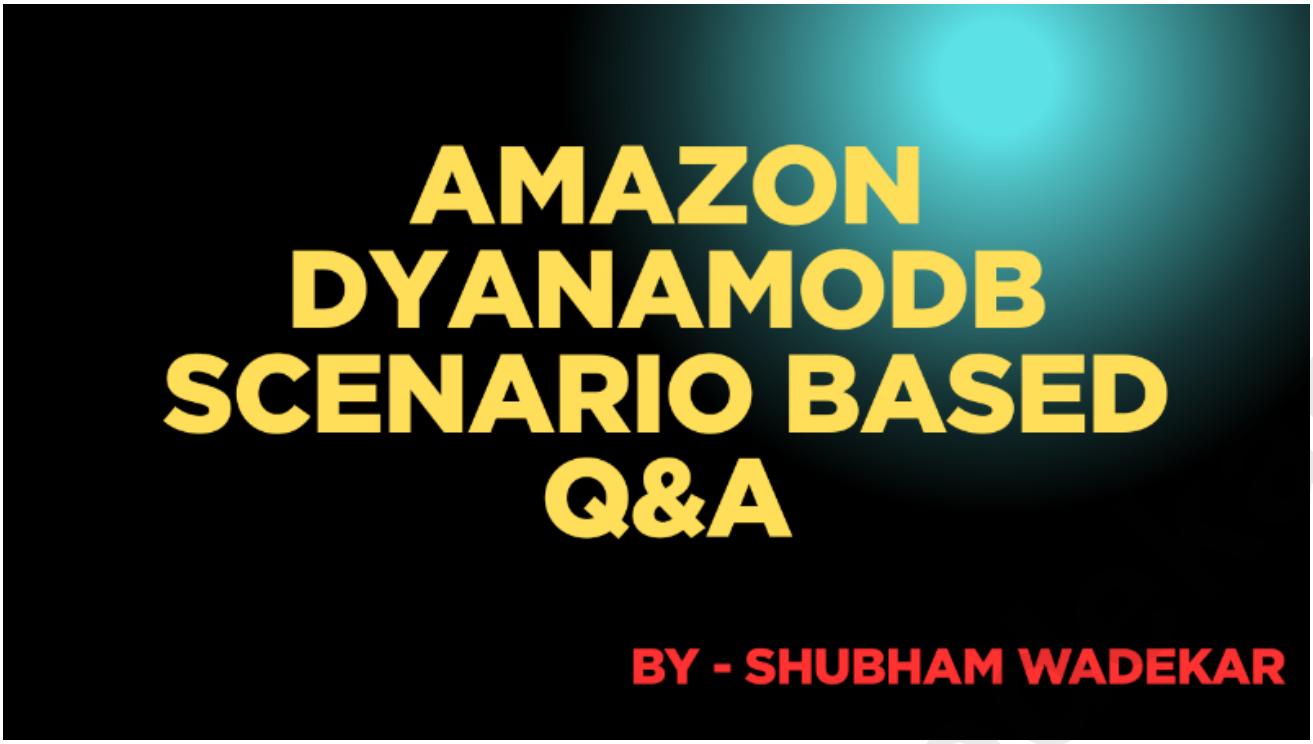

1. How would you design a DynamoDB table for a multi-tenant application?

I start from the access patterns and pick a single-table design that keeps every tenant's data together by partition key, while enforcing isolation and avoiding hot partitions.

1. Choose a primary key that scopes all data by tenant
   Partition key: tenant_id
   Sort key: entity_type#entity_id (for example, customer#123, order#2024-08-01#0001).
   This gives me fast, isolated queries like "all orders for tenant T" (Query by tenant_id with begins_with on order#).

2. Model the common access patterns up front
   • Get an entity by id → PK = tenant_id, SK = entity_type#id
   • List entities by type → Query PK = tenant_id, SK begins_with entity_type#
   • List tenant's orders by date → use SK = order#yyyymmdd#order_id so I can range-scan by date
   • Search across all tenants (admin/reporting) → add a sparse GSI, for example GSI1 with PK = entity_type, SK = yyyymmdd#tenant_id#id. Lock this GSI down so only admin roles can query it.

3. Prevent key collisions and keep items small
   Prefix sort keys with entity_type, keep payloads under 400 KB, and push large blobs to S3 with an s3_uri pointer. Store computed aggregations (for example, order_count) as separate items to avoid scans.

4. Handle noisy neighbors and fairness
   Use on-demand capacity to start, or provisioned with auto scaling. If one tenant is very hot, I either:
   • give them their own table (strong isolation and autoscaling per tenant), or
   • shard writes within that tenant by adding a small random suffix to a "bucketed" SK (for example, order#yyyymmdd#b03#ts) to spread write I/O.

5. Enforce security at the row level
   Issue each tenant an IAM role or identity that includes a Condition on dynamodb:LeadingKeys = ["${tenant_id}"] so they can only touch items where PK = their tenant_id. Combine with Lake Formation/Athena if I export to S3. Encrypt the table with KMS (customer-managed key) and restrict kms:Decrypt to the app role.

6. Global secondary indexes (only what you need)
   Add GSIs for cross-entity lookups you truly need, e.g., GSI2: PK = tenant_id, SK = status#updated_at for "all open tickets newest first". Keep GSIs sparse by writing only items of that type to them.

7. Operations and governance
   Turn on point-in-time recovery (PITR), streams for CDC to downstream systems, and per-tenant usage metrics in your app so you can spot abuse and right-size capacity. Use Standard-IA table class only for cold, rarely accessed data.

Result: a single table keyed by tenant, with GSIs for critical cross-tenant or alternate lookups, IAM row-level isolation, and optional sharding or per-tenant tables for very hot tenants.

2. How do you efficiently handle time-series data in DynamoDB?

I design the keys to support fast range queries over recent time windows, control write hot spots with bucketing, and manage data lifecycle with TTL and exports.

1. Pick keys that fit range queries
   If the main pattern is "get last N events for a device":
   Partition key: device_id (or metric_id)
   Sort key: yyyymmdd#epoch_ms (or epoch_ms padded)
   Now I can Query device_id with SK BETWEEN start_ts and end_ts, or use begins_with on the day prefix.

2. Avoid hot partitions with write bucketing
   If a single key gets very high write rates, add a small bucket to spread writes:
   PK = device_id#b{00..15} (choose bucket = hash(event_id) % 16)
   SK = epoch_ms
   To read the latest data, query a few buckets in parallel and merge results in the app. For moderate rates across many devices, plain device_id PK is fine because traffic is naturally distributed.

3. Fast "latest record" lookups
   Option A: write a small "latest" item per device (PK=device_id, SK=latest) that you overwrite on each write.
   Option B: add a GSI with PK = device_id, SK = inverted_ts (for example, 9e12 - epoch_ms) and set Limit=1 to fetch the newest item.

4. Keep items lean and move cold data out
   Store small JSON attributes; put big payloads in S3 and reference them by URI. For retention, set TTL on an expires_at attribute so old items auto-delete. Use on-demand backup/export-to-S3 or Streams → Kinesis Firehose to land data in S3 for analytics.

