REDSHIFT SCENARIO BASED Q&A

BY - SHUBHAM WADEKAR

1. Reports in Redshift are taking twice as long after the dataset grew, but CPU/memory look fine. How would you diagnose and improve performance?

When queries slow down even though CPU and memory are fine, the problem is usually not hardware but how Redshift is handling the bigger dataset.

First, I would check the query execution plan. As data grows, queries may start scanning a lot more blocks or doing expensive data shuffles across nodes. For example, if two large tables don't share a distribution key, Redshift has to move rows between nodes before joining them. That overhead grows quickly as data increases.

Second, I would check whether statistics are updated. Redshift relies heavily on stats to choose the right plan. If a table has grown by millions of rows but stats are old, the optimizer may choose a poor strategy, like broadcasting a table that's no longer small. Running ANALYZE regularly ensures queries use the best plan.

Third, I'd look at table design. If queries filter by date, but the table has no sort key on date, then Redshift must scan all blocks instead of pruning. Similarly, aligning distribution keys on common join columns avoids network shuffles. Choosing the right distkey and sortkey makes queries scale much better as data grows.

Fourth, I'd check for fragmentation and bloat. Updates and deletes in Redshift mark rows as "deleted" but don't free space until a VACUUM runs. If tables are fragmented, Redshift scans extra blocks unnecessarily. Running VACUUM and ANALYZE together often restores performance.

Finally, I would also consider query-level optimizations. For very heavy reports, I might create materialized views or pre-aggregated tables in ETL so reports read summarized data instead of raw fact tables. That way the growth in raw data doesn't double reporting time.

So my approach is: check query plans, update stats, fix dist/sort keys, clean with vacuum, and pre-aggregate where needed. This usually brings performance back under control.

2. Disk space in Redshift is 99% full, but no new large datasets were loaded. How would you troubleshoot and free space?

When disk usage spikes without obvious new data, it usually means space is being wasted rather than genuinely filled.

First, I would check for deleted or updated rows. Redshift is columnar and append-only: deletes don't remove data, they just mark rows as "dead." Until a VACUUM runs, those rows still occupy storage. So a table that looks small logically may consume huge disk space physically. Running VACUUM DELETE ONLY can free that up.

Second, I'd check for temporary or staging tables. ETL jobs often create temp tables or leave behind half-loaded staging tables if a job failed. These tables stay in disk unless explicitly dropped. I'd scan the schemas (especially dev/staging schemas) and remove anything not needed.

Third, I'd check query spill usage. Large queries with joins or sorts spill intermediate data to disk. If workloads grew, spill files can consume lots of space temporarily and not get cleared right away. Monitoring WLM queues and simplifying queries can help here.

Fourth, I'd check compression and encoding. If a column isn't compressed properly, space usage can be 3–4x higher. Running ANALYZE COMPRESSION suggests the best encoding. This can reclaim a lot of storage without deleting data.

Fifth, I'd look at system tables and history. Sometimes, internal history (like STL tables) grows but doesn't clean up if the cluster runs for a long time. Regular maintenance jobs help keep this under control.

Finally, for long-term strategy, I'd move cold/old data into S3 and query it via Redshift Spectrum, so only current data sits in Redshift storage. Or, if growth is ongoing, I'd recommend RA3 nodes with managed storage, which automatically offload cold data to S3 while keeping hot data local.

So to summarize: I'd vacuum to clean ghost rows, drop temp/staging tables, optimize compression, manage query spills, and for long-term, archive old data or use RA3 managed storage.

3. Joins between two 100M+ row tables are slow because they have different sort keys. How would you optimize joins?

I would reduce data scanned and data shuffled before the join. First, I would align table design with how the join runs most often. If the join is usually on a column like user_id or order_id, I would make both tables use the same distribution key on that column so the rows land on the same node and don't have to move during the join. Next, I would give both tables a compatible sort key that helps pruning and merging. If reports filter by date and then join on user_id, I would use a compound sort key like (date, user_id) on both, or at least make sure the first sort key reflects the common filter so fewer blocks are read.

If one of the tables is truly smaller compared to the other (for example a few million rows vs hundreds of millions), I would make it a candidate for broadcast by keeping it compressed and possibly using diststyle all so Redshift can copy it to every node and avoid shuffling the big table. If both are large, I would avoid broadcast and focus on co-location using the same distribution key.

I would also change the query to push filters before the join. Pre-filtering the big tables on date or partition columns reduces the amount of data that needs to be joined. When a report repeatedly runs the same heavy join, I would pre-build an intermediate table or a materialized view where the two tables are already sorted and distributed to match, so the report hits a smaller, join-ready dataset.

Finally, I would clean up storage so the sorted order actually helps. I would run maintenance to resort fragmented tables and refresh statistics so the optimizer chooses the right plan after these changes.

4. One node in the cluster has more data than others, causing uneven performance. How would you fix data skew?

I would fix how rows are distributed across nodes. First, I would check the distribution style and key of the skewed table. If the current key has a few hot values, most of those rows end up on the same node. I would pick a better distribution key with higher cardinality and more even spread, such as a user_id or order_id, so slices receive similar numbers of rows. If there is no good single key, I would use even distribution to spread rows randomly rather than clumping on one node.

If business keys are skewed but I still need to join on them, I would add salting in the ETL. That means adding a small hash or salt column derived from the key and writing multiple salt values per key to spread heavy keys across slices. When joining later, I would compute the same salt on both sides so the join still matches while distribution is more even.

I would then recreate the table using a create-table-as or unload and reload so Redshift redistributes data with the new settings. After that I would run maintenance to resort and analyze the table so the optimizer understands the new row distribution. If queries do not rely on co-location for this table, using even distribution can be the simplest fix to remove skew while keeping joins acceptable.

As a last step, I would look for hidden causes of skew like many nulls in the distribution key or default values that group together. Replacing nulls with a hashed surrogate or avoiding low-entropy keys prevents one slice from getting most of the rows in the future.

5. Some queries return fewer rows than expected, but ETL logs show loads succeeded. How would you investigate?

I would trace the data end to end and verify that the query is looking at the right data with the right filters. First, I would fully qualify the objects in the query to rule out schema mix-ups, for example using db.schema.table instead of relying on search_path. Then I would compare simple row counts between the ETL staging location and the final reporting table for the same date or batch id. If the staging count is higher, something in the load or merge step filtered rows.

Next, I would check for hidden filters. Many teams expose views to BI tools that include where clauses for security or business rules. Row-level filters, security views, or late-binding views pointing to older tables can all reduce rows without anyone noticing. I would run the same query directly on the base table to see if the view is the cause.

I would validate the time filters and time zones. A very common reason is filtering by date using server time while ETL writes in utc. If the report asks for "yesterday" but the table stores utc and the query uses local time, a portion of rows can be missed around midnight. I would rewrite filters with explicit utc conversions or use a date key column.

I would review the join logic. If the report changed from left join to inner join, or added a filter on the right table, valid rows can drop out. I would temporarily remove joins and filters to confirm the base fact count, then reintroduce conditions one by one to find the step that loses rows.

I would check for partial or rejected loads. Even when the ETL "succeeds," a copy command may skip bad rows if maxerror was set. I would look at load error tables and compare expected versus loaded counts per partition or batch id. If using external tables over s3, I would confirm that all partitions are registered or that partition projection is correct; missing partitions look like missing rows.

Finally, I would confirm upsert logic and dedup rules. Merge code that deletes before insert, or dedup keys that are too broad, can drop legitimate records. I would review the match keys and make sure only true duplicates are removed. If everything checks out, I would refresh statistics and rerun the query to rule out an optimizer quirk.

6. An ETL job fails with "Connection timed out" while inserting data into Redshift. How would you fix this?

I would solve this on two fronts: use the right loading method and remove network or configuration bottlenecks. First, I would stop doing row-by-row inserts over jdbc and switch to s3 plus copy. The copy command is the supported bulk path, is resilient to retries, and avoids long-lived client connections that time out.

On the connectivity side, I would verify that the writer can reach the Redshift endpoint. If the job runs in a vpc, I would check security groups, nacl rules, and dns resolution. If it goes through a proxy or firewall, I would allow the cluster endpoint and port, enable ssl if required, and confirm that the subnet has a route to the endpoint. For serverless or private endpoints, I would ensure the correct vpc endpoint is used.

I would harden the client settings. Long operations should have explicit connect and read timeouts, tcp keepalives enabled, and retries with exponential backoff. If a big transaction times out, I would reduce commit size by batching files and committing more frequently so each transaction is shorter. If the loader waits in a queue, I would check workload management rules; long waits can look like timeouts at the client. Putting copy into its own queue, or using the data api to run copy asynchronously, avoids brittle client timeouts.

I would optimize the files for copy. Writing parquet or compressed csv, with 100–512 mb files and a reasonable number of files per batch, keeps copy fast so connections are shorter. I would load into a staging table and then merge into the target; this keeps lock times down and reduces the chance of client timeouts during long transactions.

If the environment is highly elastic or event driven, I would decouple ingestion from loading. Producers write to s3 and enqueue a manifest to sqs; a small, controlled loader drains the queue, runs copy with a fixed concurrency, and retries safely on transient errors. This architecture prevents many parallel writers from competing for connections and eliminates timeouts due to stampedes.

7. How would you migrate a 10 TB MySQL database to Redshift with minimal downtime?

For a database this large, I would plan the migration in two phases: an initial bulk load followed by continuous replication until cutover.

First, I would export the MySQL tables to Amazon S3 in a Redshift-friendly format like Parquet or CSV with compression. This can be done using AWS DMS (Database Migration Service), which supports full load directly into S3, or with tools like mysqldump + AWS SCT to generate schemas and export data. Once the data is in S3, I would use Redshift's COPY command to load it in parallel, making sure to split data into files sized around 100–500 MB each so the load is fast.

Second, to keep the target in sync while the full load is happening, I would enable change data capture with AWS DMS. DMS reads the MySQL binlogs and applies ongoing changes into Redshift. This way inserts, updates, and deletes from MySQL continue to flow to Redshift while users are still on the source system.

