overlap is to take one of data sources which has the longest period, and we know the periods of the other data sources, *compute* the names of the files we will need in the other data sources.

So for example, if we have a 60 minute data source, and we are going to load the file `"2022-12-01 13:00:00 to 2022-12-01 14:00:00 version 0.csv"`, and we have a second data source which is 15 minutes, we take the start time of the file from 60 minute source, and produce four filenames from it, each being one of the 15 minutes in that hour, and we the inspect the list of files in the processed S3 bucket to see if those files are present.

(We will not know in advance the version numbers of those files, but that's no problem; we match to the leading part of the filename, and then if there are multiple matches, use the latest version.)

If all the necessary files are present, we can then proceed to load the data sources to their staging tables, whatever predatory work which needs to be done is done, and then we issue the `INSERT` which joins the staging tables and emits rows into the production table.

We can now consider the situation where bugs are fixed, and a human has to be involved.

Imagine that some time after data has been loaded to this table with two data sources, one of the existing 15 minute data files is superseded. The ETL system notices; and here, it must figure out which data files to load (not difficult), but also it will need first now to delete all rows which come from any of the files involved in the reload, to prevent duplicate rows being created (that, or introduce start and end timestamps in the `WHERE` clause of the `INSERT` statement).

Again, this is not difficult, as every row indicates the files it came from.

Finally, supporting multiple clusters concurrently is trivial; the ETL system simply iterates over the endpoints over the different clusters, or we run multiple instances of the ETL system, each with a different cluster endpoint (we still of course only run one instance of the system which pre-processes original data).

Conclusions

In my experience, every ETL system I've seen has been one-way; provided with very little information, are capable of only very simple actions - read bucket, load data to database, move data out of the way - very much a case of "ug! me Tarzan, you Jane".

Such ETL systems are fragile as they require arbitrary, unplanned human intervention not only to fix bugs and errors in data (which must fall to humans), but also the *consequences* of bugs or errors in data (which could be automated); and it's bad enough fixing bugs or data errors in a live system, without making it worse by then also needing to manually fix their consequences. I'm very much in favour of leaving open-heart surgery to the doctors.

The key to extending ETL system functionality is making additional information available to the ETL system, so the ETL system is capable of a wider range of actions.

In particular, a key enabler is providing to the ETL system knowledge of the full set of data files available in data storage (such as S3), and the set of data files loaded into the database (for example, by means of an additional column or columns, in tables, which indicate the pathnames of the file or files from which a row originated).

This additional information then easily allows a wide range of additional ETL system functionality, such as loading only files which are not yet present in the database, being able to supersede existing (presumably broken) data, being able to re-load data which is found to have gone missing from the database, and so on.

Furthermore, two very useful consequences are that cluster bring-up becomes part of normal operation, where the ETL system, which automatically loads new files into the cluster, when seeing a new cluster, which by being new is empty, simply loads all data files loaded to it; and that supporting multiple concurrent clusters is trivial, as each cluster contains within itself the state of which files have been loaded to that cluster; and supporting multiple concurrent clusters is of critical importance to Redshift, as it provides a wholly safe mechanism for cluster re-size, as both classic and elastic resize are not wholly reliable but come with a small catastrophe risk due to flaws in the re-sizing process, and multiple concurrent clusters provides a solution to the problem of `VACUUM` being a scarce resource, as each cluster provides another 24 hours of `VACUUM` time, with coherent, self-contains groups of tables (and their users) being spread over the multiple clusters.

Revision History

v1

- Initial release.

v2

- Some minor rewordings/rewriting, to improve the prose.

v3

- Some more significant rewordings/rewriting, to improve the prose.

v4

- Changed to Redshift Research Project (AWS have a copyright on “Amazon Redshift”).

v5

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

v6

- Web-site name changed to “Redshift Observatory”.
- Updated links from redshiftresearcproject.org to redshift-observatory.ch.

Appendix A : Notes on the Use of Transactions in ETL

I often find ETL systems wrap the entire ETL operation in a transaction, the thought being that if anything goes wrong, then the system will automatically rollback any work done, returning the database to its original state.

This is completely plausible, but I consider it in fact in practise to be a serious blunder.

The key problem is transactions seem simple, but in fact under the hood are complex, and by this sooner or later end up causing both expected problems, such as not allowing the use of `TRUNCATE` (and so forcing the use of `DELETE`, which is extremely bad for Redshift), and unexpected problems, typically the dreaded serialization isolation failure, where I have yet to meet a data engineer who actually understands what that really means and knows how to intentionally and correctly fix the problem (and a fix may not in fact be possible, given the behaviour of the ETL system); I see users just semi-randomly changing the order of work and hoping the problem goes away, which is no basis for a system of government.

The property of automatic rollback is valuable, there is I think often a better way to achieve effectively the same end, a way which does not require the use of transactions.

In my experience, it is often the case that tables in any given database form a pyramid; each layer consists of tables which depend upon the layer below, but have no interaction with or dependency upon on the layers above; and it is only the tables in the very top layer which drive the behaviour of the system using the database.

As such, we can in completely safety insert new values into any given layer, so long as they layers below have been fully populated with what-ever values will be needed.

In other words, we can safely populate tables *one by one*, without a transaction, so long as we populate *from the bottom upwards*.

During the ETL process, if any query fails, we abort; we will need to re-run that ETL process - but by leaving in place the records so far insert, we have done no harm, because all we have done is populate lookup tables with some extra values, which simply are not being used.

When the ETL insert queries run, they insert only new values - those not already present in the table - so they can safely run any number of times.

It is only when we come to the top layer that inserts will actually lead to changes in the behaviour of the system. Even here I find usually it's fine in practise to insert to one table at a time, and it may also be that there is only one table in the top layer.

In short, this is an idempotent ETL design. So long as inserts only insert new records, and proceed from the bottom layer of the pyramid upwards, we can run an ETL process as many times as we like, until it finally succeeds.

(Well, unless we managed to issue a successful insert which emits incorrect records, but that's a different type of problem, and a transaction would not have helped.)

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.