5. Query shapes you'll support
   • Last 15 minutes/hour/day per device → range query on SK between timestamps
   • Aggregations (counts, averages) → precompute and store rollups per time bucket as separate items (PK=device_id, SK=rollup#yyyy-mm-dd#hour#metric), updated transactionally with the raw event if needed.
   • Multi-device dashboards → fan out queries by device ids in parallel, or maintain a secondary index keyed by region/tenant for top-N use cases.

6. Capacity and throttling
   Start with on-demand, then switch to provisioned with auto scaling if traffic is predictable. With bucketing, choose bucket count high enough to keep per-partition write rates below limits, but not so high that reads have to fan out to many partitions.

7. Data quality and ordering
   DynamoDB doesn't guarantee exact time ordering for same-millisecond writes. Include a tiebreaker (for example, a sequence or unique event_id) in the sort key or as an attribute to ensure deterministic merges on the client.

8. Global tables if you need multi-region
   For low-latency reads globally, use Global Tables and keep conflict probability low by partitioning writes by device (each device writes to one region ideally) or by using idempotent upserts.

Result: time-series data is modeled for O(1) writes and O(log n) range reads on time, hot partitions are avoided with buckets, storage stays lean with TTL and S3 offload, and "latest" as well as window queries are efficient and predictable.

3. How would you model a many-to-many relationship in DynamoDB?

I use a single-table design with "association" items and, typically, one GSI to get fast lookups from both sides without duplicating data.

1. Pick clear entity keys
   Use composite keys that encode the entity type. Example for users ↔ groups:
   • PK for user items: USER#<user_id>
   • PK for group items: GROUP#<group_id>
   • SK for the main entity row: PROFILE (or META)

2. Add association items (the join table)
   Store one lightweight item per relationship under the *user* partition so I can list a user's groups quickly:
   • PK = USER#<user_id>
   • SK = GROUP#<group_id>
   • Attributes: group_name, created_at, etc.
   Now Query PK=USER#u1 returns all groups for that user.

3. Add a GSI to invert the lookup (groups → users)
   Project the association items into a GSI so I can list a group's users without duplicating items:
   • On each association item set: GSI1PK = GROUP#<group_id>, GSI1SK = USER#<user_id>
   • Create GSI1 on (GSI1PK, GSI1SK)
   Now Query GSI1 PK=GROUP#g1 returns all users in that group.

4. Optional: store denormalized summaries
   Keep the full user and group "profile" items separate (same table, different SK). Put only small, frequently needed attributes on the association items to avoid extra reads for common list screens. Fetch details with a BatchGet when needed.

5. Maintain referential integrity
   On add/remove membership, use TransactWriteItems to insert/delete the association item (and any counters) atomically. If you keep membership counts, use the sharded counter pattern to avoid a hot counter.

6. Paging and filtering
   Use begins_with(SK, 'GROUP#') with Limit + LastEvaluatedKey for pagination. If you filter by status/role, encode it in the SK (e.g., GROUP#<group_id>#role#admin) or add another sparse GSI.

Result: one table, one association item per link, one GSI for the reverse lookup. Queries are O(1) partitions and scale cleanly.

4. How do you prevent hot partitions in DynamoDB?

I design keys and write patterns so traffic spreads across many partition keys, then smooth any bursts with sharding, batching, and caching.

1. Choose a high-cardinality, well-distributed partition key
   Avoid "all writes to one PK." Good examples: tenant_id, device_id, or a hashed ID. Bad example: using a single date as PK for all events.

2. Add write sharding (bucketing) when one key is hot
   Append a small shard to the PK to fan out writes, then read across shards:
   • PK = <logical_key>#b<00..15> (bucket = hash(item_id) % 16)
   • Reads: query 16 buckets in parallel and merge.
   Tune shard count so per-partition write rate stays below limits.

3. Time-series bucketing
   Don't dump an entire day into one PK. Use short time buckets in the PK or SK prefix, e.g., device#b03 as PK and 20250823T10… as SK, or include yyyyMMddHH in the PK to spread load over time.

4. Even out bursts
   Use BatchWriteItem to coalesce many small writes, and add small client-side jitter/backoff to avoid synchronized spikes. If ingestion is spiky, buffer via SQS/Kinesis and drain steadily.

5. Split noisy tenants or workloads
   If one tenant dominates traffic, give them their own table (or a separate logical PK namespace) so they don't throttle others. This is the cleanest way to contain a hotspot.

6. Duplicate hot read items and cache
   For read hotspots, use DAX or an app cache (Redis/CloudFront for public configs). If needed, store read-only "hot" copies under multiple shard keys and round-robin reads; update all copies on write (rare, but useful for extreme skew).

7. Design GSIs with the same care
   GSIs can be hot too. Use the same sharding/bucketing on the GSI keys if that access pattern is skewed.

8. Keep items small and writes efficient
   Move large blobs to S3 and store a pointer. Smaller items use fewer WCUs and reduce write pressure per partition.

9. Capacity choices
   Start with on-demand to absorb unknown load. For steady traffic, use provisioned with auto scaling and watch throttles. Adaptive capacity helps for reads but can't save a single hot PK that exceeds hard per-partition write limits so sharding is still key.

Result: with a well-distributed PK, shard/bucket patterns, and smoothing of bursts, traffic spreads across many partitions and you avoid throttling from hot keys.

5. How would you implement strong consistency in DynamoDB?

I decide where I truly need "read-after-write" guarantees and design for it using strongly consistent reads, conditional writes, and (when needed) transactions while avoiding patterns that can't be strongly consistent (like GSIs).

1. Use strongly consistent reads where needed
   For point reads or Queries that must see the latest write, I set ConsistentRead=true. This gives read-after-write consistency in the same region. I only use it on hot paths that need it because it costs more RCUs and can increase latency.

2. Prefer keys that make consistent reads possible
   Strongly consistent reads work on the base table and on Local Secondary Indexes (LSIs), not on Global Secondary Indexes (GSIs). If I need a consistent alternate sort key, I use an LSI; if I must query via a GSI, I accept eventual consistency or redesign (denormalize the attribute into the base item so a consistent GetItem hits it).

3. Guard writes with optimistic concurrency
   I put a version (or updated_at) attribute on items and use ConditionExpression (for example, version = :expected). This prevents lost updates when multiple writers race. If the condition fails, the app retries with the latest item.

4. Use transactions when multiple items must change atomically
   If I must update several related items as one unit, I use TransactWriteItems / TransactGetItems. That gives all-or-nothing semantics and consistent reads of the transaction's results to subsequent requests in-region.

5. Keep global tables expectations realistic
   Strong consistency is per region only. With Global Tables, cross-region reads are eventually consistent by design. For multi-region "strong-ish" behavior, I route a tenant's reads/writes to the same home region and use failover only on disaster.

6. Ensure consumers see the latest write quickly
   For read-after-write on the same item, a consistent GetItem right after a successful Put/Update is sufficient. If I fan out updates to derived items (like projections), I don't rely on Streams+Lambda for "strong" timing that's asynchronous. Instead, I either read the base item consistently or write all required fields into the same item.