Third, I would optimize Redshift schema during migration. I would use AWS Schema Conversion Tool (SCT) to translate MySQL schema to Redshift, making sure to define distribution keys and sort keys based on query patterns, not just copy over MySQL indexes. Compression encodings would also be applied using ANALYZE COMPRESSION so the 10 TB footprint is reduced significantly in Redshift.

When the initial load plus replication is caught up and latency is very low, I would schedule a cutover window. At cutover, I would stop writes to MySQL, let the final changes replicate, then point applications and reporting queries to Redshift. This keeps downtime to just the final switchover.

Finally, I would validate counts and sample queries between MySQL and Redshift to confirm accuracy before decommissioning MySQL as the primary analytics store.

So in simple terms: I would bulk load data to Redshift from S3, use DMS to continuously replicate changes, optimize schema for Redshift, and switch traffic only when sync lag is near zero   minimizing downtime.

8. How would you secure PII data in Redshift so it's encrypted and only accessible to authorized users?

I would secure PII at multiple layers: encryption, access control, and masking.

First, I would ensure encryption at rest and in transit. Redshift automatically encrypts data with KMS keys (AWS-managed or customer-managed). I would enable cluster-level encryption with a customer-managed CMK so only approved accounts/roles can use it. All client connections would use SSL to encrypt data in transit.

Second, I would strictly control who can access PII columns. Using Redshift with AWS Lake Formation or role-based access, I would grant SELECT only to specific roles or users that need it. For example, BI analysts may get access to aggregated views but not the raw PII columns. Fine-grained access can be enforced by granting schema- or column-level privileges.

Third, I would use masking or tokenization where possible. Instead of storing raw emails or phone numbers, I would tokenize or hash them at ingestion and only keep the masked value in analytics tables. When raw PII must be stored, I would put it in separate secure schemas and restrict access tightly.

Fourth, I would audit and monitor access. Redshift integrates with CloudTrail and CloudWatch so every query and connection can be logged. I would set alerts for any unusual queries on sensitive schemas.

Finally, I would build secure views for downstream teams. For example, a masked view could replace full emails with just the domain, or replace names with unique ids. This lets analysts work with data without seeing the raw PII.

So in short: encrypt the cluster with KMS, enforce role-based access to only those who need it, mask/tokenize PII where possible, and monitor with audit logs. This way, PII is protected both technically and procedurally.

9. Multiple teams run queries at once, causing long wait times. How would you solve concurrency issues?

I would separate workloads, give short queries fast lanes, and scale reads without letting heavy jobs block everyone. First, I would enable auto WLM with priorities so different teams or query types get their own queues. Short interactive queries go to a "high" priority queue with fewer slots per query but more concurrency; long ETL/report queries go to a separate queue with tighter concurrency. I would turn on short query acceleration so tiny lookups skip ahead instead of sitting behind big joins.

Next, I would add query monitoring rules to protect the cluster. If a query scans too many bytes or runs too long, it gets moved to a lower-priority queue, throttled, or aborted with a clear message to the team. At the same time, I would push teams to use result cache, materialized views, and pre-aggregated tables, so the BI tools hit smaller objects and return quickly.

For scaling reads, I would use concurrency scaling on the reporting/BI queue so bursts of read-only queries spill to extra transient capacity automatically. If multiple teams are heavy power users, I would isolate them: one producer cluster writes data, and reader teams connect to separate consumer clusters (or serverless workgroups) via data sharing. This way, one team's spike doesn't slow the others. I would also schedule heavy ETL to off-peak hours and make sure queries are efficient: correct dist/sort keys, partition pruning by date, and no "select *" from giant tables.

In short, I would create separate queues with priorities, protect the system with rules, use caches/materialized views, turn on concurrency scaling for bursts, and isolate big readers with data sharing so everyone stays fast.

10. How would you design a Redshift disaster recovery strategy with minimal RTO across regions?

I would combine automatic cross-region snapshots with a ready-to-launch environment in the DR region, and align all dependencies so failover is quick. First, I would enable automated and manual snapshot copy to the DR region using a customer-managed KMS key there. Snapshots run on a schedule and keep recent restore points offsite. I would store all DDL, WLM settings, users/roles, and parameter groups as code (infrastructure as code) so I can recreate the cluster exactly.

For faster recovery, I would keep a pre-created but paused RA3 cluster (or a Redshift Serverless workgroup) in the DR region. During a drill or a real event, I would restore the latest snapshot into that target and resume it, which cuts RTO to tens of minutes instead of hours. If I need even lower RTO, I would continuously land raw and curated data in S3 with cross-region replication, and have ETL jobs ready in the DR region to replay recent increments; in parallel, I would use a CDC pipeline (for example DMS) to feed critical tables to the DR cluster so only a small final delta needs catching up at failover.

Networking, security, and endpoints must be ready: VPC, subnets, security groups, KMS keys, Secrets Manager entries, and IAM roles mirrored in the DR region. I would point BI tools and apps through a DNS name (Route 53 failover) so I can switch endpoints quickly. Finally, I would run regular DR drills that restore from a recent snapshot in the DR region, validate row counts and key reports, and document the steps and timings so the team can execute confidently when needed.

In short, I would copy snapshots cross-region, keep an environment ready to restore, replicate lake data, prepare CDC for critical tables, mirror security/networking, use DNS for cutover, and practice drills to achieve a low, predictable RTO.

11. When migrating a 50 TB warehouse, how would you choose between DC2 and RA3 nodes?

For 50 TB, I would choose RA3 because storage and compute are decoupled. DC2 stores all data on local SSDs, so 50 TB would force me to buy and manage a lot of nodes just to fit data, even if most of it is cold. RA3 uses managed storage under the hood, so I size the cluster mainly for compute and let Redshift keep colder data off the local cache. That keeps cost predictable as data grows.

My decision checklist is simple. If data size is large (tens of TB+), grows quickly, and has a mix of hot and cold partitions, RA3 wins. If I need features like data sharing across clusters, cross-region snapshot copy with simpler restores, and the option to scale compute without reshuffling all data, RA3 again. Concurrency scaling and automatic table optimization also fit better with RA3 in practice because I don't micromanage disk as much.

The only time I'd still consider DC2 is for small, steady datasets that fully fit on a few nodes and where I want the lowest entry cost and can live with resizing when data grows. With 50 TB, that isn't practical: DC2 would need many nodes and frequent resizes. So I'd migrate to RA3, set appropriate distribution/sort keys, compress well, and let managed storage handle the bulk while I tune the cluster for the actual CPU needed by queries.

12. Queries are slow even with low CPU utilization. How would you use slices in Redshift's MPP architecture to troubleshoot?

Each node is split into "slices" that process data in parallel. If queries are slow while CPU looks low, it usually means slices aren't working evenly or they're waiting on I/O, network redistribution, or locks. I'd focus on whether every slice is doing a similar amount of work.

First, I'd check for data skew. If the distribution key is poor (few hot values, many nulls), one slice gets most rows while others sit idle. That shows up as one worker running long while others finish quickly. The fix is to choose a higher-cardinality distribution key that spreads rows, or use even distribution when no good key exists. For extreme hot keys, I add a small "salt" column during ETL to spread them across slices, and compute the same salt on both sides of joins.

Second, I'd look at network redistribution. If two big tables join on columns that aren't co-located, Redshift shuffles data between slices before joining. As tables grow, that shuffle dominates time and slices spend more time waiting than computing. The fix is to align distribution keys on common join columns or make the smaller table broadcastable (keep it small/compressed, or use diststyle all if it's truly small).

Third, I'd make sure pruning works. If sort keys don't match common filters (for example, no date in the sort key while queries filter by date), slices scan far more blocks than needed. Re-choose sort keys to match filters, run maintenance to resort, and keep statistics fresh so the optimizer can prune and pick efficient plans.

Finally, I'd rule out non-slice issues that still keep CPU low: long queue waits due to concurrency limits, locks from ETL running at the same time, or heavy disk spills from poor compression/encodings. I address those by separating workloads into different queues, scheduling ETL off-peak, improving compression, and pre-aggregating hot metrics so slices read less data per query.

In short, I use the slice view to ask: are slices balanced, are we shuffling too much, are we scanning too much, or are we waiting? Then I fix the root cause with better distribution, better sort/pruning, less shuffle, and cleaner workload isolation.

13. Storage is almost full. Would you resize the cluster or switch node types? Why?

I would first check whether the problem is storage-only or both storage and compute. If it's mainly storage pressure and we're on DC2 nodes, I would switch to RA3. RA3 separates compute from storage using managed storage, so I size the cluster for CPU and let Redshift keep colder data in managed storage. That avoids buying lots of nodes just to get more disk. During the move I'd also vacuum, compress correctly, and archive very old data to S3 (queried via Spectrum) to keep the hot working set small.

If we are already on RA3 and queries are healthy, I would not add nodes only for space; managed storage grows automatically and we pay for what we store. I'd still reclaim space by vacuuming deleted rows, fixing encodings, and dropping unused staging/temp tables. If, however, we see compute saturation (high queue waits, long execution even after maintenance), then I'd add RA3 compute (more nodes) to increase parallelism.

So my rule is simple: if storage is the issue and we're on DC2, migrate to RA3; if already on RA3 and compute is fine, don't resize clean up and let managed storage handle growth; if both storage and compute are stressed, add RA3 capacity after doing housekeeping.

14. BI queries slow down at peak hours, but ETL loads are fine. Would you add compute nodes, enable concurrency scaling, or redesign workloads?

I would start by isolating workloads and giving BI its own fast lane. I'd put BI and ETL in separate WLM queues with priorities, then enable Concurrency Scaling on the BI queue so bursty read-only queries get extra transient capacity automatically during peaks. This usually fixes "slow at peak" without permanent nodes.

Next, I'd reduce per-query work so bursts are cheaper: use materialized views or pre-aggregated tables for common dashboards, ensure correct dist/sort keys for BI access patterns, and make sure dashboards hit only needed columns and recent partitions (not select *). I'd also turn on result cache where possible.

