



# Amazon Redshift Advice and Best Practices

A Real World Guide



# TABLE OF CONTENTS:

Who is this eBook for? .....	1
Introduction to Amazon Redshift Best Practices ...	3
How Redshift Works .....	5
Best Practices .....	10
Scheme Design .....	11
Loading Data .....	22
Querying Data .....	28
Summary .....	32

## WHO IS THIS E-BOOK FOR?

This e-book is for those with an interest or stake in Amazon Redshift who:

- ❖ Are existing users of Redshift, or have recently implemented it
- ❖ Are considering using Amazon Redshift
- ❖ Are evaluating Cloud-based data warehouse and business analytics technologies, and 'big data' technologies in general
- ❖ Are involved in the Amazon Web Services ecosystem – for example, partners, resellers, or AWS employees

## It examines:

- ❖ Core and important concepts of how Amazon Redshift works 'under-the-hood'
- ❖ Best practices for ensuring optimal performance in different Redshift usage scenarios
- ❖ Configuration options and their impact on performance, cost and scalability

## It includes:

- ❖ Practical advice on how to configure Redshift
- ❖ Real-world' examples
- ❖ An evaluation of Amazon's recommended best practices with insights as to which matter most
- ❖ Tips on configuration in order to optimise your Redshift performance and spend
- ❖ An explanation of how Redshift works

# CHAPTER 1:

## Introduction to Amazon Redshift Best Practices



In today's world of 'big data', analytics, and data warehousing, data volumes continue to grow, seemingly only matched by businesses' appetite to exploit data for commercial advantage, and do so ever more quickly.

Historically, to deal with the kind of large data volumes that are now becoming commonplace, expensive and heavyweight technologies from vendors such as Teradata, Netezza and Vertica were needed.

These typically cost millions of dollars, and required hardware, software, and consulting.

As Amazon Web Services (AWS) has changed forever how IT infrastructure can be delivered – on-demand, scalably, quickly and cost effectively – Amazon Redshift is doing the same for data warehousing and big data analytics. It offers a massively parallel columnar data store that can deal with billions of rows of data, yet which can be up and running in a few minutes, and operated for a few cents an hour.

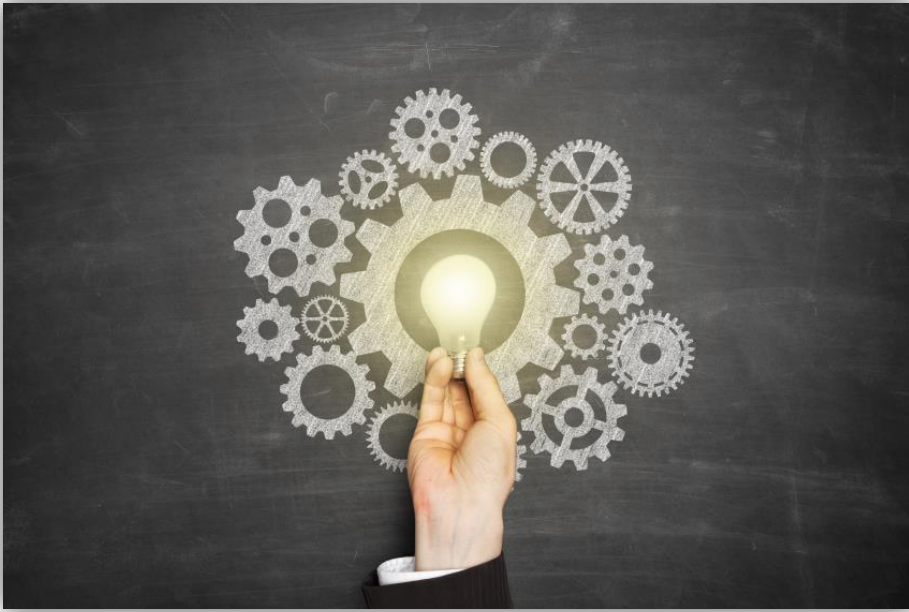
But how does Redshift work? And more importantly, how do you configure it to make sure it delivers the performance and scalability that you need for your particular application? And how do you ensure that you don't waste money by mis-configuring your Redshift cluster – failing to gain all the benefits that the platform can deliver?