7. Keep item design simple and self-contained
   The more an access pattern can be satisfied with a single consistent GetItem or Query on one partition, the easier it is to guarantee strong reads without extra moving parts.

Result: strong reads on the base table/LSI + conditional writes (and transactions where truly needed) give me predictable, in-region strong consistency for critical paths, while I keep GSIs and cross-region reads as eventual.

6. How can you reduce DynamoDB costs while handling large volumes of data?

I cut RCUs/WCUs, storage, and "waste" by shaping access patterns, minimizing payloads, using the right capacity mode, and caching.

1. Read cheaper
   • Prefer Query/GetItem over Scan (Scan reads every item and burns RCUs even if filtered out).
   • Use ProjectionExpression to fetch only needed attributes smaller responses use fewer RCUs.
   • Use eventually consistent reads where acceptable (they consume fewer RCUs than strong reads).
   • Add DAX or an app cache (Redis) for hot reads; a cache hit avoids RCUs entirely.

2. Write cheaper
   • Keep items small: move large blobs to S3 and store an s3_uri. Shorter attribute names and compact JSON reduce WCU and storage.
   • BatchWriteItem/BatchGetItem to amortize overhead for high-throughput bulk ops.
   • Avoid unnecessary GSIs on write-heavy items every GSI write is an extra write you pay for.

3. Index only what you need
   • Create GSIs sparingly and make them sparse (only project items that need to appear).
   • Use keys-only or minimal projected attributes on GSIs to reduce write/storage cost.
   • Revisit unused GSIs with CloudWatch/CloudTrail metrics and drop them.

4. Capacity mode and scaling
   • Start with on-demand if traffic is spiky/unpredictable no idle over-provisioning.
   • For steady high throughput, switch to provisioned with auto scaling; set a realistic target utilization (for example, 70%) and floor/ceiling to avoid paying for peaks all day.
   • Shard hot keys (write bucketing) so you don't have to over-provision for a single hot partition.

5. Storage optimization
   • Enable TTL on ephemeral items to auto-delete and free storage.
   • Consider Standard-IA table class for large, rarely accessed tables (cheaper GB storage; verify access patterns).
   • Export old/cold data to S3 (on-demand export, Streams → Firehose) and delete from DynamoDB; query cold data with Athena when needed.

6. Data model that avoids waste
   • Model access patterns so a single key-based Query satisfies each use case no full scans or large fan-outs.
   • Precompute small rollups you read often instead of scanning large raw sets repeatedly.

7. Operations and guardrails
   • Monitor consumed vs provisioned capacity, throttles, and item sizes; alert on scans creeping in.
   • Use reserved capacity (if predictable long-term usage) to lower $/capacity.
   • Avoid heavy transactions on hot paths they cost more and can reduce throughput; reserve them for true multi-item atomicity.

Result: by reading only what I need, writing the minimum, indexing sparingly, choosing the right capacity mode, caching hot paths, and aging out cold data, I handle large volumes at a much lower cost without sacrificing performance.

7. How would you handle highly concurrent writes to DynamoDB without conflicts?

I use optimistic concurrency with conditional writes, make updates idempotent, and design items so most operations touch a single partition key. This avoids overwrites and preserves correctness under heavy parallelism.

1. Use conditional writes as the first line of defense
   I keep a version (or updated_at) on each item and require it in every write:

- PutItem: ConditionExpression attribute_not_exists(pk) for create-once semantics
- UpdateItem: ConditionExpression version = :expected (or updated_at = :expected)
  If another writer changes the item, my write fails with ConditionalCheckFailed; I read the latest item, merge, and retry. This prevents lost updates.

2. Make updates idempotent with a request_id
   I store a last_request_id on the item and write with a condition like attribute_not_exists(last_request_id) OR last_request_id <> :rid. Retries with the same :rid do nothing, so clients can safely retry on timeouts.

3. Use atomic Update expressions instead of read-modify-write
   I avoid a separate read; I apply changes in a single UpdateItem:

- Counters: SET counter = if_not_exists(counter, :zero) + :inc
- Append-only sets/lists: ADD tag_set :new_tags
- "Only if greater/newer": SET max_value = if_not_exists(max_value,:neg_inf) then ConditionExpression max_value < :candidate
  These are server-side atomic and avoid race conditions.

4. For very hot counters, use sharded counters
   Maintain N counter items (pk=counter#b00..bNN), increment one at random, and sum them on read. This spreads writes across partitions and prevents throttling/conflicts on a single item.

5. Group related fields into one item when possible
   If an operation needs to update A and B together, put them in the same item so a single conditional UpdateItem covers both. Only when you must span items use a transaction.

6. Use transactions for multi-item invariants
   When two or more items must change atomically (for example, move credits from one account to another), I use TransactWriteItems with ConditionChecks (balances, versions). Keep transactional hot paths minimal they cost more and have tighter limits.

7. Backoff and retry policy
   On ConditionalCheckFailed, I merge and retry quickly; on throttling, I use exponential backoff with jitter. Because writes are idempotent, retries are safe.

8. Reduce contention by sharding keys
   If many writers hit the same PK/SK, add a small shard/bucket in the sort key (or PK) and reconcile via a reader/aggregator pattern. This prevents a single partition from becoming a bottleneck.

Result: conditional + idempotent updates handle conflicts cleanly, atomic Update expressions avoid races, and sharding counters/keys removes write hotspots under extreme concurrency.

8. How do you implement cross-region replication for DynamoDB?

I use DynamoDB Global Tables (multi-active) when I want built-in, near-real-time replication; otherwise, I build selective replication with Streams + Lambda. I design for eventual consistency across regions and last-writer-wins.

1. Prefer Global Tables (recommended)

- Create the table (version 2019.11.21) in Region A, then add Regions B, C via "Global tables". Key schema, TTL, and GSIs must be identical everywhere.
- DynamoDB replicates writes asynchronously; conflict resolution is last-writer-wins based on a per-item timestamp maintained by the service. Don't rely on millisecond ordering for cross-region conflicts design to avoid concurrent updates to the same item from different regions when possible (e.g., route a tenant to a "home region").
- Reads are local and fast in each region; strong consistency is only within a region (cross-region reads are eventual).

2. Security and keys

- Encrypt each replica with KMS. Use multi-Region KMS keys (MRKs) or ensure each region's CMK policy allows DynamoDB to use it.
- Grant your app roles in each region the minimal IAM permissions for that table and its key.

3. Capacity, costs, and limits

- Each region's write costs apply independently (a write replicates to all regions). Choose on-demand vs provisioned per region.
- Monitor replication lag and throttles per region; scale before backlogs grow.

4. Failover and routing

- Use Route 53/ALB health checks to route clients to the nearest healthy region. On regional failure, shift traffic to another region.
- After failback, your app should continue with the same item versions; if conflicting edits happened during split-brain, last-writer-wins applies store an application-level updated_at and updated_by to reconcile business-wise if needed.

5. Streams consumers in multi-region setups

- If you have Streams → Lambda consumers, deploy them in every region (one per replica). Use idempotent handlers so an event processed in multiple regions doesn't double-apply changes downstream.