If, after isolation and Concurrency Scaling, BI queries are still CPU-bound, I'd add nodes (scale out RA3) to increase baseline parallelism. If different teams contend heavily, I'd consider data sharing to create separate reader clusters (or serverless workgroups) so one team's spike doesn't impact others.

So the order is: isolate BI vs ETL, turn on Concurrency Scaling for BI, optimize BI tables/views, and only then add compute if sustained CPU pressure remains. This keeps costs controlled while maintaining responsive dashboards at peak.

15. How would you design a Redshift cluster for both nightly ETL (5 TB) and daytime analyst queries?

I would size and configure the cluster for two distinct behaviors: heavy batch loads at night and fast, bursty reads in the day. First, I would choose RA3 nodes so compute and storage are decoupled. ETL writes land in S3 and are loaded with COPY in parallel into staging tables sized as 100–500 MB files (Parquet or compressed CSV). After COPY, I would run merge into target tables, followed by analyze and vacuum so daytime queries see fresh stats and sorted blocks.

I would isolate workloads with separate queues. A queue for ETL runs at night with larger memory per slot and lower concurrency, and a queue for BI runs all day with higher concurrency and short query acceleration enabled. I would turn on concurrency scaling for the BI queue, so dashboard spikes borrow transient capacity without permanently adding nodes.

I would design tables for both loading and querying. Distribution keys align the biggest joins to avoid data shuffles, and sort keys follow the most common filters, usually the date column so pruning works. For daytime speed, I would create materialized views or pre-aggregated summary tables the dashboards can hit instead of raw facts. Result cache helps for repeated dashboard queries.

To keep costs and performance balanced, I would schedule the ETL window, ensure automatic table optimization is enabled, and push cold or historical partitions to S3 for Spectrum if analysts rarely touch them. If multiple teams need strict isolation, I would share data from a producer cluster to one or more reader clusters or serverless workgroups so one team's spike does not impact others. This gives predictable nightly loads and responsive daytime analytics.

16. Joins are slow on EVEN distribution. How would you decide whether switching to KEY distribution would help?

I would check how the tables are actually joined and whether co-locating rows would remove big data shuffles. If two large tables frequently join on the same column, and that column has high cardinality and a fairly even value distribution, using the same distribution key on both tables can place matching rows on the same slices so the join happens locally instead of shuffling across nodes.

I would validate the candidate key carefully. It should be stable over time, mostly non-null, and not dominated by a few hot values; otherwise KEY distribution can create skew. If the key is low cardinality or very skewed, EVEN is safer, or I would consider adding a salt column in ETL to spread hot keys while still enabling co-location by joining on key and salt together.

I would run a quick experiment. I would create small copies of both tables with KEY distribution on the proposed column and keep the current versions on EVEN. I would run the representative join query on both and compare the plan and runtime. If the new plan removes large redistribution steps and runtime improves, KEY is the right choice. If one side is actually small enough to broadcast, diststyle all on that small table can be simpler and still avoid shuffles.

If there are many different join patterns with no single dominant key, I would stay with EVEN and optimize elsewhere: pre-filter before joins, pre-aggregate to smaller tables, and use sort keys that match filters so scans are smaller. The decision is based on data shape and query patterns: one big, consistent join on a good, evenly distributed key favors KEY distribution; diverse joins or skewed keys favor EVEN or a mixed approach with broadcast on small tables.

17. One slice is processing more rows than others, causing skew. How would you identify and fix it?

I would first prove skew exists, then change how data is distributed so all slices do similar work.

How I identify it
I compare how much work each slice is doing. If one slice processes many more rows, queries slow down because everyone waits for the slowest slice. I check table health to see if the distribution key is the problem, for example too many nulls or very low cardinality on the key. I also look at recent slow queries to see if the shuffle steps show a big imbalance across slices. If a table with even distribution still shows skew, it is often caused by the join producing a few massive partitions due to hot keys.

How I fix it (simple playbook)

1. Pick a better distribution key. I choose a high-cardinality, well-spread column that is also used to join the biggest tables together. This spreads rows evenly and co-locates join pairs so there is less network movement.
2. Avoid hot or null keys. If many rows have null or a default value, I replace the distribution key with a hashed surrogate of business keys or fill nulls with a hashed placeholder so rows do not pile onto one slice.
3. Use even distribution when no good key exists. If joins are diverse or every candidate key is skewed, I switch to even so the table spreads randomly and removes single-slice hot spots.
4. Salt very hot keys. When a few values are extremely heavy but I must still join on that column, I add a small salt column in ETL (for example 0–7 based on a hash). I write salted rows on both sides and join on key plus salt. This spreads the hot key across slices but keeps the join correct.
5. Broadcast the small side. If one table in the join is truly small, I keep it tiny and use dist style all on that table so it is copied to every node and the large table is not shuffled.
6. Recreate and maintain. After changing distribution, I recreate the table and reload so rows redistribute, then run analyze and vacuum so the optimizer understands the new shape.

In short, I prove the skew, then either re-key to a better column, use even distribution, or add salting to break up hot keys, and I broadcast small tables to avoid shuffling the large one.

18. You have a large fact table and small dimension tables. How would you choose distribution styles (KEY, EVEN, ALL)?

I choose styles to avoid shuffles on big joins while keeping loads and maintenance simple.

Simple rules I follow

1. Fact table: key distribution on the main join column. I pick the surrogate key that most fact joins use (for example customer_id or product_id). This co-locates fact rows with matching dimension rows and reduces data movement during joins. I make sure this key is high cardinality and well distributed; otherwise even distribution can be safer.
2. Small dimensions: all distribution. If a dimension is small and changes infrequently, I replicate it to every node using all. That removes shuffles completely because every slice has a local copy. This is great for classic star schemas where dimensions are a few gigabytes or less.
3. Medium dimensions: match the fact's key. If a dimension is not tiny anymore, I avoid all because it increases maintenance and memory. I distribute it by the same key as the fact so joins stay local.
4. Very large dimensions: even or key, depending on joins. If a dimension is large and joins on a single stable key to the fact, I use key on that column. If it joins on many different columns or the candidate key is skewed, I use even and rely on other tactics such as pre-filtering or pre-aggregating before the join.
5. Think about write patterns. All distribution makes joins fast but makes loads heavier because each node stores a full copy; that is fine for small, slowly changing dimensions but not for frequently updated or medium/large ones. Key and even keep loads lighter.
6. Sort keys are separate. I still choose sort keys for common filters, usually a date on the fact table so scans prune blocks. Sort keys do not change distribution but make queries faster by scanning less.

Typical star schema choice
• Fact: key distribution on the main dimension key and sort by date.
• Small dimensions: all distribution, sort by their natural key.
• Growing dimensions: key distribution on the same join key as the fact to keep joins local.

This approach keeps big joins local, minimizes shuffles, and balances load cost with query speed.

19. A DISTKEY on customer_id causes skew since a few customers generate huge data. How would you fix this?

I would remove hot spots by changing how rows are distributed and, if needed, breaking up the heavy customers.

First, I would switch the fact table to a better distribution strategy. If there is another high-cardinality join key that spreads evenly (for example order_id or session_id) and is used in the main joins, I would use that as the distribution key on both sides of the join. If there isn't a good alternative, I would move the fact table to even distribution so rows spread randomly and no single slice gets overloaded.

Second, I would keep small dimensions fast with dist all. If the customer dimension is small, I would use dist all on it. Then joins between the even-distributed fact and the small dimension happen locally without shuffling large amounts of data.

Third, if I must keep customer_id as the business join but a few customers are extremely hot, I would introduce salting. In ETL I would add a small salt column, for example customer_salt = hash(customer_id) % 8, and store both columns in the fact and any large table that joins on it. Queries join on customer_id and customer_salt together. This spreads a single hot customer across multiple slices while keeping correct matches.

Fourth, I would fix data quality that worsens skew. Nulls or default values in the dist key send lots of rows to the same slice. I would replace nulls with a hashed surrogate so they spread.

Finally, I would recreate and maintain the table. After changing distribution, I would rebuild the table (create-as-select or unload/reload), then analyze and vacuum so the optimizer understands the new shape. This consistently removes the single-slice bottleneck created by a hot customer_id.

20. A query is slower in Redshift than Athena due to data movement. How would you optimize distribution keys/styles?

I would design distribution so Redshift does less shuffling before the join, and let small tables be local everywhere.

First, I would identify the dominant joins. If two large tables frequently join on the same column, I would use the same distribution key on both tables for that column. This co-locates matching rows on the same slices and removes the big redistribute step that makes Redshift slower than Athena for this query.

Second, I would replicate small dimensions. Any table that is truly small and used in many joins should be dist all. That copies it to every node so joins with large facts are local and fast.

Third, if join patterns vary, I would stage for the query. For reports that join on different keys at different times, I would build a small, query-specific staging table with the right distribution key before the final join. This avoids choosing one permanent distribution that is wrong half the time.

Fourth, when no stable, well-distributed join key exists, I would keep the large table on even distribution and push filters early. I would reduce shuffled rows by filtering and pre-aggregating before joining, and ensure join columns have the same data type with no casts or functions so the optimizer can use the distribution correctly.

Fifth, if a few keys are extremely hot, I would use salting to spread those keys across slices. I would add a small salt column in ETL to both sides and join on key plus salt, which reduces single-slice overload during the join.

Finally, I would maintain sort keys for pruning rather than distribution. I would sort facts by the common filter (often date) so scans read fewer blocks. Then I would analyze and vacuum after changes so the optimizer picks the new, low-shuffle plan.

21. A sales fact table filters mostly by date but sometimes by region. Would you use a compound or interleaved sort key? Why?

I would use a compound sort key with date first, region second. Most queries filter by date, so making date the leading sort column gives the best block pruning and the biggest win. Region as the second column still helps for queries that filter by both date and region, because within each date range the data is clustered by region. Compound works best when there is a clear “primary” filter that dominates, which is true here.