This e-book examines the best practices that you need to understand in order to ensure your Amazon Redshift project really flies, as well as explaining the underlying concepts of how the platform works, and why those best practices make the difference that they do.

It is based on real-world, hands-on experience gained in delivering literally dozens of data warehousing and analytics projects using Amazon Redshift, over a period of approximately 18 months between September 2013 and March 2015. Enjoy the read!

# CHAPTER 2:

## How Redshift Works



When considering best practices for Amazon Redshift, it is really useful to understand exactly how Redshift works under the hood.

In this section we examine the key fundamentals of the Redshift columnar database engine and how it does its stuff.

## How Clusters Are Arranged In Redshift

- ❖ **Cluster:** the cluster is the overall instance or configuration of Redshift. You cannot have just a Redshift server, only a Redshift cluster. A cluster is made up of a leader node and one or more nodes. At cluster level you decide the storage technology for your Redshift implementation - SSD or magnetic.
- ❖ **Leader Node:** The leader node is what you and your application treat as the database. It is the front-end facade to the complexity of the Redshift cluster behind the scenes. Behind the leader node sit one or more *nodes* that actually do the work - but what your application sees is just one, nice, simple leader node. You can only have one leader node per Redshift cluster. It looks, to the outside world, like a Postgres database.



Redshift acts like a Postgres database because once upon a time, it kind of was. Redshift's technology is based on a product called ParAccel which is a commercial data warehouse platform, itself forked from Postgres. Amazon invested \$20m into ParAccel in 2012 and launched Redshift shortly after in early 2013



- ❖ **Nodes:** Data is distributed across nodes, and an individual node is roughly analogous to a virtual machine. How data is distributed across the nodes depends on the schema design and **this is where a lot of the focus of the rest of this e-book will be.** A lot of the best practices are about getting the right data into the right nodes, for optimum performance.

In layman's terms, the more nodes you have, the more potentially powerful your Redshift cluster is. However, if you don't follow the best practices properly, additional nodes can yield negligible (or in fact negative) real world

*"If you don't follow the best practices properly, additional nodes can yield negligible (or in fact negative) real world performance and scalability gains"*

performance and scalability gains. Obviously, it is worth bearing in mind that the more nodes your Redshift cluster has, the more expensive it is to run. However, you can re-size the cluster at any time (as long as you don't reduce the cluster size below the amount of storage that you need.)

- ❖ **Slices:** The type of node that you select governs the number of slices that each node has. A slice is roughly analogous to a processor (or core) allocated to working on the data stored on that node.

The correct use of slices allows a node to make use of its multiple cores. When the cluster allocates work to a node, the node can further split this work down across its available cores/CPU's, assuming that the data is structured in a way so that it *can* be practically and efficiently split up.

It is important to understand how nodes, clusters and slices are arranged when following best practices around distribution styles and sort keys - **much more on this later.**

## **How Data Is Stored In Redshift**

If you were implementing a data warehouse in a traditional relational database technology – MS SQL Server or MySQL, for example – then you would design a star-schema, fill it with data, and then index the fields that users want to filter, group by, and use. But because you don't know in advance what fields those are (users have an annoying habit of wanting to use the one you hadn't thought of), then you end up indexing *everything*. At which point, you have got two full copies of your data: one in the main tables, and one in the indices.

And so, columnar data stores (like Redshift, Vertica, Netezza and Teradata), at a very simplified level, admit in advance to themselves that *every column* is going to need an index, and thereby do away with the main data store. They are, in effect, therefore, *just* an index for every column.

As a by-product of this indexing, all the values of the same type and those which have similar values, are organised next to each other in the indices. As such, compression in columnar data stores is far more efficient than in traditional RDBMs.

This is all relevant to your use of best practices in Redshift. For instance, when you are defining a table in Redshift, you may, if you wish, choose from one of 11 different compression strategies (Redshift calls them 'column encodings') for each column in your table. Picking the right one will impact your storage and as a consequence, can also impact performance. Or, if you follow other best practices, you can leave Redshift to pick the correct one for you.