6. Selective replication (when you don't need full multi-active)

- Use DynamoDB Streams in Region A, process with Lambda, and write a subset of items to Region B's table (or to S3/Kinesis).
- Add a dedup key/request_id to make the target writes idempotent.
- This is useful for "read-mostly" DR or for exporting to analytics regions without paying for full global tables.

7. Data model considerations

- Keep items self-contained so reads don't need cross-region joins.
- Avoid cross-region transactions; they aren't supported. If two regions might update the same logical record, partition writers by tenant or record to a single "owner region".

8. Operations

- Enable PITR in each region; backups/restores are per-region.
- Test region removal/addition in a staging account.
- Watch CloudWatch metrics for replication latency, subscriber errors (if you have Streams consumers), and WCU/RCU trends per region.

Result: with Global Tables you get simple, managed multi-active replication and local low-latency reads; where that's overkill, Streams-based selective replication covers targeted needs. In both cases, design for eventual consistency and avoid cross-region write conflicts.

9. How can you perform efficient searches in DynamoDB?

I design keys and secondary indexes so every search is a key-based Query (not a Scan). When I truly need text search or ad-hoc filters, I offload to OpenSearch while keeping DynamoDB as the system of record.

1. Start with access patterns, then choose keys
   I map each query I must support to a partition key and sort key pattern. Example: list all orders for a customer in date order → PK = customer_id, SK = order#yyyymmddhhmmss. Now Query can fetch a precise range with SK BETWEEN or begins_with.

2. Use composite sort keys to support multiple filters
   I encode extra attributes into SK to filter without scanning. Examples:
   • SK = status#yyyymmddhhmmss → Query by PK + begins_with(status#open)
   • SK = type#region#timestamp → prefix queries for type or type+region

3. Add secondary indexes only for necessary alternate lookups
   • GSI for alternate partition key (cross-tenant/admin searches, "find by email", etc.). Keep GSIs sparse by writing only relevant items to them and project minimal attributes to save cost.
   • LSI if I need a second sort key for the same PK with strong consistency.

4. Precompute "searchable" views
   For common filters like status or category, write small index items alongside the main item (same table) with a different SK prefix (for example, idx#status#open#timestamp). Querying those prefixes is fast and avoids a Scan.

5. Avoid Scan; if you must, constrain it
   If a one-off Scan is unavoidable, restrict it by FilterExpression + ProjectionExpression and run it in parallel with careful page size. But in production paths, redesign so it's a Query.

6. Full-text / fuzzy search
   DynamoDB doesn't do full-text well. Use Streams → Lambda → Amazon OpenSearch (or CloudSearch) to index titles/descriptions and run text queries there. Store only the doc ID and fetch the item from DynamoDB by key after the search.

7. Secondary tools for analytics
   For wide aggregations (top-N, GROUP BY), stream to S3 (Firehose/Export to S3) and query with Athena/Glue or keep aggregates precomputed in DynamoDB (rollup items) for real-time dashboards.

Result: every "search" becomes a cheap, predictable Query on a base key or index; true text search is handled by OpenSearch with DynamoDB as the source of truth.

10. How do you automate backups and disaster recovery for DynamoDB?

I combine continuous backups (PITR), scheduled backups, exports to S3, and if I need multi-region RTO/RPO Global Tables. I also script restores and test them regularly.

1. Turn on Point-In-Time Recovery (PITR)
   This keeps a rolling 35-day history. I can restore the table to any second in that window (new table name) to fix accidental deletes or corruption.

2. Schedule on-demand backups
   Use AWS Backup or a simple schedule (EventBridge → Lambda) to take daily/weekly full backups, retained per compliance policy. Store them in a Backup vault (optionally cross-account) with lifecycle rules.

3. Export to S3 for analytics/archival
   Use "Export to S3" (no table impact) or Streams → Firehose to land data (Parquet/JSON) in S3. This gives point-in-time snapshots you can query with Athena and keeps long-term archives cheap.

4. Plan for regional disasters
   • Easiest: Global Tables (multi-active). Writes replicate asynchronously to other regions; on failover, route traffic to a healthy region. Design for eventual consistency and avoid cross-region write conflicts.
   • Cost-saving alternative: regular cross-region backups (AWS Backup copy to another region) plus a tested restore Runbook. RTO is higher than Global Tables but cheaper.

5. Encryption and access control
   Encrypt tables with KMS (consider multi-Region keys for Global Tables). Ensure backup vault policies and KMS keys allow restores in DR accounts/regions.

6. Automate restore Runbooks
   Script: pick a recovery point (PITR timestamp or backup ARN) → restore into temp table → verify item count/index status → swap application config/aliases to the restored table → re-enable Streams/triggers. Keep infrastructure as code so cutover is predictable.

7. Include Streams/GSIs/TTL in the plan
   Backups include GSIs/TTL settings; after restore, reattach Lambda consumers and downstream pipelines. If you use Streams for CDC, decide from which sequence number to resume.

8. Monitor and test
   Create alarms on Backup/Restore jobs, PITR status, and Streams lag. Run quarterly DR drills: restore to a staging account/region, run validation queries, and measure RTO/RPO.

Result: continuous short-term recovery with PITR, automated long-term backups and exports, and when needed multi-region availability via Global Tables, all documented and tested so recovery is fast and reliable.

11. How would you design keys in DynamoDB to store IoT readings so you can quickly get the latest reading and also query by time?

Use one partition per device and a time-ordered sort key.

- Partition key = device_id
- Sort key = ts_epoch_ms (number) or an ISO-8601 string that sorts lexicographically (e.g., 2025-08-23T10:15:30Z)

With this:

- Get latest reading: Query with PK=device_id, set ScanIndexForward=false, Limit=1.
- Query by time window: KeyConditionExpression: device_id = :d AND ts BETWEEN :t1 AND :t2.

Helpful add-ons:

- "Latest pointer" item you overwrite each write: PK=device_id, SK=LATEST → O(1) consistent GetItem for latest.
- If a single device is extremely hot, shard writes: PK=device_id#b{00..15}, SK=ts. Pick bucket by hash(event_id)%N. For latest, either keep the LATEST pointer (unsharded) or query N buckets (limit 1, descending) and pick the max ts in code.
- Use TTL to age off old readings and export historical data to S3; keep payloads small and store blobs in S3 with a pointer.

12. In a SaaS app with many customers, how would you design DynamoDB keys to keep each customer's data separate and avoid hot partitions?

Scope every item by tenant and shape the sort key to your access patterns; shard only for very hot tenants.

- Partition key = tenant_id (or TENANT#<id>)
- Sort key = entity_type#entity_id or entity_type#yyyymmdd#id (so you can list by type and/or time)

This gives:

- "All orders for a tenant" → Query PK=tenant_id, SK begins_with order#
- "Orders by date" → SK BETWEEN order#20250801 AND order#20250831
- "Get one record" → PK+SK exact match

To avoid hot partitions:

- Most tenants naturally distribute load (many PKs). If one tenant is noisy, per-tenant sharding: PK=tenant_id#b{00..15}, SK=... (bucket by hash). Reads fan out over the small bucket set and merge results. For extreme outliers, move that tenant to its own table.