I would choose interleaved only if filters are truly balanced across several columns and no single column dominates. Interleaved splits sort weight evenly, which is great for many independent filter patterns but weaker if one column is used most of the time. Since date is the common filter in sales facts, compound keeps performance predictable and maintenance simpler.

To implement, I would rebuild the table with sortkey (sale_date, region) and then run vacuum and analyze so Redshift reorders blocks and updates statistics. I would also check unsorted percentage over time; if it rises due to heavy incremental loads, I would schedule regular vacuum to keep pruning effective.

22. Queries filtering on customer_id scan the entire table. How would you check and fix sort key design?

First, I would confirm what the current sort key is. If the table is sorted by date only, queries that filter by customer_id alone will scan a lot of blocks. I would check table metadata and the query plan to see whether any block pruning happens when filtering by customer_id. If the plan shows a full scan, the sort key is not helping those queries.

Next, I would look at real query patterns. If a meaningful share of queries filter by customer_id, I would consider adding customer_id to the sort key. The simplest option is a compound sort key where the leading column still reflects the dominant filter. For example, if most queries filter by date and some by customer within a date range, I would use sortkey (date, customer_id). That lets pruning on date happen first, and then the blocks are clustered by customer within that range.

If customer_id filters are common without date filters, an interleaved sort can help because it gives equal weight to both columns. Interleaved helps when different columns are used independently for filtering. The tradeoff is that interleaved sort is more sensitive to unsorted data, so I would plan regular vacuum reindex and monitor unsorted percentage.

As a quick check, I would run the same customer_id query on a small sample table sorted by customer_id to see the pruning difference. If it shows large pruning and faster runtime, I would adjust the production table’s sort key accordingly, rebuild with the new sort definition, then vacuum and analyze so the optimizer uses the new order.

23. Frequent UPDATE/DELETE operations slowed performance. How do sort keys and columnar storage affect this, and how would you fix it?

Redshift is columnar and append-only under the hood. When you update or delete, old rows are marked as deleted but still occupy blocks until maintenance runs. Over time this creates bloat and fragments the sort order. A fragmented table scans more blocks and loses the benefit of the sort key, so queries get slower even though data volume hasn't truly grown.

Sort keys work best when new data lands in order (for example, by date). Frequent updates or out-of-order inserts break clustering. Once blocks are unsorted, Redshift can't prune efficiently, so it reads far more data than needed. That's why performance drops after lots of update/delete cycles.

To fix it, I would:

- Reduce row-by-row updates. Load into a staging table, then apply a merge pattern: delete the target slice for the affected keys or date range, insert the replacements, and keep transactions short.
- Use append/replace where possible. Create a new, sorted table with the latest data and swap in with alter table ... append or rename; this avoids rewriting the whole table in place.
- Run regular maintenance. Vacuum delete only to reclaim space from deleted rows, vacuum sort only (or auto) to restore clustering, and analyze so the optimizer has fresh stats.
- Choose sort keys that match how data lands and how it's queried. For facts, sort by the ingest date so inserts stay naturally ordered; keep business updates out of the hot recent range if possible.
- Consider table splitting by time window. Keep the last N months in an "active" table (frequently updated), and freeze older, read-only partitions in separate tables queried via a view; this limits churn to a small object that's easy to vacuum.
- Keep small dimensions small. For frequently updated dimensions, prefer dist all (if truly small) or even, and avoid heavy, wide updates. Tokenize or upsert only changed columns when practical.
- Let automatic table optimization help. On RA3, enable automatic table optimization and compression so Redshift reclusters and re-encodes as needed.

In simple words: updates/deletes create dead rows and break the sort order; reclaim space and restore clustering with vacuum and analyze, shift to merge/append patterns instead of constant in-place updates, and align the sort key with ingest and query patterns.

24. A report query filtering by last 3 months is slow, even with a date sort key. What might be wrong, and how would you fix it?

The usual reasons are that pruning can’t kick in or the table isn’t actually sorted anymore. Common mistakes include filtering with a function on the date column (which blocks pruning), using a different data type than the sort key (implicit casts), high unsorted percentage, or an interleaved sort that hasn’t been reindexed after heavy loads.

What I would check and do:

- Make the predicate sargable. Use a simple range on the raw column, not a function. For example: order_date >= current_date - interval '90 days' and order_date < current_date, not date_trunc('month', order_date) >= ....
- Match data types exactly. If order_date is date, compare to date. Avoid comparing timestamps to dates or casting in the predicate.
- Restore clustering. Check unsorted percentage; if high, run vacuum sort only (or auto vacuum) to re-cluster by date, then analyze. Ensure new loads land roughly in date order so clustering stays healthy.
- Confirm you’re hitting the right table. Views can point to older tables or add hidden filters; test the base table directly.
- Revisit sort strategy. If many queries filter by date and region together, a compound sort key (order_date, region) can help; if many filters run on different columns independently, consider interleaved and schedule regular reindexing.
- Pre-aggregate hot windows. If dashboards always hit the last 90 days, keep a rolling 3-month summary table or a materialized view refreshed daily to avoid scanning the full fact table.
- General hygiene. Keep encodings optimal and avoid tiny unsorted batches during COPY; aim for 100–500 MB compressed files so blocks are well-formed.

In simple words: write the filter as a clean date range with matching types, re-cluster the table so the date sort key actually prunes, verify the view/table you query, and, if needed, refine the sort key or add a rolling summary so the last 3 months stay fast.

25. IoT data queries filter by time, device_id, and location. How would you design sort keys for performance and storage efficiency?

I would start from the most common filter. In IoT, almost every query filters by a time range, so I would make time the first sort column. That lets Redshift prune blocks quickly and read only the recent window.

If most queries also add device_id or location on top of time, I would use a compound sort key with time first and the next most selective column second. Examples:

- If queries are "last 7 days for one device or a small set of devices," I would use sortkey (event_time, device_id). That gives great pruning by time and then clusters rows by device inside each day.
- If queries are "last 7 days for one location across many devices," I would use sortkey (event_time, location).

If filters are truly split across time, device_id, and location (different teams filter by different columns independently), I would consider an interleaved sort key on all three columns. Interleaved gives balanced pruning when any of the columns is used alone, but it needs regular vacuum reindex to stay effective.

For storage efficiency, I would pick the sort order that also improves compression. Sorting by device_id second often creates long runs per device within a day, which compresses well. Location is usually low cardinality, so placing it after time can also group values and improve encoding. After choosing the sort key, I would run analyze compression once and keep loads roughly in time order so clustering stays healthy. If two patterns are equally important (for example, time-first for dashboards and device-first for investigations), I would keep the fact table sorted by time and build a small, device-centric materialized view sorted by (device_id, event_time) for fast lookups.

In short: time first for pruning, then the next most common filter; interleaved only if filters are balanced across columns; keep loads in time order and let compression benefit from the clustering.

26. A 500 GB daily COPY load from S3 is slow. How would you optimize file format, partitioning, and COPY settings?

I would make the files columnar and right-sized, point COPY at just the day's prefix, and run COPY with settings that avoid extra work during the load.

File format and size:

- Write Parquet if possible. It cuts scanned bytes and loads faster than raw CSV. If CSV is required, compress with gzip and avoid very wide rows.
- Target 100–500 MB per file compressed, and produce enough files to feed all slices in parallel (at least a few files per slice). Avoid thousands of tiny files.

Partitioning in S3:

- Organize data by date (and hour if needed) so COPY reads only s3://bucket/path/dt=YYYY-MM-DD/. Do not point COPY at a top-level bucket that forces a huge list operation.
- If file listing is still a bottleneck, use a manifest file for the day's objects so COPY skips the S3 list entirely.

COPY settings and workflow:

- Use COPY FROM S3 with IAM role and FORMAT AS PARQUET (or CSV with the right delimiters and ESCAPE). For big loads, turn COMPUPDATE OFF and STATUPDATE OFF during COPY if encodings are already set; run ANALYZE afterward once per table. This avoids the optimizer work on every load.
- Load into a staging table first, then merge or swap into the target. That keeps locks short and makes retries easier.
- Put COPY in its own WLM queue with enough memory and a small, fixed concurrency (for example 1–2) so it is not competing with BI queries.
- If the dataset has many small late-arriving files, run a compaction job upstream so the daily load is a few hundred good-sized files, not tens of thousands of tiny ones.

General hygiene:

- Ensure Redshift and S3 are in the same region and enhanced VPC routing is configured if you run inside a VPC.
- After load, run analyze (and vacuum sort if needed) so the table stays well-clustered for query speed.
- For multiple large tables, pipeline loads: one table per queue or serialize large COPY commands to avoid saturating the cluster.

In short: Parquet with 100–500 MB files, per-day prefixes or a manifest, COPY with stats/compression updates deferred, a dedicated load queue, and a staging-to-final pattern. This combination consistently turns a slow 500 GB daily load into a predictable, fast bulk ingest.

27. How would you ingest streaming Kinesis data into Redshift near real-time, and what trade-offs would you consider?

I would pick one of three simple patterns based on how "real-time" we need, how much transform we do, and how much ops we want to manage.

1. Kinesis Data Firehose → Redshift (micro-batches)
   Firehose buffers records for a short time or size (for example 1–60 seconds or a few MB), then runs COPY into a staging table in Redshift. I keep the buffer small for low latency, then merge from staging into final tables. This gives near real-time without me writing loaders.
   Trade-offs: very low ops and simple, but it's still micro-batch (seconds). If I set buffers too small, I create many tiny loads and more overhead. I handle duplicates with idempotent merge keys.

2. Kinesis → S3 → Redshift COPY (my own loader)
   Producers write to Kinesis; I deliver to S3 (Firehose or a stream consumer), then a small, controlled loader issues COPY every N seconds/minutes using a manifest. I load into staging, then upsert into final.
   Trade-offs: more control and cheaper at scale, but I own the loader and retries. Latency is usually tens of seconds to a couple of minutes depending on batch size.