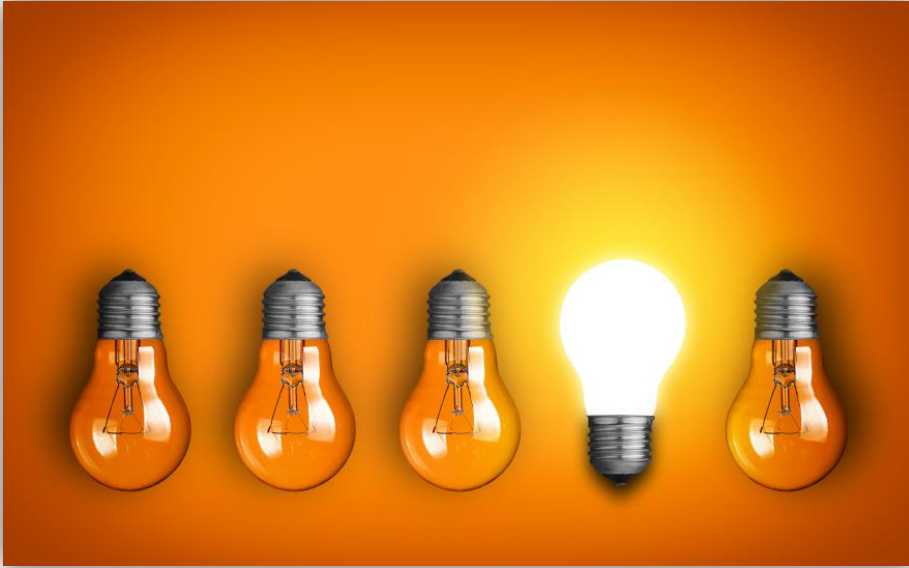
In most databases, column encoding refers to the character set used on the column e.g. Latin-1, EBCDIC, Unicode. This is an easy-to-fall-into "gotcha" in Redshift.

In Redshift, all character data is UTF-8 encoded Unicode, so there are no character sets to worry about.

The term 'column encoding' in Redshift refers instead to the compression scheme to use on the particular column.

# CHAPTER 3:

## Best Practices



The Amazon best-practice documentation contains dozens of recommendations.

Based on our real-world experience of using Redshift, some of these are more important than others. There are some best practices that, in our opinion, you absolutely have to implement. There are others which, if you've done some other stuff right, you can judiciously ignore.

*"There are some best practices that, in our opinion, you absolutely have to implement. There are others which, if you've done some other stuff right, you can judiciously ignore."*

In this section are the best practices that we think, based on our experience of using Redshift in the real world, that you *should* implement, and why.


We've broken down the best practices into 3 sections:


- ❖ **Schema design** - the layout of your tables, and how they relate to each other
- ❖ **Loading data** - Getting the data into the Redshift cluster in the first place
- ❖ **Querying the data** - Getting it back out again, hopefully aggregated and transformed into something useful.


## Schema Design

The relevant section of the AWS documentation for schema design best practices can be found here:

[http://docs.aws.amazon.com/redshift/latest/dg/c\\_designing-tables-best-practices.html](http://docs.aws.amazon.com/redshift/latest/dg/c_designing-tables-best-practices.html)

Amazon Best Practice	Our experience
<p data-bbox="149 260 482 347">Use the smallest possible column size</p>  <p data-bbox="154 620 478 651"><a href="#">Go to Amazon best practice</a></p>	<p data-bbox="551 260 751 299"><b>What is it?</b></p> <p data-bbox="551 307 1308 394">Designing each column to be the smallest size required to fit your data into it and no bigger.</p> <p data-bbox="551 442 751 481"><b>Our advice</b></p> <p data-bbox="551 488 1308 662">In our experience, using the smallest possible column size will make little or <i>no difference</i> to the stored data size, as compression within Redshift will compress out any inefficiencies.</p> <p data-bbox="551 710 1322 1116">Depending on the query it <i>could</i> potentially have an impact on query performance, however. When the data is decompressed for use in a query, it will be held in its full width, wasting memory and potentially impacting query performance. At a basic level, this could negatively impact query performance and at the extremes of data volume, it could eventually lead to an out-of-memory error.</p> <p data-bbox="551 1164 1300 1387">We typically deal with data volumes in the hundreds-of-millions of rows when we use Redshift. These are usually as part of star schemas, where the query complexity is quite simple.</p> <p data-bbox="551 1435 1282 1570">In this situation, we find that we don't worry about using the smallest size of column possible, as the effect is negligible.</p> <p data-bbox="551 1619 765 1657"><b>The Verdict</b></p> <p data-bbox="551 1665 1272 1839">Don't worry too much about it unless you're dealing with gazillions of rows AND you're getting out of memory errors when your queries run.</p>