Security and alternate lookups:

- Enforce isolation with IAM conditions like dynamodb:LeadingKeys = ["${tenant_id}"].
- Add sparse GSIs only as needed (e.g., GSI1PK=tenant_id, GSI1SK=status#updated_at for dashboards; an admin-only cross-tenant GSI if required).

Practical tips:

- Keep items <400 KB; big artifacts go to S3.
- Precompute small rollups (counts, "latest" pointers) as separate items under the tenant's partition.
- Start with on-demand; switch steady/hot workloads to provisioned + auto scaling when patterns are clear.

13. In a social media app, how would you design keys so a user can fetch all their posts in order?

Use one partition per user and a time-ordered sort key.

- Partition key = user_id
- Sort key = post_ts_epoch_ms#post_id (or an ISO-8601 timestamp string followed by post_id to ensure uniqueness)

With this design:

- Fetch all posts newest-first: Query with PK=user_id, set ScanIndexForward=false, and use Limit for paging; keep the LastEvaluatedKey as your cursor.
- Fetch posts within a time window: KeyConditionExpression with SK BETWEEN start_ts AND end_ts.
- Avoid large payloads per item; store media in S3 and keep s3_uri in the item.
- For extremely prolific users, consider light sharding: PK=user_id#b{00..03}, SK=post_ts#post_id; read N buckets in parallel and merge. Keep a tiny pointer item per user (PK=user_id, SK=LATEST_POST) for quick "latest" lookups.
- If you need alternate per-user ordering (for example, by number_of_likes), add an LSI with the same PK=user_id and LSI sort key = likes_desc#post_ts so you can query "top liked posts" strongly consistently.

Result: a single Query on the user's partition returns posts in the desired order without scans, with clean pagination and optional alternatives via an LSI if you need different orderings.

14. For product reviews, how would you support queries by product_id and also by rating using LSI or GSI?

Model the base access by product_id, then add an LSI for "by rating within a product" and a GSI only if you also need "by rating across products."

- Base table (by product):
    - PK = product_id
    - SK = review_ts#review_id
      This supports "all reviews for a product" and time-window queries.
- Per-product sort by rating (use LSI for strong consistency):
    - Define LSI1 with the same PK=product_id and LSI1-SK = rating#review_ts.
    - Query product_id + begins_with(rating#5) for 5-star reviews, or use a range on rating if you encode rating as a zero-padded number.
    - Because LSI shares the base PK, you can do strongly consistent reads when needed.
- Cross-product queries by rating (use GSI if required):
    - GSI1 PK = rating#category (or rating#tenant) to avoid hot partitions from only 5 rating values; GSI1 SK = product_id#review_ts.

    - Query GSI1 for rating#5 and optionally filter by category/tenant to list recent 5-star reviews across a segment. This is eventually consistent and incurs extra write cost add only if you truly need it.
- Projections and cost:
    - Project only the attributes needed for list views (title, stars, snippet, review_ts) into the LSI/GSI to keep write and storage costs down; fetch full text with a follow-up GetItem by product_id + review_ts#review_id when the user opens a review.
- Hot product protection:
    - If one product is exceptionally hot, add light sharding in SK (e.g., review_ts#b02#review_id) or throttle writes via your API layer to avoid bursts; the LSI index key should mirror the same suffix.

Result: the base table handles by-product queries; the LSI gives fast, per-product rating ordering with optional strong consistency; and a carefully keyed GSI supports cross-product rating views without creating hot partitions.

15. A user table must support queries by email and also by signup date range. Would you use LSI, GSI, or both?

Use a GSI for email, and use LSI or another GSI for signup date depending on the scope of the date-range query. In many real apps you'll use both.

- Email lookup (unique or near-unique): use a GSI. Set GSI1 PK = email, GSI1 SK = user_id (or created_at). This gives O(1) "get by email" without scanning. LSI can't help because it shares the base table's partition key, and you don't partition users by email.
- Signup date range:
    - If you query "users by signup date within a tenant/account," model the base table as PK = tenant_id, SK = created_at#user_id. Then add an LSI only if you need a different per-tenant sort key. For simple date-range per tenant you can just query the base table by PK=tenant_id and SK BETWEEN start/end (no index needed).
    - If you need "users by signup date across all tenants/customers," add a second GSI: GSI2 PK = created_date_bucket (for example, yyyymm or yyyymmdd), GSI2 SK = created_at#tenant_id#user_id. This supports global time windows without scans.
- Summary: GSI for email; for signup date use base table (per-tenant) or a time-bucketed GSI (cross-tenant). That's why "both" is common.

16. If queries need to filter by a non-key attribute (like region) across all partitions, would you use LSI or GSI? Why?

Use a GSI. LSI shares the base table's partition key, so it only helps you re-sort items within one partition. When you need to query across all partitions by some attribute, you must promote that attribute into an index key.

- Create a GSI with PK = region (and an SK that suits your sort, for example signup_at#user_id). Then Query GSI PK='APAC' returns only APAC items without scanning the whole table.
- To avoid hot partitions on popular regions, bucket the GSI key: PK = region#yyyymm (or region#hash_suffix) and query the few relevant buckets in parallel.
- Keep the GSI sparse and cheap by projecting only the attributes needed for the list view; fetch full details with a GetItem on the base table by primary key afterward.

LSIs can't solve cross-partition filtering; GSIs are built exactly for that.

17. For a ticket booking app with heavy flash-sale spikes, would you use On-Demand or Provisioned with Auto Scaling, and why?

I would default to On-Demand for launch and flash-sale events because traffic is bursty and hard to predict. On-Demand instantly absorbs big spikes without pre-planning capacity, so we avoid throttling during the first critical minutes of a sale. After we learn traffic patterns, we can evaluate switching some tables to Provisioned with Auto Scaling plus scheduled scaling (pre-scale minutes before a known sale window) to reduce cost if the pattern is predictable.

How I'd run it in practice

- Start On-Demand for orders, inventory locks, and sessions (most spiky).
- Keep less spiky/analytics tables on Provisioned + Auto Scaling.
- Add SQS/Kinesis in front of writes to smooth bursts, implement idempotent writes and retries with jitter.
- Use hot-key resistant keys (shard/bucket popular items) so a single partition isn't the bottleneck.
- After a few events, compare On-Demand spend vs Provisioned+Scheduled scaling; move only if savings are clear and risk of under-provision is low.

18. If IoT devices write 2KB items at 500 writes/sec, how many WCUs do you need? How would you handle sudden growth?

A write of up to 1 KB = 1 WCU. A 2 KB item needs 2 WCUs (rounded up).
Required WCUs = 500 writes/sec × 2 WCUs per write = 1,000 WCUs for the base table.