3. Redshift streaming ingestion with materialized view (direct from Kinesis)
   Redshift can read from Kinesis directly and materialize into a table with frequent refresh. This removes S3 staging and cuts latency to seconds.
   Trade-offs: lowest latency, but it uses cluster CPU for ingest and needs careful refresh/monitoring. I still design for idempotency and late/out-of-order events.

Common design points I always include:
• Staging then merge: always land into a staging table, then upsert to final using a stable key to prevent duplicates.
• Small, fixed-latency batches: set buffers (or refresh) to seconds, not per record.
• Workload isolation: put loads in their own queue so BI queries aren't impacted.
• Schema evolution: wrap the ingest with a view or a small transform step so new fields don't break COPY/ingest.
• Ordering and late data: if per-key order matters, keep ordering by shard or add event time + sequence; handle late events with a small correction job.

In short, if I want the easiest low-latency path, I choose Firehose → Redshift with tiny buffers. If I need tighter control or extra transforms, I do Kinesis → S3 → periodic COPY. If I need seconds-level latency without S3, I use Redshift streaming ingestion and watch cluster CPU.

28. During a load, some slices process much more data than others. What could cause this imbalance, and how would you fix it?

This happens when work isn't split evenly across the cluster. I look at both the files I'm loading and how the table distributes rows.

Likely causes
• Too few or unevenly sized files: if I load a handful of big files, a few slices get all the work while others sit idle.
• Skewed distribution key: many rows hash to the same slice because the dist key has hot values or many nulls.
• Mismatched join/load path: casting or functions on the dist key during load can send rows to the same hash bucket.
• Tiny files storm: thousands of tiny files cause overhead and some slices finish quickly while others keep getting assigned more pieces.
• One massive "bad" file: a single very large file makes one slice do most of the load.

How I fix it
• Make files "slice friendly": produce many files of similar size (for example 100–500 MB compressed) and at least a few times more files than total slices so all slices stay busy. If listing many files is slow, use a manifest.
• Improve distribution: pick a high-cardinality, well-spread dist key used in joins. If none exists, use EVEN for the large fact table. Replace nulls in the dist key with a hashed surrogate so they don't all map to one slice.
• Use a staging table: load into an EVEN-distributed staging table first (fast and balanced), then CTAS/APPEND into the final table with the intended distribution. This separates "fast load" from "optimal query" layout.
• Compact tiny files upstream: batch on the producer side so daily loads are a few hundred good-sized files, not tens of thousands of small ones.
• Keep types clean: ensure the dist key column types in files match the table exactly (no implicit casts or trimming) so hashing is consistent.

Simple recipe

1. Generate 100–500 MB files, count(files) >> count(slices).
2. Load to EVEN staging, then APPEND into the final table.
3. Choose a better dist key (or stay EVEN) and avoid nulls/hot keys.
4. Analyze after load so the optimizer understands the new distribution.

This consistently evens out slice work, speeds up COPY, and prevents one slice from becoming the bottleneck.

29. A downstream team needs daily results in Parquet on S3. How would you design the UNLOAD for efficiency and low cost?

I would write only the columns they need, in Parquet, into a clean date folder, with files sized for fast reads.

First, I filter and pre-aggregate in Redshift so I don't dump unnecessary rows or columns. Smaller result sets mean fewer S3 files and cheaper Athena scans later.

Second, I unload in Parquet with a daily prefix like s3://bucket/reports/sales/dt=YYYY-MM-DD/. Parquet is columnar and compressed, so it's both faster and cheaper to query than CSV. I also target reasonable file sizes so Athena reads fewer files. In Redshift I control this using maxfilesize and keeping the result parallel:

```
UNLOAD ('SELECT col_a, col_b, col_c
    FROM analytics.sales_daily
    WHERE dt = ''2025-08-24'' ')
TO 's3://bucket/reports/sales/dt=2025-08-24/run_id=123/'
IAM_ROLE 'arn:aws:iam::123456789012:role/redshift-unload'
FORMAT AS PARQUET
PARALLEL ON
MAXFILESIZE 300 MB;
```

Third, I write to a run-specific temp path and then "promote" by copying or updating a small marker object (for example _SUCCESS or latest.json). This avoids readers seeing partial data.

Fourth, I keep partitioning simple: dt=YYYY-MM-DD and, if needed, a second level like region. I avoid over-partitioning because it creates many tiny files.

Fifth, I keep the cluster and S3 in the same region and schedule the unload at the end of ETL, after analyze and vacuum on the source tables, so the query is fast and stable.

Finally, I monitor average file size. If files are too small, I adjust maxfilesize or add a small final coalesce step in SQL (for example, grouping before unload) to reduce file count.

In short: filter and summarize first, unload Parquet into a date-partitioned folder with 100–500 MB files, write to a run path then promote, and keep everything in-region for speed and cost control.

30. COPY fails due to bad JSON records. How would you handle malformed data without breaking the load?

I would load safely into staging, quarantine bad rows, and keep the daily load moving.

First, I land the raw JSON in S3 and load into a staging table, not the final table. I use JSON 'auto' or a JSONPaths file to pick only the fields I expect. I set a small but nonzero maxerror so a few bad records don't fail the whole batch, and I capture errors for review:

```
COPY staging.events_raw
FROM 's3://bucket/events/dt=2025-08-24/'
IAM_ROLE 'arn:aws:iam::123456789012:role/redshift-copy'
FORMAT AS JSON 'auto'
TIMEFORMAT 'auto'
TRUNCATECOLUMNS
ACCEPTINVCHARS
MAXERROR 100;
```

Second, I review load errors in the system error tables to see what failed and why. The good rows are already in staging; the bad ones are identified for quarantine or fix-up.

Third, I normalize from staging to the final table using safe casts. If the payload is semi-structured, I ingest to a SUPER column in staging, then select into the final schema with try_cast so one bad field doesn't break the whole row:

```
INSERT INTO analytics.events_final (...)
SELECT
  try_cast(payload['user']['id']   AS bigint)     AS user_id,
  try_cast(payload['amount']       AS decimal(18,2)) AS amount,
  try_cast(payload['event_time']   AS timestamp)  AS event_time,
  payload['source']::varchar(20)                  AS source
FROM staging.events_raw
WHERE dt = '2025-08-24';
```

Fourth, I quarantine truly bad rows. Anything that still fails validation goes to a quarantine table or an S3 "rejects" prefix with the reason code, so I can reprocess after fixing upstream.

Fifth, I prevent repeats and keep it fast. I validate JSON upstream where possible, ensure files are well sized (100–500 MB), and keep COPY simple and consistent. I avoid row-by-row inserts; COPY in batches is more reliable and easier to retry.

If schema changes often, I prefer SUPER in staging plus a stable projection into the final typed table. That way new fields don't break the load, and I can add them to the final schema on my schedule.

In short: load to staging with tolerant COPY options and limited maxerror, inspect errors, transform with try_cast into final tables, quarantine bad rows, and improve upstream validation so the main load never fails due to a handful of malformed JSON records.

31. Queries are slowing after months of ETL. How would you use VACUUM and ANALYZE, and when would you run them?

Over time, updates/deletes and out-of-order inserts create “dead rows” and break the sort order. That makes Redshift scan more blocks and join more data than needed. I fix this with a regular maintenance cycle:

What I run

- VACUUM DELETE ONLY to reclaim space from deleted/updated rows. This reduces table size and I/O.
- VACUUM SORT ONLY (or a full VACUUM if the table is very fragmented) to re-cluster rows by the sort key so pruning works again.
- ANALYZE to refresh statistics so the optimizer picks good join and scan plans after data changes.

When I run it

- Nightly or after the main ETL load finishes, starting with the largest/frequently queried tables.
- More frequently on hot, fast-growing fact tables; less often on static dimensions.
- Immediately after big backfills or bulk updates.
- For interleaved sort keys, I also run an interleaved reindex periodically, because interleaved needs more frequent maintenance to stay effective.

Operational tips

- Keep maintenance windows short by targeting the worst tables first (highest unsorted or deleted percent).
- Run COPY with data roughly in sort-key order (for example by date) so you need less vacuum.
- Avoid huge in-place updates; prefer stage-and-swap or ALTER TABLE ... APPEND patterns so tables stay well-clustered.
- After vacuuming large tables, run ANALYZE again so stats match the new layout.

Simple summary: reclaim space (vacuum delete), restore clustering (vacuum sort), then refresh stats (analyze), scheduled right after ETL so daytime queries stay fast.

32. A fact table consumes more storage than expected. How would you check and optimize compression encodings?

The usual reason is missing or sub-optimal encodings on wide, large columns. I do three simple things:

How I check

- Review table/column sizes to see which columns are eating space the most.
- Run ANALYZE COMPRESSION on a sample of the table; Redshift tests different encodings and recommends the best per column.
- Look for red flags: VARCHAR without encoding, low-cardinality columns not using RLE/mostly encoding, numeric columns missing AZ64, and large JSON/text blobs stored uncompressed.

How I optimize

- Apply the recommended encodings by creating a new table with those encodings and loading data into it (CTAS), then swapping it in. Re-encoding in place is not supported, so CTAS is the clean path.
- Use column types that compress well. For example, prefer numeric types (DECIMAL, BIGINT) over wide VARCHAR when possible, and trim VARCHAR lengths to realistic max.
- For semi-structured data, land raw into SUPER or stage in Parquet and project only needed fields into the fact table, avoiding giant free-text columns in the fact.
- Keep files right-sized on load (100–500 MB compressed) so blocks form well, which helps compression ratios.

Ongoing practices

- Re-run ANALYZE COMPRESSION after big schema or data profile changes.
- Separate hot, narrow fact tables from rarely used, wide attributes (move those to side tables) so the core fact stays compact.
- Make sure COPY has COMPUPDATE OFF if you already chose good encodings; set encodings explicitly so they don't get reset by accident.

Simple summary: find the biggest columns, get Redshift's encoding suggestions, rebuild the table with those encodings and tighter types, and keep the fact narrow this usually cuts storage by multiples and speeds up scans.