Amazon Best Practice	Our experience
<p data-bbox="158 262 444 392">Let 'Copy' choose compression encodings</p>  <p data-bbox="139 620 462 649"><a href="#">Go to Amazon best practice</a></p>	<p data-bbox="521 262 721 297"><b>What is it?</b></p> <p data-bbox="521 307 1315 571">The 'Copy' command is one of the ways of getting data <i>into</i> Redshift (the one we suggest you use) and you can ask it to automatically choose the best column encoding (compression) settings for the data it is uploading, rather than you having to decide yourself.</p> <p data-bbox="521 620 721 654"><b>Our advice</b></p> <p data-bbox="521 664 986 707">This one's an important one.</p> <p data-bbox="521 755 1308 929">Firstly, we absolutely agree with the Amazon advice. <i>Do</i> let 'Copy' choose the best column encoding scheme for you. The 'Copy' command is clever, and it knows more than you do!.</p> <p data-bbox="521 977 1325 1199">However, there's a 'gotcha' to be aware of. The 'Copy' command will do a great job of choosing the right column encoding scheme, but it only does so <b>on the first upload of data into an empty table</b>.</p> <p data-bbox="521 1248 1308 1653">As such, ensure that the first time you use 'Copy' to upload data into an empty table, that you use a significant (and crucially, representative) data set, which Redshift can then evaluate to best set the column encodings. If you just upload a few lines of test data, Redshift is not going to know how best to optimise the compression for your real-life workload.</p> <p data-bbox="521 1702 736 1738"><b>The Verdict</b></p> <p data-bbox="521 1748 1293 1831">Redshift does know best (as long as you give it some representative data).</p>

Amazon Best Practice	Our experience
<p>Choose the Best Distribution Style and Key</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What is it?</b></p> <p>The distribution style is how the data is distributed across the nodes. For instance, a distribution style of 'All' copies the data across all nodes.</p> <p>You apply distribution style at table level i.e. for each table in your cluster, you tell Redshift how you want to distribute it... All, Even or Key.</p> <p>We have found that how you specify distribution style is super important in terms of ensuring good query performance for queries with joins. The options you choose here also have an impact on data storage requirements, required cluster size and the length of time it takes to execute the 'Copy' command (i.e. to upload data into Redshift).</p>

Continued on page 18





## Distribution Styles In Detail

There are 3 different distribution styles and it's important to understand each one, as well as how they work together.

- ❖ **'All'** is the simplest distribution style. If you set a distribution style of 'All', you instruct Redshift to simply make a copy of the table to every node in the cluster. The upside of this is that when you are asking the cluster to return a query which includes a join, each node executing that join definitely has a local copy of the table you have distributed using 'All'. As such, Redshift does not have to get involved copying the required data across the network from node to node, to complete the query. If a particular node was tasked with completing part of a joined query and didn't have a required table locally, it would have to get the data it needed across the network, negatively and significantly affecting query performance. The downside of using 'All' is that you have a copy of the table on every node in the cluster - taking up space, and increasing the length of time that it takes to use the 'Copy' command to upload data into Redshift, and ultimately meaning that you'll need a larger cluster.

- ❖ 'Even' – Specifying 'Even' distribution spreads the table rows over all the nodes in the cluster, well, evenly! Queries involving that table are then distributed over the cluster with each slice on each node working to provide the answer in parallel. With no joins involved, this is a good choice, but when joins are involved then the rows matched by different tables involved in the join may not all be on the same node and need to be distributed over the network. Of course, if you join an 'Even' and an 'All' table together, no redistribution is required because the rows of the 'All' table are available everywhere.
- ❖ 'Key' - With a key distribution set, you specify a column to distribute on and then, cleverly, Redshift ensures that all the rows with the same value of that key are placed on the same node.