Important considerations

- Indexes: each GSI write also consumes WCUs. With one GSI, total ≈ 2,000 WCUs (1,000 for table + 1,000 for GSI). Multiply by the number of GSIs you project the item into.
- Sudden growth handling:
    - Use On-Demand to absorb spikes, or raise Provisioned capacity ceilings and enable Auto Scaling with realistic target utilization (e.g., 70%).
    - Shard hot keys (e.g., device_id#b00..15) so partition-level limits aren't hit.
    - Buffer with Kinesis/SQS and drain steadily; make writes idempotent so retries are safe.
    - Keep items small (move blobs to S3) to keep WCUs down.

19. Your API is throttling in provisioned mode. What would you check or change to fix it?

I'd verify if throttles are from overall capacity, partition hot spots, or indexes and fix the root cause.

What I check first

- CloudWatch metrics: Consumed vs Provisioned (table and each GSI), ThrottledRequests, and SuccessfulRequestLatency.
- Key design: a hot partition (few partition keys or a single popular key) will throttle even if table capacity looks high.
- Auto Scaling limits: are min/max and target utilization set too low? Is scaling too slow (cooldowns)?
- Request patterns: Scans instead of Queries, strong reads where eventual would do, large item sizes, transactional writes (2× capacity cost), or big Batch/Transact spikes.

Fixes

- If overall capacity is low: raise provisioned RCU/WCU or widen auto-scaling max; consider On-Demand during peaks.
- If hot partitions: add write sharding/bucketing to the PK (e.g., key#b00..15), spread traffic; for reads, use DAX/app cache and denormalized lookup items.
- Reduce per-request cost: switch to Query by key, add ProjectionExpression to fetch only needed attributes, prefer eventual consistency for reads, shrink item size (move blobs to S3).
- Index throttles: remember GSIs have their own capacity raise GSI capacity or make the GSI sparse/minimally projected.
- Client behavior: implement exponential backoff with jitter; batch writes/gets where appropriate; ensure idempotency so safe retries don't double-write.

- If spikes are predictable: schedule pre-scaling before the surge; after it, scale back down.

20. For a product catalog read thousands of times but rarely updated, how would DAX help? What's the trade-off?

DAX gives you an in-memory cache in front of DynamoDB, so catalog reads come from RAM with micro- to single-digit millisecond latency and far fewer RCUs. It shines on read-heavy, mostly-static data like product cards, pricing, and metadata.

How I'd use it

- Put a small DAX cluster in the same VPC/subnets as the app.
- Swap the SDK client to the DAX client (endpoint change); no schema changes.
- Cache keys: GetItem, BatchGetItem, and common Query patterns (e.g., by category).
- Set TTLs per use case (e.g., 5–30 minutes). For the few updates, use write-through so DAX invalidates/refreshes after writes.

Trade-offs

- Eventual consistency only for cached reads; if a read must be strongly consistent, bypass DAX and hit DynamoDB directly.
- You pay for the DAX cluster (but usually save far more on RCUs).
- Staleness up to TTL and cache-warm time on cold start.
- Doesn't help writes or unsupported ops (e.g., scans don't benefit much).

Result: much faster, cheaper reads for the catalog, with small risk of slightly stale data and the cost of a modest DAX cluster.

21. In checkout, customer lookups in DynamoDB are slow. How would DAX help, and would app code change?

DAX can cut lookup latency (profile, address, payment prefs) and offload RCUs. Code changes are minimal: use the DAX SDK client and point it at the DAX endpoint; keep a plain DynamoDB client for the few places that require strong consistency.

Approach

- Add DAX, switch GetItem/Query calls to the DAX client.
- Keep TTL short for checkout (e.g., 1–5 minutes) or use write-through on updates to keep the cache fresh.
- For "must see my latest write" paths (e.g., user just changed address), bypass DAX and call DynamoDB directly for that read.
- Secure/network: same VPC, security groups, IAM role for DAX.

Code impact

- Instantiate a DAX client (and optionally keep the standard DynamoDB client).
- Replace client usage; business logic, keys, and schemas are unchanged.

Net effect: faster, cheaper reads on checkout pages with minimal code drift; strong-consistency reads still go straight to DynamoDB.

22. A leaderboard dashboard makes frequent reads and costs are high. How would DAX help, and what are its limits?

DAX can cache the hot leaderboard queries (top-N lists, player cards), slashing RCUs and latency for dashboards/polls.

How I'd set it up

- Model the leaderboard as a precomputed table (no scans).
- Use DAX to cache GetItem for player rows and Query for "top-N" partitions (e.g., season#division with SK sorted by score desc via inverted score).
- Very frequent refresh: set a small TTL (e.g., 5–30 seconds) or refresh via write-through when scores change.

Limits to be aware of

- Eventual consistency: dashboards may be seconds behind truth. If you need "exact now," bypass DAX for that read.
- High update rates churn the cache; benefit drops if scores change many times per second.
- DAX doesn't speed writes or reduce WCUs, and cached scans don't help design for key-based queries.
- You run (and pay for) a DAX cluster; place it close to the app.

Bottom line: DAX is great for caching hot, key-based leaderboard reads and cutting cost/latency, as long as slight staleness is acceptable and the data model avoids scans.

23. In banking, balances must be 100% correct. Would you use strong or eventual consistency? Why?

I would use strong consistency for reads that drive a balance-sensitive decision (e.g., "can I withdraw now?") and protect writes with conditional expressions (and transactions where needed). Strongly consistent reads give read-after-write guarantees in the same region, so the app won't show a stale balance after a successful update. I also add optimistic concurrency (a version or updated_at field) so two writers can't overwrite each other. For multi-item changes (e.g., transfer between accounts), I use TransactWriteItems so all updates commit together or none do. Eventual consistency is fine for non-critical views (historical lists, statements), but not for the balance check path.

Key pieces I'd put in place

- Strongly consistent GetItem/Query on the base table (not GSIs) for "what's the balance right now?".
- Conditional writes: ConditionExpression version = :expected to prevent lost updates.
- For debits: SET balance = balance - :amt with ConditionExpression balance >= :amt AND version = :v.
- Transactions for multi-account transfers, plus idempotency keys so retries don't double-apply.
- One "home region" per account if multi-region, because cross-region is eventual by design.

24. Two sellers update stock at the same time. How would you prevent overwrites in DynamoDB?

Use optimistic concurrency and atomic, idempotent updates.

Practical pattern

- Keep a version (or updated_at) attribute on the stock item.
- Update with a single atomic UpdateItem and a condition:
    - SET qty = qty + :delta, version = :v_next
    - ConditionExpression version = :v_current AND qty + :delta >= 0
      This avoids read-modify-write races and prevents going below zero.
- Make the request idempotent: include request_id on the item and add attribute_not_exists(last_request_id) OR last_request_id <> :rid
  so retried requests don't apply twice.
- If stock is extremely hot, shard the counter (e.g., stock#b00..b15) and sum shards on read; this removes a single-item hot spot.
- If multiple related items must change together (e.g., bundle SKUs), wrap them in a TransactWriteItems with ConditionChecks.

Result: simultaneous writes either succeed one at a time or fail with ConditionalCheckFailed, and the app retries using the latest version.

25. In a booking app, flight + hotel + payment must succeed or fail together. How would you use DynamoDB transactions?

Use a short, idempotent saga with DynamoDB transactions for your state changes, and pre-authorize payment so the commit is atomic from the user's perspective.