33. ETL queries block reporting queries. How would you configure WLM queues to balance workloads?

I separate ETL and BI into different queues and give each what it needs. ETL gets bigger memory per query but low concurrency; BI gets higher concurrency, short query acceleration, and the ability to burst.

I use automatic WLM with priorities. I create two queues: one for ETL with normal or low priority, and one for BI with high priority. I route by user groups or query groups, so ETL users or ETL-tagged queries never land in the BI queue. I enable concurrency scaling on the BI queue, so daytime bursts of dashboards spin up extra transient capacity without adding permanent nodes.

For BI, I keep many short queries moving. I enable short query acceleration, cap slot usage per query, and set a timeout that is strict enough to stop accidental heavy scans. For ETL, I allow fewer concurrent queries with larger memory, and schedule heavy loads off-peak so they don't overlap with BI when possible.

I add simple guardrails. Query Monitoring Rules move big scans out of the BI queue or abort truly runaway jobs. I also use result cache, materialized views, and pre-aggregated tables so BI reads less data. If multiple teams still contend, I use data sharing to give each team its own reader cluster or serverless workgroup, isolating them completely from ETL.

In short, I classify queries into the right queues, prioritize BI, let BI burst with concurrency scaling, keep ETL constrained and preferably off-peak, and add rules so no single query can block everyone.

34. How would you set up Query Monitoring Rules (QMR) to detect and stop runaway queries?

I define a few clear rules that watch for known bad patterns and take action automatically. I scope rules to the BI queue and to the ETL queue separately so actions match the workload.

I create a scan-guard rule. If a query scans more than a set threshold (for example 200 GB in BI), action is to hop it to a lower-priority queue or abort with a clear message. This protects dashboards from accidental select star on huge tables.

I create a runtime guard. If a BI query runs longer than, say, 5–10 minutes, it's likely not interactive. I downgrade its priority or abort it. For ETL, the threshold is higher, maybe 60–90 minutes, and the action is log or hop, not immediate abort.

I create a row-return guard. If a query tries to return an extremely large result set to the client, I abort it to protect the cluster and the client tool. I also add a nested-loop guard: if a plan shows a large nested loop with high row counts, I abort or hop it because it will likely thrash.

I classify using query groups and user groups. BI tools run with a specific query group so the BI rules apply only there. ETL jobs use a different query group with looser limits. Each rule has an action (log, hop, change priority, abort) and a helpful error message telling users how to fix the query.

I review and tune. I watch rule hits for a week, adjust thresholds to reduce false positives, and keep rules simple: one for scan bytes, one for runtime, one for rows returned, and an optional one for nested loops. With these in place, accidental heavy queries are contained automatically and the cluster stays responsive.

35. A dashboard query joining multiple large tables is slow. How would you diagnose and optimize using system tables, dist keys, and sort keys?

I start by finding where the time is spent, then I remove unnecessary scans and shuffles. First, I run the query with EXPLAIN and look for big redistribute or broadcast steps. In system views, I check SVL_QUERY_SUMMARY and SVL_QRY_PLAN to see bytes scanned, join types, and whether the plan shows DS_DIST_BOTH (both sides shuffled), which is a red flag on large joins. I also look at STL_WLM_QUERY to separate queue wait from actual execution time.

Next, I check table health and layout. In SVV_TABLE_INFO I review diststyle, distkey, sort key, unsorted percentage, and deleted rows. If two large tables join on the same column but use different dist keys, Redshift must shuffle a lot of data; I align them to the same distribution key to co-locate rows. If one side is truly small, I make that table diststyle ALL so it's copied to every node and the big table doesn't shuffle.

Then I reduce scanned data with the right sort keys. If dashboards always filter by a date, I sort big fact tables by that date so block pruning works. If they also filter by a secondary column (like region) within the date, I consider a compound sort key (date, region). If filters vary across several columns, I may try an interleaved sort but keep an eye on maintenance, because interleaved needs periodic reindex to stay effective.

I refresh statistics so the optimizer chooses good plans. After large loads or schema changes, I run ANALYZE on the hot tables. If unsorted percentage or deleted rows are high, I VACUUM SORT/DELETE ONLY so the sort order is restored and dead rows are reclaimed.

Finally, I simplify the query path. I push filters down before joins, pre-aggregate heavy facts into smaller staging tables with the same dist/sort as the final join, and for repeated dashboards I create materialized views so the BI tool reads a precomputed result. Together, these steps remove big shuffles, scan fewer blocks, and make the plan both simpler and faster.

36. Dashboards slow when many users query at once. How would you use concurrency scaling, and what are the trade-offs?

I split workloads and let BI burst to extra capacity only when needed. I place BI queries in their own WLM queue (or workgroup) and enable Concurrency Scaling on that queue. During peaks, read-only queries overflow to transient, automatically managed clusters, so users don't wait behind each other. I keep ETL in a separate queue so loads never block dashboards.

To get the most benefit, I keep BI queries read-only, avoid temp tables in those queries, and leverage result cache and materialized views so the overflow capacity does less work per request. I also set Query Monitoring Rules to catch accidental heavy scans that would waste burst capacity.

The trade-offs are mainly cost and scope. Concurrency Scaling is billed per second when it's used, so I monitor usage and set limits; the good news is you often get free credits each day that cover typical bursts. It only helps read-only queries; write-heavy ETL will not benefit. Plans that rely on local temp tables don't spill over, so I design BI queries to be stateless. There is also a small warm-up effect on the first burst; pre-warming with steady light traffic or keeping critical dashboards on materialized views reduces that impact.

If peaks are constant rather than spiky, I also consider adding RA3 nodes (more baseline parallelism) or using data sharing to give heavy teams their own reader clusters or serverless workgroups. In short, I enable Concurrency Scaling for the BI queue to absorb bursts, keep queries lightweight, cap spend with limits and QMR, and use reader isolation if peaks are frequent or belong to different teams.

37. End-of-month workloads are slow. Would you use concurrency scaling, Spectrum, or Serverless? Why?

I would mix these based on the exact bottleneck, but my default is to first enable concurrency scaling for the BI queue, then use Spectrum for cold history, and use a Serverless reader when I need full isolation.

If dashboards pile up at month-end (many users firing read-only queries), I turn on concurrency scaling for the BI queue. It gives extra read capacity only during the spike, so users don't wait, and I don't pay for extra nodes all month. I keep ETL in a separate queue so loads don't block dashboards.

If month-end queries scan a lot of old data that we rarely touch, I offload that to S3 and query it via Spectrum. The main cluster keeps only the hot months; Spectrum reads the historical partitions in S3. That shrinks local storage and reduces the amount of data the cluster has to scan.

If finance needs a guaranteed lane that won't be affected by anyone else, I set up a Redshift Serverless workgroup (or a separate reader cluster) and share data from the producer cluster. At month-end they run on their own capacity, so there's zero contention, and I pay only when they use it.

So the order I try is: enable concurrency scaling for bursty reads, push cold history to Spectrum to reduce scan size, and use a Serverless reader for strict isolation. Often I use more than one of these together.

38. Storage and compute needs grow quickly. Would you choose elastic resize or classic resize? Why?

I choose elastic resize for quick, low-downtime changes to the same node family, and classic resize when I need a bigger structural change like moving to RA3 or a large change in node count.

If I just need to add or remove a few nodes on the same node type to handle growth or a seasonal spike, I use elastic resize. It completes fast (typically minutes), causes minimal interruption, and lets me scale back later.

If I need to change node type (for example DC2 to RA3), or make a big jump in size that benefits from redistributing data thoroughly, I use classic resize (or create a new RA3 cluster from snapshot). It takes longer because it redistributes data across nodes, but it's the right long-term move when storage is tight or I want managed storage and newer features.

Simple rule: for short-term capacity tweaks on the same node family, elastic resize. For strategic upgrades, especially to RA3 to decouple storage and compute, classic resize or snapshot-restore to a new RA3 cluster.

39. Resize downtime is too long, affecting dashboards. How would you redesign scaling to reduce downtime?

I would separate reads from writes and use scaling methods that don't require a long cluster pause. First, I would move to RA3 if not already there and enable data sharing. Then I keep a small "producer" cluster for ETL and create one or more "reader" clusters (or a serverless workgroup) that only serve dashboards. When I need more daytime capacity, I scale the reader side with elastic resize or spin up an additional reader without touching the producer. Dashboards keep running during producer maintenance.

For short, bursty peaks, I would enable concurrency scaling on the BI queue so extra read capacity appears automatically for minutes at a time. That removes the need to resize at peak and avoids visible downtime. I would also schedule any unavoidable elastic resize for off-hours and keep the change small (same family, a few nodes) to minimize interruption.

For larger upgrades (for example DC2 → RA3 or a big node change), I would do blue/green: restore a snapshot to a new cluster, warm caches/materialized views, test, then switch dashboards via a DNS/endpoint cutover. With data sharing, I can even point readers to the new producer without reloading data. Finally, I keep dashboards fast during transitions by using materialized views and result cache, so a brief backend change doesn't stall user queries.

40. How would you design a cost-optimized scaling strategy using reserved nodes, concurrency scaling, and elastic resize?

I would cover the steady baseline cheaply, burst only when needed, and right-size quickly for seasonal changes. For the baseline, I run RA3 nodes and cover that steady capacity with Reserved Instances or Savings Plans (this locks in a lower rate for what I always use). I keep ETL and core BI on this base.

For short spikes in read traffic (dashboards), I enable concurrency scaling on the BI queue. It gives extra capacity only for the minutes I need it, often covered by daily free credits. I put Query Monitoring Rules in place so accidental heavy scans don't burn burst capacity.

For predictable seasonal changes (month-end, holiday season), I use elastic resize to add or remove a few RA3 nodes quickly with minimal interruption, then scale back after the peak. If teams need strict isolation, I share data from the producer to a small reader cluster or serverless workgroup sized for typical days; during peaks, I elastic-resize that reader or rely on its concurrency scaling so the producer stays steady.