Key distribution across nodes is really important.


Let's say you had a table of costs, with a column called department code. If you set the table to be distributed across the cluster using the 'department code' column as the key, Redshift will ensure that all the costs for a given department are neatly placed onto a single given node. As such, if you then issue a query asking for the total costs, by department, Redshift knows that each node will return the complete result for a given department. It therefore has no further processing to


do and can resolve the query efficiently.


Another good use would be to optimise a join between two large tables, for instance, a large dimension table, such as 'customers' on an online retail site, and a large sales transaction table on that same site. In this example you could have lots (e.g. millions) of rows in the customers table, and even more (e.g. tens or hundreds of millions) of rows in the sales transactions table.

Here, you don't want to set ALL on either table, because that would be too expensive in terms of storage, and also may impact on the 'Copy' upload process. If you distribute both of the tables on their common key, or keys, then the join will be fast, because all of the keys that they both have in common are guaranteed to already be co-located on the same nodes.

Amazon Best Practice	Our experience
<p>Choose the Best Distribution Style and Key (continued)</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>Our advice</b></p> <p>We typically set the distribution style to ALL for smaller dimension tables, e.g. a date dimension with only a few thousand entries.</p> <p>We set EVEN for tables that are not joined with other tables or are only joined to tables with ALL style specified. For example, a fact table with joins to small dimensions (because each of the small dimensions is already set to 'All').</p> <p>And if we have a very large dimension we will DISTRIBUTE both the dimension and any fact associated with it on their join column. You can (currently) only optimise for a single large dimension, so if we have a second large dimension we would take the storage-hit and distribute ALL, or design the dimension columns into the fact.</p> <p><b>The Verdict</b></p> <p>If you don't get it right it's a killer! So if your query is running very slowly, or more slowly than you would like, optimise here.</p>

Amazon Best Practice	Our experience
<p data-bbox="129 247 475 334">Choose the Best Sort Key</p>  <p data-bbox="139 562 462 591"><a href="#">Go to Amazon best practice</a></p>	<p data-bbox="519 247 719 285"><b>What is it?</b></p> <p data-bbox="519 295 1279 513">Redshift maintains the data in a table in the order of a sort-key-column, if you specify one. This is sorted within each partition not overall, so each cluster node maintains its partition in that order.</p> <p data-bbox="519 562 722 600"><b>Our advice</b></p> <p data-bbox="519 610 1186 697">Choosing a sort key can optimise several different things.</p> <p data-bbox="519 745 1296 1103">The first is data filtering. If your where-clause filters on a sort-key-column, entire blocks of data are skipped. This is possible because Redshift stores data in blocks, and the block header section records the minimum and maximum value of the sort key there. If your filter is outside of that range, the entire block can be skipped. Neat.</p> <p data-bbox="519 1151 1300 1644">Another possible optimisation is when joining two tables when one, or both, tables are sorted on their join keys. In that case, the data can be read in matching order and a merge-join becomes possible without a separate sort-step. If you are joining a large dimension to a large fact table, this is going to help a lot, since neither would fit into a hash-table (the other available efficient join strategy). In fact, in this example, sorting and distributing on the same key is even better.</p> <p data-bbox="519 1692 736 1731"><b>The Verdict</b></p> <p data-bbox="519 1740 1215 1827">Our advice is: Optimise for joins first, then optimise for filtering.</p>