Flow I'd implement

1. Place holds with conditional writes
    - Write "hold" items for flight and hotel with TTL (expires if not confirmed), each using ConditionExpression to ensure availability > 0.
2. Pre-authorize payment (external gateway)
    - Get an authorization token; no capture yet. Store an auth_id on a booking record (not confirmed).
3. Commit with a single TransactWriteItems
    - Decrement confirmed inventory for flight and hotel.
    - Mark the booking as CONFIRMED, persisting the auth_id and an idempotency_key.
    - Delete the temporary holds (or flip their status).
      If any condition fails (no seats/rooms), the whole transaction aborts.
4. Capture payment only after a successful transaction
    - If the transaction commits, call the gateway to capture using auth_id.
    - If capture fails, run a compensating TransactWriteItems to revert the confirmation (increment inventory back, mark booking FAILED).
5. Idempotency and retries
    - Every step carries an idempotency_key; writes check it to avoid duplicates on retry.
6. Limits and design notes
    - DynamoDB transactions: up to 25 items, 4 MB total; keep records small.
    - Keep all keys designed for single-partition queries (e.g., PK=booking_id).
    - Use a state machine (Step Functions) to orchestrate: hold → auth → transact-confirm → capture → finalize/compensate.
    - If you need cross-tenant/regional isolation, keep each booking's writes in one region; strong consistency is per-region.

This gives true all-or-nothing semantics for your DynamoDB state, with safe payment handling and clean rollbacks if any step fails.

26. One IoT device sends huge data, causing a hot partition. How would you redesign the table to spread the load?

Use write sharding so that one device's traffic fans out across many physical partitions, while keeping reads practical.

- Add buckets to the partition key
  Partition key = device_id#b{00..15} (or {00..63}); Sort key = ts_epoch_ms (or ts#event_id). Choose bucket = hash(event_id) % N or round-robin. This turns one hot PK into N keys that DynamoDB can place on different partitions.

- Reading it back
  • Latest reading: either keep a tiny pointer item (PK=device_id, SK=LATEST) you overwrite on each write, or query each bucket with ScanIndexForward=false, Limit=1 and pick the max timestamp in the app.
  • Time windows: query the N buckets in parallel for ts BETWEEN :t1 AND :t2 and merge results (fan-in at app side or Lambda).

- Keep buckets sensible
  Start with 16 or 32; increase only if throttle persists. Too many buckets make reads expensive.

- Alternative/extra measures
  • Move oversized payloads to S3; store s3_uri to shrink WCUs.
  • Buffer with Kinesis/SQS to smooth burst writes.
  • If this one device dominates the fleet, put it in its own table so it can scale independently.

Result: the device's writes distribute across many keys/partitions, eliminating a single hot partition while preserving efficient queries.

27. In a gaming leaderboard, top players generate too many writes on one key. How would you shard the writes?

Shard the hot key and precompute aggregates so no single PK carries all updates.

- Shard the player (or board) key
  Option A (player-centric): PK=player_id#b{00..15}, SK=season#division#inverted_score#event_id. Choose bucket by hash of event/match id.
  Option B (board-centric): PK=season#division#b{00..15}, SK=inverted_score#player_id. Writers pick a bucket; readers query all buckets and merge top-N.
- Maintain a fast "current score" item
  Keep a canonical per-player item (PK=player_id, SK=season#division#CURRENT) updated with an atomic SET score = if_not_exists(score,0) + :delta. If this becomes hot, use sharded counters (N items like CURRENT#b00..b15) and sum on read.
- Read paths
  • Player profile: single consistent GetItem on CURRENT (or sum shards).
  • Leaderboard page: query each bucket's top K (small Limit) and merge/heap select the global top-N in the app/Lambda cheap because each bucket returns only K rows.
- Idempotency and contention
  Use request_id with conditional updates to avoid double-counting on retries. If two updates race, conditional expressions (or transactions for multi-item updates) prevent lost updates.

Result: updates spread across buckets, "current score" stays atomic and fast, and top-N reads remain efficient by merging tiny per-bucket slices.

28. How does DynamoDB adaptive capacity help with uneven traffic, and what are its limits?

Adaptive capacity automatically shifts a table's provisioned capacity to hotter partitions so they get more throughput than a naive even split reducing throttles without you changing keys. It's great for modest skews, but it can't break hard limits.

What it does well

- Rebalances RCUs/WCUs across partitions based on recent traffic so a warm partition can "borrow" from cooler ones.
- Works for base tables and GSIs (each index has its own adaptive behavior).
- Helps with uneven traffic when you have plenty of headroom at the table level.

Key limits to know

- It cannot save a single hot partition key that exceeds physical per-partition limits if all writes hammer one PK, you must shard the key.
- It isn't instantaneous; it reacts over short time windows. Sudden spikes can still throttle before the system adapts.
- It only reallocates within your provisioned capacity (or underlying limits). If the whole table is under-provisioned, you'll still throttle raise capacity or use On-Demand.
- GSIs need enough capacity too; a hot index key can throttle even if the base table is fine.
- It doesn't reduce request cost; bad access patterns (Scans, huge items, strong reads everywhere) still burn capacity.

Practical use

- Keep good key distribution and add write sharding for known hot keys; rely on adaptive capacity as a safety net for normal skews.
- Set realistic auto-scaling ceilings so adaptive capacity has headroom to borrow from.
- Monitor throttles at table and GSI levels; if a single PK keeps throttling, shard or isolate that workload.

Bottom line: adaptive capacity smooths typical unevenness, but hot-key problems still require data-model fixes like sharding/bucketing.

29. Logs have small metadata but very large text (200KB). How would you reduce storage cost while still querying?

Keep only compact metadata in DynamoDB and offload the big text to cheaper storage. Index for search elsewhere.

- Store in DynamoDB only: ids, timestamps, app/service, level, small summary, and an s3_uri to the full log body. This shrinks item size so writes/reads cost fewer WCUs/RCUs and table storage drops a lot.
- Put the full 200KB text in S3 with compression (GZIP/Zstandard). Use lifecycle to transition old logs to cheaper tiers (Intelligent-Tiering → Glacier).
- For search, don't Scan DynamoDB. Stream new items to OpenSearch (full-text, phrase, wildcard). Search there to get matching ids, then fetch metadata or the S3 body as needed.
- For simple filters (time/app/level), design keys so Query works: PK = app#date, SK = ts#log_id, and add GSIs for level or error_code. This avoids scans and still keeps DDB tiny.
- Optionally store a short snippet/first 1–2 KB in DynamoDB for quick previews; click-through loads the full body from S3.

Result: DynamoDB is a fast, cheap index; S3 holds the large, compressed text; OpenSearch powers rich queries.

30. How would you auto-delete old session records in DynamoDB after logout?

Use DynamoDB TTL for automatic deletion, and mark sessions revoked immediately on logout.