To keep the footprint small, I offload cold history to S3 with Spectrum, maintain good compression/encodings, and use materialized views so queries do less work. For dev/test, I schedule pause/resume to avoid paying when idle. Net result: baseline on reserved, bursts via concurrency scaling, seasonal shifts via elastic resize, with data sharing and storage optimizations to keep costs predictable and low.

41. Analysts want to query both S3 clickstream data and Redshift fact tables. How would you design a hybrid with Spectrum?

I would keep raw/big clickstream data in S3 (cheap and scalable) and keep high-value, curated facts in Redshift, then join them through Redshift Spectrum. The flow is simple:

1. Store clickstream in S3 as Parquet, partitioned by date (and hour if needed), using Hive-style folders like s3://.../clicks/dt=YYYY-MM-DD/hr=HH/. Keep files 128–512 MB so scans are efficient.
2. Use the Glue Data Catalog as the shared metastore. Create an external schema in Redshift that points to Glue, and define external tables for the clickstream paths.
3. Expose analyst-friendly views in Redshift that join local fact tables to the Spectrum external tables. Push filters into the clickstream side first (time range, site, device) so Spectrum reads only the needed partitions before the join happens.
4. Keep Redshift facts optimized for joins and pruning (distribution key matching the common join, sort key on date). Keep Spectrum tables narrow: only the columns analysts need for joins and filters.
5. Control cost and consistency. Always write to a run-specific S3 path, then promote to a stable dt=.../hr=... location with a success marker so queries never hit partial data. Consider materialized views in Redshift for the most common hybrid joins so dashboards don't re-scan S3 every time.
6. Secure the lake. Use Lake Formation/IAM to grant only needed columns and partitions to Redshift's role. Encrypt S3 with KMS and keep Redshift and S3 in the same region.
7. Operate smoothly. Partition projection for wide date ranges avoids listing millions of partitions. Schedule periodic compaction of tiny clickstream files. Refresh statistics in Redshift after large fact loads so the optimizer chooses good hybrid plans.

This keeps raw volume in S3, curated facts in Redshift, and lets analysts run one SQL that joins both efficiently.

42. Spectrum queries scan too much data despite partitions. What might be wrong, and how would you optimize pruning?

This usually means pruning can't activate because of metadata or query patterns. I would check these common issues:

1. Wrong partition layout or registration. Ensure Hive-style folders (dt=YYYY-MM-DD[/hr=HH]) and that partitions are registered (ALTER TABLE ADD PARTITION or partition projection). If the table LOCATION points too high (the parent folder), Spectrum may read everything.
2. Filters not sargable. Pruning fails if you wrap the partition column in functions or casts. Use clean ranges like dt >= '2025-08-01' AND dt < '2025-08-25'. Avoid casting timestamps to strings or vice versa in the WHERE clause. Make partition columns in the table the same data type you filter on.
3. Partition column not used. Queries that filter on non-partitioned columns force scans. If analysts often filter by hr or region, add those as partitions (within reason) or create a second external table more aligned with that access pattern.
4. No partition projection. With very many daily/hourly partitions, listing them is slow and sometimes incomplete. Define partition projection with ranges and formats so Spectrum computes partitions instead of listing S3.
5. Mismatched case or hidden characters. Folder names or column names with case mismatches, spaces, or hidden characters can break pruning. Keep names lowercase and consistent.
6. Over-casting in joins. Joining on expressions (for example cast(dt as timestamp)) prevents pushdown. Pre-normalize types so joins and filters are on raw columns.
7. Tiny or poorly formed files. Millions of tiny files or unsorted Parquet hurt min/max stats and pushdown. Compact to 128–512 MB and write Parquet with proper types so column stats work.
8. Selecting parent prefixes accidentally. Using wildcards or a LOCATION that spans multiple datasets forces scans of unrelated partitions. Point external tables at the tightest prefix, and for ad-hoc ranges use manifests to load only specific keys.
9. Missing or stale catalog. If new partitions weren't added (and no projection), Spectrum can't find the narrow set and may scan broader areas. Automate partition adds or rely on projection.

To fix: clean up table LOCATION and partition columns, rewrite filters as simple ranges on partition columns with matching data types, enable partition projection for large ranges, compact small files to Parquet, and keep the catalog accurate. After that, Spectrum should prune to just the requested date/hour/region instead of scanning the lake.

43. Schema changes in S3 must auto-reflect in Spectrum queries. How would you use Glue Data Catalog for this?

I would make the Glue Data Catalog the single source of truth and keep it updated automatically, so Redshift Spectrum always sees the latest schema. First, I would create external tables in Redshift that point to databases and tables managed by the Glue Data Catalog. Second, I would attach a Glue Crawler to each dataset path in S3 (for example, clicks/dt=YYYY-MM-DD/). I would schedule the crawler after every landing window (or trigger it on S3 PUT events via EventBridge) so it adds new partitions and new columns automatically. For stable evolution, I would use columnar formats like Parquet, because adding columns is naturally supported; I would avoid breaking changes like type narrowing or renaming columns.

For very large partition counts, I would enable partition projection on the table (date, hour, region), so Spectrum computes partitions from rules instead of listing S3, while the crawler focuses on schema (columns and types) rather than enumerating every partition. To protect downstream queries, I would put a thin view layer in Redshift that selects only the approved columns; when new columns arrive, the view can be updated in a controlled way, while power users can query the base table immediately. For governance, I would manage permissions in Lake Formation so only the right roles see new columns (especially if they contain sensitive data).

In short: Spectrum uses the Glue Catalog; a scheduled or event-driven Glue Crawler keeps schemas and partitions fresh; partition projection handles scale; Parquet makes column additions painless; and a view layer plus Lake Formation gives safe, controlled rollout of changes.

44. Spectrum queries are slower than native Redshift. What are the bottlenecks, and how would you improve?

Spectrum reads data from S3 at query time, so the main bottlenecks are S3 I/O latency, too many or poorly sized files, weak partition pruning, and heavy joins between external and internal tables. If filters use functions or casts on partition columns, pruning fails and Spectrum scans far more data. Uncompressed CSV or millions of tiny files also slow scans. Cross-region S3 access, KMS misconfiguration, or joins that force large shuffles back into the cluster add more delay.

I would fix this in a few steps. First, I would store data as Parquet or ORC with snappy and target 128–512 MB files; I would run a periodic compaction job to merge tiny files. Second, I would design clean Hive-style partitions (dt=YYYY-MM-DD[/hr=HH][/region=...]) and make queries use simple range predicates with matching data types; when partitions are very large in number, I would turn on partition projection. Third, I would push filters and pre-aggregations to the external side before joining to large Redshift tables, and I would avoid casts or functions on join/filter columns so pushdown works. Fourth, I would keep S3 and Redshift in the same region and use VPC endpoints; I would confirm IAM/KMS permissions are not causing retries.

For recurring queries, I would consider pulling the hot slice of S3 data into native Redshift tables on a schedule (CTAS or COPY) or materializing a summary table inside Redshift so dashboards do not hit S3 each time. I would also review distribution and sort keys on the Redshift tables used in the join so the cluster does minimal shuffling when combining with the filtered external data. In short: use columnar formats and right-sized files, enforce partition pruning, push filters early, keep data in-region, and cache or ingest hot data into Redshift when low latency is required.

45. How would you join real-time S3 data via Spectrum with Redshift fact tables while managing schema, cost, and performance?

I would keep raw, fast-arriving data in S3 as Parquet and expose it to Redshift through Spectrum, then join it to Redshift fact tables using views or materialized views. The key is to write only finished, partitioned data to S3, keep the Glue Catalog up to date, and push filters before the join so Spectrum reads as little as possible.

My simple design:

1. Land and partition the stream. Kinesis or a small writer batches records to S3 in Parquet under a clean layout like s3://.../events/dt=YYYY-MM-DD/hr=HH/. Every micro-batch writes to a temporary run path, then promotes to the final partition and drops a success marker. This avoids partial reads.
2. One catalog for both sides. Glue Data Catalog holds the external table for S3 and the schemas for Redshift COPY'ed tables. A Glue Crawler or explicit DDL updates new columns and adds partitions right after each batch. For very large partition counts, I use partition projection so Spectrum doesn't have to list S3.
3. Make joins cheap. In Redshift I create a view that first filters the Spectrum table by time and any other selective predicates, then joins to the local fact. If the local fact is big, I align distribution keys with the join column so the join is local. If the dimension is small, I replicate it (dist all) so the join doesn't shuffle.
4. Keep latency predictable. I use small, fixed batch windows (for example 30–120 seconds) so analysts see near real-time but we still write right-sized Parquet files (128–512 MB). If a dashboard needs sub-minute latency, I precompute a tiny hot window inside Redshift (CTAS or materialized view) from the last few partitions and let the main view union that with older S3 data.
5. Control schema drift and PII. The writer enforces an allowed schema (additive changes only). New columns are nullable and documented. Sensitive fields are tokenized before landing. I add a thin Redshift view that selects only approved columns so a sudden new column doesn't break dashboards.
6. Cost and performance hygiene. Keep S3 and Redshift in the same region, use Parquet with proper types, compact tiny files periodically, and push time filters to the Spectrum side. For recurring heavy joins, materialize a summary inside Redshift on a schedule so dashboards don't re-scan S3 each time.

In short: write small, finished Parquet batches to S3, keep the Glue Catalog current, filter first on the Spectrum side, align distribution on the Redshift side, and materialize common joins to keep both cost and latency low.

46. Both Redshift and Athena need to query the same schema in S3. How would you design the pipeline for consistency?

I would make S3 the system of record, Glue Data Catalog the single schema source, and write data in a way that is atomic, partitioned, and consistent for both engines.

My simple plan:

1. One catalog, many readers. All S3 tables are defined in Glue Data Catalog. Athena reads them directly; Redshift creates external schemas that point to the same Glue databases. Permissions are managed in Lake Formation so both engines see the same columns and partitions, with row/column controls if needed.
2. Atomic, partitioned writes. Every batch writes to a temporary run path in S3, then promotes to a dt=...[/hr=...] partition and drops a success marker. Writers never overwrite an active partition in place. This prevents either engine from seeing half-written data.
3. Schema evolution rules. The writer enforces additive changes only (new nullable columns). When a new column appears, the pipeline updates the Glue schema (crawler or DDL). A compatibility check fails the run if a breaking change is attempted. Downstream, I use views to expose only approved columns so tools aren't surprised by new fields.
4. Columnar, right-sized files. Data lands as Parquet with correct types, 128–512 MB per file, and no millions of tiny files. This keeps both Athena scans cheap and Spectrum fast.
5. Partition management at scale. For daily/hourly layouts with many partitions, I enable partition projection so both Athena and Redshift avoid listing S3. If projection isn't used, a small job adds partitions immediately after promote.
6. Consistency pointers. Consumers read only partitions that contain a success marker or a manifest. If late data arrives, the writer opens a new run path for that partition and promotes it when done; a versioned pointer (for example latest.json) tells both engines which run is canonical.
7. Optional ACID table format. If I need upserts, deletes, or time travel, I consider a lake table format that both engines support in the environment. That gives transactionality and simpler schema evolution while keeping the Glue Catalog as the metastore.

In short: single Glue Catalog, atomic partitioned writes with success markers, additive schemas, Parquet with good file sizes, partition projection for scale, and consistent pointers so Athena and Redshift always read the same, finished data.

47. How would you integrate Redshift with SageMaker for ML feature pipelines, and what are the cost/performance considerations?

I would keep heavy joins and aggregations inside Redshift (it's great at SQL over large facts), then hand only the final, narrow feature set to SageMaker for training and online serving.

Simple design I would explain:

1. Build features in Redshift. Create SQL or stored procedures that produce a clean feature table (one row per entity/time). Use proper dist/sort keys and materialized views for daily refresh. This keeps scans small and repeatable.
2. Export features to S3 in Parquet. Use UNLOAD to write only needed columns into s3://.../feature_set/dt=YYYY-MM-DD/ with 128–512 MB files. This becomes the offline data for SageMaker training and SageMaker Feature Store.
3. Train and validate in SageMaker. A Pipeline (or Processing job) reads the Parquet features from S3, does any light normalization, trains models, logs metrics, and writes artifacts back to S3/Model Registry.
4. Serve features two ways:
   • Offline store (batch): analysts and batch scoring jobs read the same Parquet via Athena/Redshift Spectrum.
   • Online store (real-time): write the subset of low-latency features to SageMaker Feature Store online (DynamoDB) keyed by entity_id, or to a small purpose-built cache. Keep a job that materializes fresh features from Redshift into the online store on a schedule.
5. Score from Redshift when needed. For batch inference inside SQL, either UNLOAD features to S3 and score in a SageMaker Processing job, or use Redshift ML if the model fits that path (train in SageMaker Autopilot behind the scenes and use PREDICT in SQL). Use this for periodic scoring, not per-row online calls.

Cost and performance points I'd call out:

- Push work down: do wide joins/aggregations in Redshift; ship only a narrow, final feature set to S3. This lowers S3 I/O and training time.
- File/layout matters: Parquet + partition by date/entity segment keeps SageMaker I/O fast and cheap. Avoid thousands of tiny files.
- Isolate workloads: run ETL and exports on a producer cluster; analysts train in SageMaker without loading the Redshift cluster. Use data sharing or Spectrum to avoid duplicate storage.
- Online cost: only publish features needed for real-time to the online store; keep TTL/compaction so DynamoDB costs don't explode.
- Avoid chatty per-row calls: don't call SageMaker endpoints from Redshift per row. Batch score outside SQL or materialize predictions back into Redshift for BI.
- Governance: Glue Data Catalog for the offline store schema, Lake Formation for access, and KMS everywhere.

In short, Redshift builds the features, S3 is the handshake, SageMaker trains/serves, and we keep data narrow, columnar, and partitioned to control cost and latency.

48. How would you use Lambda UDFs in Redshift to trigger downstream workflows automatically?

I would use a Lambda UDF as a small, reliable "notifier" that fires once per batch at the end of an ETL step, not per row.

How I'd set it up:

1. Create a Lambda that accepts a JSON payload (dataset, date, run_id, S3 path) and publishes to SNS/SQS/EventBridge. Downstream subscribers (Glue job, Step Functions, Airflow) start the next task.
2. Allow Redshift to invoke it. Grant the cluster IAM role permission to invoke the Lambda. Then create the external function in Redshift that wraps this Lambda and takes a few scalar params.
3. Call it from a stored procedure at the end of the transaction. After COPY/merge completes and validations pass, the proc calls the Lambda UDF exactly once with a compact payload. If the proc rolls back, the call doesn't happen, so consumers never see half-finished data.
4. Make it idempotent. Include run_id and partition in the payload. The Lambda writes an idempotency key to DynamoDB and no-ops if it has seen it before. This prevents duplicate downstream triggers if someone re-runs the proc.
5. Handle errors safely. Have the UDF call inside a small try/catch block in the proc. On failure, log to an audit table and surface a clear error, but don't fire partial notifications. Lambda itself retries to SNS/SQS automatically; you also get CloudWatch logs.
6. Control rate. Never invoke per row; invoke once per table or partition. If you must send many items, the Lambda publishes a single message with a manifest (list of partitions/paths) that downstream reads in bulk.

Limits and cautions I would state:

- Lambda UDFs have payload and time limits; keep payload small and the Lambda fast. Use queues for fan-out.
- Don't put heavy logic in the UDF; it's a trigger, not a processor.
- Monitor invocations and failures; add alarms on the Lambda and on the audit table in Redshift.

In short, I call a Lambda UDF once per successful batch to publish a small event, downstream systems subscribe and run next steps, and idempotency keeps the workflow safe even on retries.

49. How would you synchronize metadata between Glue Data Catalog and Redshift to keep schemas updated?

I keep one “source of truth” and automate both directions so people never see stale schemas. My simple plan is:

1. Pick the source of truth per zone
   For lake data on S3, Glue Data Catalog is the source of truth. Redshift references it via external schemas (Spectrum). For native Redshift tables, Redshift is the source and Glue learns from it via a crawler.
2. Automate S3 → Glue updates
   Every time new data lands in S3, I update Glue automatically. I either run a Glue Crawler after each batch (or trigger it with EventBridge on S3 PUT) or, for very large partition counts, I use explicit DDL/partition projection so I don’t have to crawl millions of folders. Column changes must be additive (new nullable columns) to stay backward compatible.
3. Reflect Glue in Redshift automatically
   Redshift Spectrum points to Glue databases with an external schema. When Glue updates (new columns or partitions), Redshift sees the changes immediately no extra sync step needed. For performance, I still keep queries using simple range filters on partition columns.
4. Keep Redshift-native tables in Glue for discovery
   I run a Glue Crawler against Redshift using the Redshift connection. It picks up new tables and columns in selected schemas (for example analytics, marts) so Athena/other tools see the same catalog entries. I schedule it nightly or after DDL migrations.
5. Control changes with IaC and views
   I store table DDLs (both Glue external tables and Redshift CREATE TABLE) in version control and apply via pipelines, so schema changes are reviewed. For consumers, I expose stable views (in Redshift and Athena) that list only approved columns. When a new column appears in Glue, I can add it to views when ready, avoiding surprise breakages.
6. Governance and security
   I use Lake Formation on top of the Glue Catalog to grant column/row-level access once, and both Athena and Redshift honor the same permissions. KMS is used for S3 encryption; Redshift has its own KMS key. This keeps access consistent across engines.

Result: S3 schemas evolve in Glue and flow straight into Redshift Spectrum; Redshift-native tables are crawled into Glue for cross-tool visibility; everything is driven by code and guarded by Lake Formation.

50. How would you design Redshift UNLOAD to S3 so that data is partitioned and queryable in Athena/Glue?

I export only what's needed, in Parquet, into Hive-style folders, and register partitions so Athena can query instantly.

1. Shape the dataset first
   I pre-filter and pre-aggregate in Redshift so I unload a narrow result set (fewer columns/rows). This reduces S3 files and future scan cost.
2. Write Parquet with right-sized files
   I use UNLOAD FORMAT AS PARQUET with PARALLEL ON and MAXFILESIZE around 128–512 MB so downstream reads are fast and cheap.
3. Create Hive-style partitions
   Athena expects folder keys like dt=YYYY-MM-DD/region=US. UNLOAD can't "auto-partition" into multiple folders in one command, so I loop per partition (for example per dt or per dt+region) from a stored procedure or orchestration step. Each loop runs one UNLOAD to the exact prefix, for example:

- s3://bucket/mart/table/dt=2025-08-24/
- s3://bucket/mart/table/dt=2025-08-24/region=US/

4. Use a run path then promote
   Each UNLOAD writes to a run-scoped temp prefix (includes run_id). After all partitions succeed, I copy or move objects to the final dt=.../region=... paths and drop a _SUCCESS marker. Consumers only read partitions that have _SUCCESS to avoid partial data.
5. Register partitions in Glue (or use projection)
   Option A: after promote, call ALTER TABLE ADD PARTITION for each new partition (simple when daily).
   Option B: enable partition projection (date/region ranges) so I don't need to add partitions each day Athena and Spectrum compute them from naming rules.
6. Keep schemas consistent
   I keep a Glue table defined with the exact column types that match the UNLOAD output. If columns change, I update the Glue table schema in code first, then UNLOAD with the new shape. Views in Redshift/Athena expose only approved columns to downstream users.
7. Performance and cost hygiene
   Keep Redshift and S3 in the same region. Limit UNLOAD to the day you're exporting (not the whole table). Use a dedicated queue in Redshift for UNLOAD so it doesn't fight with BI. Monitor average file size and adjust MAXFILESIZE or degree of parallelism to avoid tiny files.

End result: partitioned Parquet under dt=/region= folders, produced via per-partition UNLOADs, promoted atomically, and registered in Glue (or projected), so Athena and Redshift Spectrum can query immediately with minimal cost.