Amazon Best Practice	Our experience
<p data-bbox="137 247 451 378">Define Primary Key and Foreign Key Constraints</p>  <p data-bbox="132 606 455 637"><a href="#">Go to Amazon best practice</a></p>	<p data-bbox="505 247 705 285"><b>What is it?</b></p> <p data-bbox="505 293 1308 378">Telling Redshift about referential relationships in your data, just as you would in a normal RDBMS.</p> <p data-bbox="505 428 705 467"><b>Our advice</b></p> <p data-bbox="505 475 1322 1190">Primary and Foreign Key relationships are understood by Redshift in a similar way to how they are in a conventional database. But unlike in a proper RDBMS, they are not enforced which means you can have duplicate primary key values. And you can have foreign key values that do not match a primary key value. Weird, huh? The reason Redshift is still interested in the primary key/foreign key relationships is that setting them does provide hints to its query planner. As such, it's important that if you do set them, the relationships are correctly implemented, but you have to ensure this yourself in the data, rather than assuming Redshift will do this for you (because it specifically won't).</p> <p data-bbox="505 1240 1308 1642">We currently do not specify primary key/foreign key relationships in the Redshift schema, and we still get good query response times - but we may try this in future to see if we can squeeze out a little more performance. Infer from that what you will - we're pretty hard-core users of Redshift, but then again, maybe when we get around to doing this, we'll kick ourselves that we only just started to implement it.</p> <p data-bbox="505 1692 719 1731"><b>The Verdict</b></p> <p data-bbox="505 1738 1279 1823">Yes, if you're a neat freak. And if it really helps, please let us know.</p>

Amazon Best Practice	Our experience
<p>Use Date/Time Data Types for Date Columns</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What is it?</b> Specifying the correct data type for a date column</p> <p><b>Our advice</b> We're not sure why anyone wouldn't do this... given the rich set of date functions that would be unavailable if you don't store dates as dates.</p> <p><b>The Verdict</b> Our advice - just do it.</p>

## Schema Design Summary


So, as we've seen in the sections above, correct schema design is pretty critical in ensuring your Redshift cluster performs at a high level. Alas, it's not as simple as just 'throwing' your data at Redshift and expecting it to fly. However, with a few simple considerations in the schema design, you can dramatically optimize the performance of your database.

Focus on getting your data in the right place over the cluster to suit your data and query types, as in our experience this is the single most important factor in getting the best out of Redshift.



## Loading Data


The relevant section of the AWS documentation for loading data best practices can be found here:



[http://docs.aws.amazon.com/redshift/latest/dg/c\\_loading-data-best-practices.html](http://docs.aws.amazon.com/redshift/latest/dg/c_loading-data-best-practices.html)

Amazon Best Practice	Our experience
<p>Use a COPY command to Load Data</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What is it?</b> A COPY command is used to get data into a table from various input sources, notably S3 and SSH.</p> <p><b>Our advice</b> Definitely. And have it do automatic column encoding at the same time, as discussed in the previous section.</p> <p>COPY is miles faster than using single line inserts, and as it's likely that you'll be using Redshift with large volumes of data, you absolutely want to be using the COPY command.</p> <p>There are things that actually slow down the COPY that are still good things to do - for example, setting a sort key will slow down the load into a table, but will (potentially) improve the performance of many queries subsequently made against it.</p> <p><b>The Verdict</b> It's really the only way to get large amounts of data loaded in the first place. Use it.</p>



Amazon Best Practice	Our experience
<p>Use a Single Copy command to Load from Multiple Files</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What is it?</b> The COPY command doesn't have to name a single file - it can name a whole bunch of them in one go, which are then used as if they were all one big file.</p> <p><b>Our advice</b> Absolutely.</p> <p>When loading from S3, split up the data into a number of chunks roughly equivalent to (or greater than) the number of slices in the cluster i.e. give every slice some work to do.</p> <p>The load will then happen in parallel, with each slice taking data from its own dedicated file.</p> <p><b>The Verdict</b> Why pay for an 8-node / 16-slice cluster, then force the copy down to a single thread!</p>
<p>Compress your data files with gzip or lzop</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What is it?</b> Input data can be stored in compressed form and decompressed by Redshift during the load (copy) operation.</p> <p><b>Our advice</b> Compression/decompression is a CPU-intensive task, but compared to ferrying gigabytes of data across the network (and especially from remote networks to get data up into S3), we find that the cost of compression/decompression is more</p>