- Add an attribute expires_at (Unix epoch seconds) to each session item and enable TTL on the table using that attribute. Set expires_at = now + session_ttl on create; on logout, update expires_at = now to expire soon.
- Remember TTL is best-effort (minutes to hours). To block access instantly, also set status = revoked and validate status on every request (don't rely on TTL alone for security).
- If you keep secondary data (for example, per-user session index), enable Streams and a small Lambda that reacts to TTL deletions to clean up any related items.
- For rolling inactivity timeouts, refresh expires_at on activity; write with a conditional expression so races don't shorten/extend incorrectly.

Result: sessions disappear automatically without manual jobs, while immediate revocation is enforced by checking status.

31. Analytics workloads spike unpredictably. How would you optimize capacity and cost?

Use On-Demand for spiky access, cache hot reads, and offload heavy analytics to S3/Athena instead of hammering DynamoDB.

- Capacity mode: switch the analytics-facing tables (or replicas) to On-Demand so you don't over-provision and you won't throttle on surprise spikes. For steady traffic, keep provisioned + auto scaling.
- Query shape: avoid Scan. Serve dashboards from key-based Queries or from precomputed aggregates you update via Streams/Lambda. This slashes RCUs.
- Caching: put DAX or an app cache in front of hot lookups/top-N pages. Cache TTLs of seconds/minutes can cut RCUs dramatically.
- Offload analytics: stream/export data to S3 (Streams → Firehose or Export to S3) and run ad-hoc/wide queries in Athena/EMR. Keep DynamoDB for serving workloads only.
- Index discipline: keep GSIs sparse and minimally projected; drop unused ones. Each GSI write/read costs extra.
- Cost guardrails: monitor Consumed vs Provisioned (or On-Demand spend), throttles, and item sizes; enable TTL on ephemeral items; consider Standard-IA table class for rarely accessed history.

Result: DynamoDB handles real-time, spiky access cheaply and reliably, while heavy analytics runs against S3 so you don't pay DynamoDB prices for scans.

32. How would you build a pipeline to move DynamoDB data into Redshift for reporting (using Glue/EMR/Kinesis)?

I'd keep DynamoDB as the system of record, land data in S3 as Parquet with change history, and load Redshift from there in an incremental, idempotent way.

Design (recommended path)

- Capture changes: enable DynamoDB Streams. Use Kinesis Data Streams or Firehose as the fan-out.
- Land to S3:
    - Easy mode: Firehose → S3 (buffer by size/time) → Parquet via record format conversion.
    - Or Glue Streaming/EMR Structured Streaming reads Streams and writes partitioned Parquet/Iceberg/Hudi on S3 (e.g., dt=YYYY-MM-DD/hour=HH, op=INSERT|UPDATE|DELETE). Include the table's PK and a change_ts.
- Curate & compact: a small Glue batch job periodically compacts tiny files, enforces schema, and (if using Iceberg/Hudi/Delta) upserts so S3 reflects latest state as well as change history.
- Load Redshift efficiently:
    - Stage tables in Redshift (raw_change_log, current_snapshot).
    - COPY from S3 (Parquet, Snappy/ZSTD) into staging using a manifest or partition predicates for that hour/day.
    - MERGE from staging into target (upsert by PK; apply deletes). Truncate staging after success.
- Initial backfill: use "Export to S3" (point-in-time export) or parallel Scan with segments to create a snapshot in Parquet; then start Streams for CDC from the export time onward.
- Idempotency & ordering: order by change_ts + sequence number; de-dupe by a change_id if present. Make MERGE idempotent so retries don't double-apply.
- Ops: monitor Firehose/Kinesis lag, S3 file counts/size, Glue DPU-hours, Redshift COPY/MERGE times. Partition S3 so COPY scans only recent partitions. Keep Secrets Manager for Redshift creds and SSE-KMS everywhere.

Alternatives

- DMS: DynamoDB → S3 (full + CDC) → Redshift COPY; quick to start, less flexible.
- EMR instead of Glue for heavy compaction or multi-table pipelines.

Result: low-latency CDC to S3, compact Parquet for cheap COPYs, and reliable MERGEs so Redshift stays fresh without hammering DynamoDB.

33. How would you use Global Tables to sync DynamoDB across regions, and what challenges might you face?

I'd use Global Tables (version 2019.11.21) for managed, multi-active replication, keep writes tenant- or record-scoped to a "home region," and design for eventual consistency across regions.

Steps

- Create the table in Region A; add Regions B/C as replicas (identical key schema, GSIs, TTL). Encrypt with KMS (multi-Region keys preferred).
- Route clients to their nearest region for low-latency reads; route each tenant's writes to a single "home region" to minimize cross-region conflicts.
- Use last-writer-wins awareness: store updated_at/updated_by in items so business logic can reconcile if needed.
- Re-deploy Streams/Lambda consumers per region (each replica emits its own stream). Make consumers idempotent.
- Capacity per region: pick On-Demand for bursty workloads; otherwise Provisioned + Auto Scaling. Monitor replication latency and throttles in every region.
- DR/failover: use Route 53 health checks to shift traffic; after recovery, reads remain local no special catch-up step needed.

Challenges

- Conflict resolution is LWW; simultaneous updates in different regions may overwrite. Avoid by scoping writers to a home region or using per-field conflict logic in the app.
- Strong consistency is regional only; cross-region reads are eventual. Critical reads should target the writer's region.
- Costs scale by region; every write is replicated to all regions.
- Features must match across replicas (GSIs, TTL, Streams). Schema drift isn't allowed.
- Backfills/migrations: bulk loads can flood replication; throttle or load one region and let replication fan out.

34. How would you query DynamoDB with SQL using Athena connectors, and what are the trade-offs?

Use the Athena DynamoDB connector (Lambda-based) for ad-hoc SQL when you can't or don't want to ETL first but know it's best for selective, key-based queries on small/medium datasets.

How it works

- Deploy the Athena DynamoDB connector (Serverless App) in your account/VPC.
- In Athena, create a data source for the connector. Define external tables (schema mapped to item attributes).
- Run SQL; the connector issues DynamoDB operations under the hood and streams results back.

When it's useful

- Quick exploration, governance via SQL, small joins via Athena federation.
- Key-based queries where predicates map to PK/SK (pushdown works reasonably).
- Occasional reporting without building a pipeline.

Trade-offs / caveats

- Performance: many queries degrade to parallel Scans with FilterExpressions. That burns RCUs and can be slow/expensive.
- Cost model: you pay DynamoDB RCUs + Lambda/DynamoDB connector execution (Athena bytes scanned is not the main driver).
- Limits: Lambda timeouts/concurrency, result size/time limits, eventual consistency (no strong reads), limited pushdown for complex predicates, and no magic for full-text search.
- Not for heavy analytics: wide aggregations, GROUP BY across large tables, or frequent joins are better via export to S3 → Athena on Parquet or Redshift.
- Schema/type quirks: DynamoDB's flexible types map imperfectly to SQL; watch for strings vs numbers, nested maps/lists, and nulls.

Good pattern

- Use the connector for occasional, selective SQL (by key, narrow ranges).
- For recurring or heavy reporting, export to S3 (Export to S3 / Streams → Firehose), store as Parquet with partitions, and query with Athena or load into Redshift.