Amazon Best Practice	Our experience
	<p>than paid back in reduced bandwidth and data transfer time. (This really depends on your bandwidth of course).</p> <p>We load data over SSH as well as from S3, and GZIP is still possible then too (and still saves bandwidth). The trick is, when generating the data on the source system, just pipe it through GZIP before returning it to STDOUT (and tell Redshift you've done that on the COPY command of course!).</p> <p><b>The Verdict</b> If you can, you should.</p>
<p>Use a multi-row insert</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What is it?</b> This refers to inserting data using SQL statements, i.e. <i>insert into mytable values (1,2,3), (4,5,6), (7,8,9)</i>.</p> <p><b>Our advice</b> While this might be better than a series of single-row inserts, it is still orders of magnitude slower than a COPY (whether using S3, SSH or whatever).</p> <p>Unless you're dealing with a few hundred rows, then stop being lazy, prepare an input file on S3 (or sort out an SSH command) and bulk-load it!</p> <p><b>The Verdict</b> Avoid inserts, whether they are single or multi-row ones.</p>

Amazon Best Practice	Our experience
<p>Use a Bulk Insert</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What is it?</b> A bulk insert is for creating large tables on the cluster when the input data for that table is already in the same Redshift database.</p> <p>Our advice This is really quick, and you can use the wealth of Redshift functions to help transform your data along the way.</p> <p>In fact, we are starting to upload data from source systems straight into Redshift, and transforming them into star-schemas 'in-place' within Redshift, with pretty fast results.</p> <p>The Verdict If your data is already in the database, and you want to re-spin it into a different format, then use a bulk insert.</p>
<p>Load Data in Sort Key Order</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What Is It?</b> This relates to loading your data into a table in the order specified by its existing sort key. This then simply removes the need for the data to be sorted on the way in, which is potentially a time-saver.</p> <p><b>Our Advice</b> The cost of pre-sorting is probably worse than Redshift sorting, so only really relevant if it's possible to actually generate your data in that preferred order (which is possible for log data and your sort key is transaction's date).</p>

Amazon Best Practice	Our experience
	<p>Of course, the natural ordering of generated data doesn't necessarily make it the best sort key anyway.</p> <p><b>The Verdict</b> Unless your data is already in the right order, or trivially easy to get in the right order, then take the hit and have Redshift sort it for you.</p>
<p>Use Time-Series tables for log-style data or transactional data</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What Is It?</b> This refers to splitting data into multiple smaller tables instead of keeping one enormous one. The natural split for those is time-based (but actually could be anything).</p> <p>Once you have done that, managing that data becomes easier. For example, removing data older than two years simply means dropping some tables - which is much faster than deleting rows within a table, which then involves the 'vacuum' command to reclaim the space, together with a further sort operation.</p> <p><b>Our Advice</b> We don't but probably should - it makes a lot of sense.</p> <p><b>The Verdict</b> Get back to us on this in a few months...</p>

## Loading Data Summary


To combine all of these, you'll want to use COPY to load multiple, compressed files that are already in sorted order. But, sorting is probably best left to Redshift if your data isn't already in that order. However, splitting data into multiple files and compressing it isn't too arduous and will help a lot.


Most of the advice in this section is to optimise loading data, but the schema design will affect the performance of this. For example, setting a sort key will slow down the load but will hopefully make queries faster.


## Querying Data

The relevant section of the AWS documentation for schema design best practices can be found here:


[http://docs.aws.amazon.com/redshift/latest/dg/c\\_tuning-queries-best-practices.html](http://docs.aws.amazon.com/redshift/latest/dg/c_tuning-queries-best-practices.html)

Amazon Best Practice	Our experience
<p>Design for Performance</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What is it?</b> This simply means following all the other best practices when creating your schema.</p> <p><b>Our advice</b> In other words, get the design right in the first place, and don't rely on complicated queries to get over a poor design!</p> <p>Star-schema → good Sort/Dist keys → good if chosen properly</p> <p>Read the 'explain plan' to see if your design is good for your typical queries.</p> <p><b>The Verdict</b> You will want to spend plenty of time getting this right, because once you have a schema with oodles of data in it, it will be time-consuming to change.</p>

Amazon Best Practice	Our experience
<p data-bbox="227 297 358 336">Vacuum</p>  <p data-bbox="132 568 455 596"><a href="#">Go to Amazon best practice</a></p>	<p data-bbox="505 297 705 336"><b>What is it?</b></p> <p data-bbox="505 345 1250 432">As the name suggests, this is a housekeeping activity.</p> <p data-bbox="505 481 812 519">This is required to:</p> <ol data-bbox="505 568 1315 877" style="list-style-type: none"> <li>1. Reclaim disk space from deletes/updates</li> <li>2. Re-sort the data according to the sort key (because if you do several COPY command into a table, each one is sorted on the way in but tagged onto the bottom of the table - so you end up with multiple sorted blocks of data, but not an overall sort).</li> </ol> <p data-bbox="505 935 705 973"><b>Our advice</b></p> <p data-bbox="505 983 1255 1108">This is only required if you maintain data in existing tables. Fresh loads into empty tables don't require it.</p> <p data-bbox="505 1157 1305 1379">It uses a LOT of CPU so will affect performance while it is running. But the biggest limitation is that you can only have one vacuum operation running across the entire cluster at any one time - scheduling them is therefore difficult.</p> <p data-bbox="505 1427 1298 1601">We have implemented a system where vacuums are requested rather than performed immediately, put into a queue, and then carried out in a batch run overnight.</p> <p data-bbox="505 1649 719 1688"><b>The Verdict</b></p> <p data-bbox="505 1698 1305 1785">Avoid them if you can. But if you can't, minimise the impact by doing them out-of-hours.</p>

Amazon Best Practice	Our experience
<p>Configure WLM</p>  <p><a href="#">Go to Amazon best practice</a></p>	<p><b>What is it?</b></p> <p>WLM (Work Load Management) effectively carves up your cluster's available resource into a number of slots, arranges those slots into queues, and then assigns incoming queries to those queues.</p> <p>More slots means more concurrent queries can execute, but each of them will have fewer resources.</p> <p>Queues allow assigning more slots to 'important' queues, and less slots to 'less important' queues to effectively prioritise queries if the system is being asked to serve many queries.</p> <p><b>Our advice</b></p> <p>We use WLM to ensure that several customer queries can happen alongside production data-loads, but only a very small number of queries against test data are allowed concurrently.</p> <p><b>The Verdict</b></p> <p>If you are using Redshift for internal use only, you may not need this at all, but if you have mixed workloads with competing priorities, you will want to ensure more slots are assigned to higher priority queries.</p>
<p>Maintain up-to-date statistics</p>	<p><b>What is it?</b></p> <p>In a nutshell, running Analyze regularly. This scans a sample of rows in each tables and stores summary statistics about them.</p>



Amazon Best Practice	Our experience
 <p><a href="#">Go to Amazon best practice</a></p>	<p><b>Our advice</b>  This can be done during the COPY which means you don't need to think too much about it.</p> <p>Your query plans will change after an Analyze! SQL is declarative, it says nothing about the best way of actually running the query. The database decides how to run the query most efficiently, and without up-to-date statistics it can make poor choices!</p> <p><b>The Verdict</b>  Do it automatically on load and keep an eye on the execution plans. They complain bitterly about missing statistics so it's quite hard to miss.</p>

## Querying Data Summary

Even though this section is about query performance, schema design pops up once again. Get the design right and other things will fall into place.

The difference with Redshift is that schema design includes things that physically alter where data goes in its multi-node architecture, so it's even more important than usual to get it right.

That said, there are a few housekeeping chores (vacuum, analyse, WLM) to keep things tip-top.

# CHAPTER 4:

## Summary



By implementing the best practices described in this e-book, you will ensure your Amazon Redshift implementation flies, and delivers the data warehouse or big data analytics performance that you need to support your business's hunger for answers!

## Next steps:



Matillion, creators of [Matillion ETL for Redshift](#), the high-performance data management tool for Amazon Redshift, offer a free, zero-obligation, 30-minute 'Redshift Setup Assurance' session, where our expert will:

- ✓ Review your Redshift setup
- ✓ Validate your schema and cluster design
- ✓ Make best practice recommendations
- ✓ Answer any questions you might have

Follow this link and complete the form to arrange your free ['Redshift Setup Assurance Audit'](#) session.

**Alternatively**, check out our Amazon partner page for more information:

