Amazon Redshift re-invented

Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta
Venkatraman Govindaraju, TJ Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy
Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak
Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan
Sriram Subramanian, Doug Terry
Amazon Web Services
USA
redshift-paper@amazon.com

ABSTRACT

In 2013, Amazon Web Services revolutionized the data warehousing industry by launching Amazon Redshift, the first fully-managed, petabyte-scale, enterprise-grade cloud data warehouse. Amazon Redshift made it simple and cost-effective to efficiently analyze large volumes of data using existing business intelligence tools. This cloud service was a significant leap from the traditional on-premise data warehousing solutions, which were expensive, not elastic, and required significant expertise to tune and operate. Customers embraced Amazon Redshift and it became the fastest growing service in AWS. Today, tens of thousands of customers use Redshift in AWS's global infrastructure to process exabytes of data daily.

In the last few years, the use cases for Amazon Redshift have evolved and in response, the service has delivered and continues to deliver a series of innovations that delight customers. Through architectural enhancements, Amazon Redshift has maintained its industry-leading performance. Redshift improved storage and compute scalability with innovations such as tiered storage, multi-cluster auto-scaling, cross-cluster data sharing and the AQUA query acceleration layer. Autonomics have made Amazon Redshift easier to use. Finally, Amazon Redshift extends beyond traditional data warehousing workloads, by integrating with the broad AWS ecosystem with features such as querying the data lake with Spectrum, semistructured data ingestion and querying with PartiQL, Redshift ML, Federated Query and cross-service materialized views with Glue Elastic Views.

CCS CONCEPTS

• **Information systems → Database design and models**; **Database management system engines**.

KEYWORDS

Cloud Data Warehousing, Data Lake

ACM Reference Format:
Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, TJ Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, Doug Terry . 2022. Amazon Redshift re-invented. In *SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Amazon Web Services launched Amazon Redshift [10] in 2013. Today, tens of thousands of customers use Redshift in AWS's global infrastructure of 25 launched Regions and 81 Availability Zones (AZs) to process exabytes of data daily. The success of Redshift inspired a lot of innovation in the analytics segment [1, 2, 6, 18], which in turn has benefited customers. In the last few years, the service has evolved at a rapid pace in response to the evolution of the customers' use cases.

Redshift's development has focused on meeting four main customer needs: First, customers demand high-performance execution of increasingly complex analytical queries. Redshift provides industry-leading data warehousing performance through innovative query execution that blends database operators in each query fragment via code generation. State-of-the-art techniques like prefetching and vectorized execution, further improve its efficiency. This allows Redshift to scale linearly when processing from a few terabytes of data to petabytes.

Second, as our customers grow, they need to process more data and scale the number of users that derive insights from data. Redshift disaggregated its storage and compute layers to scale in response to changing workloads. Redshift scales up by elastically changing the size of each cluster and scales out for increased throughput via multi-cluster autoscaling that automatically adds and removes compute clusters to handle spikes in customer workloads. Users can consume the same datasets from multiple independent clusters.

Third, customers want Redshift to be easier to use. For that, Redshift incorporated machine learning based autonomics that fine-tune each cluster based on the unique needs of customer workloads. Redshift automated workload management, physical tuning, and the refresh of materialized views (MVs), along with preprocessing that rewrites queries to use MVs.

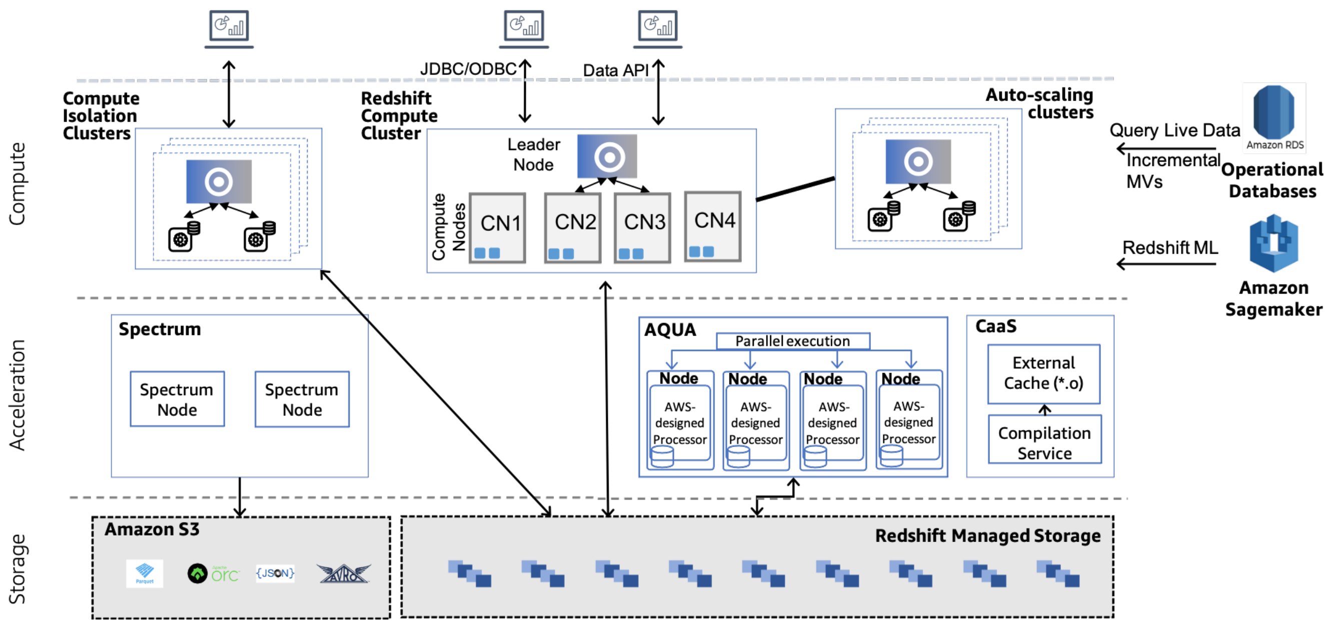


Figure 1: Amazon Redshift Architecture

Fourth, customers expect Redshift to integrate seamlessly with the AWS ecosystem and other AWS purpose built services. Redshift provides federated queries to transactional databases (e.g., DynamoDB [7] and Aurora [19]), Amazon S3 object storage and the ML services of Amazon Sagemaker. Through *Glue Elastic Views*, customers can create Materialized Views in Redshift that are incrementally refreshed on updates of base tables in DynamoDB or Amazon OpenSearch. Redshift also provides ingestion and querying of semistructured data with the SUPER type and PartiQL [**?**].

The rest of the paper is structured as follows. Section 2 gives an overview of the system architecture, data organization and query processing flow. It also touches on AQUA, Redshift's hardware-based query acceleration layer and Redshift's advanced query rewriting capabilities. Section 3 describes Redshift Managed Storage (RMS), Redshift's high-performance transactional storage layer and Section 4 presents Redshift's compute. Details on Redshift's smart autonomics are provided in Section 5. Lastly, Section 6 discusses how AWS and Redshift make it easy for their customers to use the best set of services for each use case and seamlessly integrate with Redshift's best-of-class analytics capabilities.

2 PERFORMANCE THAT MATTERS

2.1 Overview

Amazon Redshift is a column-oriented Massive Parallel Processing (MPP) data warehouse designed for the cloud [10]. Figure 1 depicts Redshift's architecture. A Redshift cluster consists of a single coordinator (leader) node, and multiple worker (compute) nodes. Data is stored on Redshift Managed Storage, backed by Amazon S3, and cached in compute nodes on locally-attached SSDs in a compressed column-oriented format. Tables are either replicated on every compute node or partitioned into multiple buckets that are distributed among all compute nodes. The partitioning can be automatically derived by Redshift based on the workload patterns and data characteristics, or, users can explicitly specify the partitioning style as round-robin or hash, based on the table's distribution key.

Amazon Redshift provides a wide range of performance and ease-of-use features to enable customers to focus on business problems. Concurrency Scaling allows users to dynamically scale-out in situations where they need more processing power to provide consistently fast performance for hundreds of concurrent queries. Data Sharing allows customers to securely and easily share data for read purposes across independent isolated Amazon Redshift clusters. AQUA is a query acceleration layer that leverages FPGAs to improve performance. Compilation-As-A-Service is a caching microservice of optimized generated code for the various query fragments executed in the Redshift fleet.

In addition to accessing Redshift using a JDBC/ODBC connection, customers can also use the Data API to access Redshift from any web service-based application. The Data API simplifies access to Redshift by eliminating the need for configuring drivers and managing database connections. Instead, customers can run SQL commands by simply calling a secure API endpoint provided by the Data API. In just a few months after launch, Data API has been serving millions of queries each day.

Figure 2 illustrates the flow of a query through Redshift. The query is received by the leader node ① and subsequently parsed, rewritten, and optimized ②. Redshift's cost-based optimizer includes the cluster's topology and the cost of data movement between compute nodes in its cost model to select an optimal plan. Planning leverages the underlying distribution keys of participating tables to avoid unnecessary data movement. For instance, if the join key in an equi-join matches the underlying distribution keys of both participating tables, then the chosen plan avoids any data movement by processing the join locally for each data partition ③.

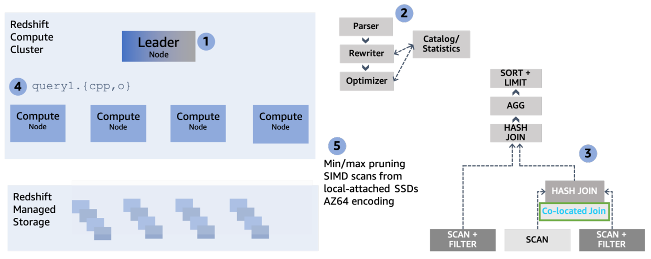
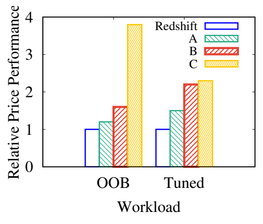
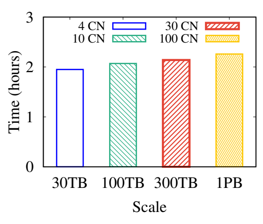


Figure 2: Query flow



(a) Price-Performance comparison



(b) Scaling TPC-DS from 30TB to 1PB

Figure 3: Price-Performance and Scalability

After planning, a workload management (WLM) component controls admission to Redshift's execution phase. Once admitted, the optimized plan is divided into individual execution units that either end with a blocking pipeline-breaking operation or return the final result to the user. These units are executed in sequence, each consuming intermediate results of previously executed units. For each unit, Redshift generates highly optimized C++ code that interleaves multiple query operators in a pipeline using one or more (nested) loops, compiles it and ships the binary to compute nodes (4). The columnar data are scanned from local-attached SSDs or hydrated from Redshift Managed Storage (5). If the execution unit requires exchanging data with other compute nodes over the network, then the execution unit consists of multiple generated binaries that exchange data in a pipelined fashion over the network.

Each generated binary is scheduled on each compute node to be executed by a fixed number of query processes. Each query process executes the same code on a different subset of data. Redshift's execution engine employs numerous optimizations to improve query performance. To reduce the number of blocks that need to be scanned, Redshift evaluates query predicates over *zone maps* i.e., small hash tables that contain the min/max values per block and leverages late materialization. The data that needs to be scanned post zone-map filtering is chopped into shared work-units, similar to [13, 17], to allow for balanced parallel execution. Scans leverage vectorization and *Single Instruction, Multiple Data* (SIMD) processing for fast decompression of Redshift's light-weight compression formats and for applying predicates efficiently. Bloom filters, created when building hash tables, are applied in the scan to further reduce the data volume that has to be processed by downstream query operators. Prefetching is leveraged to utilize hash tables more efficiently.

Redshift's execution model is optimized for the underlying Amazon EC2 Nitro hardware, resulting in industry leading price/performance. Figure 3(a) demonstrates Redshift's competitive edge when it comes to price-performance. It compares Amazon Redshift and three other cloud data warehouses and shows that Amazon Redshift delivers up to 3× better price-performance ratio out-of-the-box on untuned 3TB TPC-DS benchmark[1]. After all cloud data warehouses are tuned, Amazon Redshift has 1.5× better price performance than the second-best cloud data warehouse offering.

Customer data grows very rapidly rendering scalability a top priority. Figure 3(b) depicts the total execution time of tuned TPC-DS benchmark while scaling dataset size and hardware simultaneously. Redshift's performance remains nearly flat for a given ratio of data to hardware, as data volume ranges from 30TB to 1PB. This linear scaling to the petabyte-scale makes it easier, predictable and cost-efficient for customers to on-board new datasets and workloads.

The following sub-sections discuss selected aspects from Redshift's rewriting/optimization and execution model.

2.2 Introduction to Redshift Code Generation

Redshift is an analytical database focusing on fast execution of complex queries on large amounts of data. Redshift generates C++ code specific to the query plan and the schema being executed. The generated code is then compiled and the binary is shipped to the compute nodes for execution [9, 12, 14]. Each compiled file, called a *segment*, consists of a pipeline of operators, called *steps*. Each segment (and each step within it) is part of the physical query plan. Only the last step of a segment can break the pipeline.

```
1   // Loop over the tuples of R.
2   while (scan_step->has_next()) {
3     // Get next value for R.key.
4     auto field1 = fetcher1.get_next();
5     // Get next value for R.val.
6     auto field2 = fetcher2.get_next();
7     // Apply predicate R.val < 50.
8     if (field2 < constant1) {
9       // Hash R.key and probe the hash table.
10      size_t h1 = hash(field1) & (hashtable1_size - 1);
11      for (auto* p1 = hashtable1[h1]; p1 != nullptr; p1 = p1->next) {
12        // Evaluate the join condition R.key = S.key.
13        if (field1 == p1->field1) sum1 += field2;
14      }
15    }
16  }
```

Figure 4: Example of generated code

Figure 4 shows a high-level example of the generated C++ code on a single node cluster for a simple scan ⟶ join ⟶ aggregate query: '`SELECT sum(R.val) FROM R, S WHERE R.key = S.key AND R.val < 50`'. This segment contains the pipeline that scans base table `R` (lines 3-6), applies the filter (line 8), probes the hash table of `S` (line 10), and computes the aggregate `sum()` (line 13). Omitted for simplicity from Figure 4 are segments to build the hash table from table `S` and a final segment to combine the partial sums across compute nodes and return the result to the user. The generated code follows the principle of keeping the working set

[1]We use the Cloud DW benchmark [?] based on current TPC-DS and TPC-H benchmarks without any query or data modifications and compliant with TPC rules and requirements.

as close to the CPU as possible to maximize performance. As such, each tuple that is processed by multiple operators in a pipeline is typically kept in CPU registers until the tuple is sent over the network, materialized in main memory or flushed to disk.

The main property of this style of code generation is that it avoids any type of interpreted code since all operators for a specific query are generated in the code on the fly. This is in contrast to the standard Volcano execution model [8], where each operator is implemented as an iterator and function pointers or virtual functions pick the right operator at each execution step. The code generation model offers much higher throughput per tuple at the cost of latency, derived from having to generate and compile the code specific to each query. Section 2.6 explains how Redshift mitigates the compilation costs.

2.3 Vectorized Scans

In Figure 4 lines 4 and 6, function *get_next*() returns the next value for the corresponding field of the base table *R* defined by a unique *fetcher*. Such functions are inlined instead of virtual but are inherently pull-based rather than push-based since the underlying structure of the base table is too complicated to represent in the generated code directly. This model is relatively expensive as it retains a lot of state for each column scanned, easily exhausting the CPU registers if the query accesses more than a few columns. Moreover, the filter predicate evaluation (line 8) involves branching that may incur branch misprediction costs if the selectivity of a certain predicate is close to 50%, stalling the pipeline. Finally, each fetcher may inline a large amount of decompression code, which can significantly slow down compilation for very wide tables that access a large number of columns.

To address these issues, Redshift added a SIMD-vectorized scan layer to the generated code that accesses the data blocks and evaluates predicates as function calls. In contrast to the rest of the steps that compile the code on the fly, the vectorized scan functions are precompiled and cover all data types and their supported encoding and compression schemes. The output of this layer stores the column values of the tuples that qualify from the predicates to local arrays on the stack accessed by downstream steps. In addition to the faster scan code due to SIMD, this reduces the register pressure and the amount of inline code that must be compiled, leading to orders of magnitude faster compilation for certain queries on wide tables. The design combines column-at-a-time execution for a chunk of tuples during the scan step and tuple-at-a-time execution downstream for joins and aggregation steps. The size of the chunk that is processed column-at-a-time is dynamically determined during code generation based on the total width of the columns being accessed and the size of the thread-private (L2) CPU cache.

2.4 Reducing Memory Stalls with Prefetching

Redshift's pipelined execution avoids the materialization of intermediate results for the outer stream of joins and aggregates by keeping the intermediate column values in CPU registers. However, when building or probing hash tables as part of a hash join, or probing and updating hash tables as part of aggregations, Redshift incurs the full overhead of a cache miss if the hash table is too large to fit in the CPU cache. Memory stalls are prominent in this push-based model and may offset the eliminated cost of materialization for the outer stream in joins. The alternative would be to partition the input until the hash table of the partition fits in the CPU cache, thus avoiding any cache misses. That model however is infeasible for the execution engine since it may not be able to load large base tables in memory and thus cannot access payload columns using record identifiers. Instead, Redshift transfers all the needed columns downstream across the steps in the pipeline and incurs the latency of a cache miss when the hash table is larger than the CPU cache.

Since cache misses are an inherent property of our execution engine design, stalls are mitigated using prefetching. Our prefetching mechanism is integrated in the generated code and interleaves each probe in the hash table or Bloom filter with a prefetch instruction. Redshift keeps a circular buffer in the fastest (L1) CPU cache and, for each tuple that arrives, prefetches and pushes it in the buffer. Then, an earlier tuple is popped and pushed downstream to the rest of the steps. Once the buffer is filled up, rather than buffering multiple tuples, individual tuples are processed by pushing and popping one at a time from the buffer.

This model trades some materialization cost to the cache-resident prefetching buffer for the benefit of prefetching the hash table accesses and reducing the memory stalls. We have found this trade-off to always be beneficial if the hash table is larger than the CPU cache. If the hash table is known or expected to be small enough to fit in the CPU cache, this additional code is never generated. The same happens if the tuple is too wide and storing it in the buffer would be more expensive than paying for the cache miss stall. On the other hand, the prefetching code may be generated multiple times in the same nested loop if there are multiple joins and group-by aggregation in the same pipeline, while ensuring that the total size of all prefetching buffers is small enough to remain in the fastest (L1) CPU cache.

2.5 Inline Expression Functions

While the examples above cover basic cases of joins and aggregations with simple data types, an industrial-grade database needs to support complex data types and expression functions. The generated code includes pre-compiled headers with inline functions for all basic operations, like hashing and string comparisons. Scalar functions that appear in a query translate to inline or regular function calls in the generated code, depending on the complexity of the query. Most of these functions are scalar, as they process a single tuple, but may also be SIMD-vectorized internally.

In Redshift, most string functions are vectorized with SIMD code tailored to that particular function. One such example are the `LIKE` predicates that use the `pcmpestri` instruction in Intel CPUs, which allows sub-string matching of up to 16-byte patterns in a single instruction. Similarly, functions such as `UPPER()`, `LOWER()`, and case-insensitive string comparisons, use SIMD code to accelerate the ASCII path and only fall back to (optimized) scalar code when needed to handle more complex Unicode characters. Such optimizations are ubiquitous in expression functions to maximize throughput. The code generation layer inlines function calls that are on the critical path when advantageous.

2.6 Compilation Service

When a query is sent to Redshift, the query processing engine compiles optimized object files that are used for query execution. When the same or similar queries are executed, the compiled segments are reused from the cluster code compilation cache, which results in faster run times because there is no compilation overhead. While Redshift minimizes the overhead of query compilation, the very first set of query segments still incurs additional latency. In some cases, even a small additional latency can impact a mission critical workload with tight service-level-agreements (SLAs), particularly when a large number of segments need to be compiled increasing contention for cluster resources.

The compilation service uses compute and memory resources beyond the Redshift cluster to accelerate query compilation through a scalable and secure architecture. The compilation service caches the compiled objects off-cluster in an external code cache to serve multiple compute clusters that may need the same query segment. During query processing, Redshift generates query segments and leverages the parallelism of the external compilation service for any segments that are not present in a cluster's local cache or the external code cache. With the release of compilation service cache hits across the Amazon Redshift fleet have increased from 99.60% to 99.95%. In particular, in 87% of the times that an object file was not present in a cluster's local code cache, Redshift found it in the external code cache.

2.7 CPU-Friendly Encoding

Performance is closely tied to CPU and disk usage. Naturally, Redshift uses compression to store columns on disk. Redshift supports generic byte-oriented compression algorithms such as LZO and ZSTD, as well as optimized type-specific algorithms. One such compression scheme is the recent AZ64 algorithm, which covers numeric and date/time data types. AZ64 achieves compression that is comparable to ZSTD (which compresses better than LZO but is slightly slower) but with faster decompression rate. For example, a full 3TB TPC-H run improves by 42% when we use AZ64 instead of LZO for all data types that AZ64 supports.

2.8 Adaptive Execution

Redshift's execution engine takes runtime decisions to boost performance by changing the generated code or runtime properties on the fly based on execution statistics. For instance, the implementation of Bloom filters (BFs) in Redshift demonstrates the importance of dynamic optimizations [?]. When complex queries join large tables, massive amounts of data might be transferred over the network for the join processing on the compute nodes and/or might be spilled to disk due to limited memory causing network and/or I/O bottlenecks that can impact query performance. Redshift uses BFs to improve the performance of such joins. BFs efficiently filter rows at the source that do not match the join relation, reducing the amount of data transferred over the network or spilled to disk.

At runtime, join operations decide the amount of memory that will be used to build a BF based on the exact amount of data that has been processed. For example, if a join spills data to disk, then the join operator can decide to build a larger BF to achieve lower false-positive rates. This decision increases BFs pruning power and may reduce spilling in the probing phase. Similarly, the engine monitors the effectiveness of each BF at runtime and disables it when the rejection ratio is low since the filter burdens performance. The execution engine can re-enable a BF periodically since temporal pattern of data may render a previously ineffective BF to become effective.

2.9 AQUA for Amazon Redshift

Advanced Query Accelerator (AQUA) is a multi-tenant service that acts as an off-cluster caching layer for Redshift Managed Storage and a push-down accelerator for complex scans and aggregations. AQUA caches hot data for clusters on local SSDs, avoiding the latency of pulling data from a regional service like Amazon S3 and reducing the need to hydrate the cache storage in Redshift compute nodes. To avoid introducing a network bottleneck, the service provides a functional interface, not a storage interface. Redshift identifies applicable scan and aggregation operations and pushes them to AQUA, which processes them against the cached data and returns the results. Essentially, AQUA is computational storage at a data-center scale. By being multi-tenant, AQUA makes efficient use of expensive resources, like SSDs, and provides a caching service that is unaffected by cluster transformations such as resize and pause-and-resume.

To make AQUA as fast as possible, we designed custom servers that make use of AWS's Nitro ASICs for hardware-accelerated compression and encryption, and leverage FPGAs for high-throughput execution of filtering and aggregation operations. The FPGAs are not programmed on a per-query basis, but rather used to implement a custom multi-core VLIW processor that contains database types and operations as pipelined primitives. A compiler within each node of the service maps operations to either the local CPUs or the accelerators. Doing this provides significant acceleration for complex operations that can be efficiently performed on the FPGA.

2.10 Query Rewriting Framework

Redshift features a novel DSL-based Query Rewriting Framework (QRF), which serves multiple purposes: First, it enables rapid introduction of novel rewritings and optimizations so that Redshift can quickly respond to customer needs. In particular, QRF has been used to introduce rewriting rules that optimize the order of execution between unions, joins and aggregations. Furthermore it is used during query decorrelation, which is essential in Redshift, whose execution model benefits from large scale joins rather than brute force repeated execution of subqueries.

Second, QRF is used for creating scripts for incremental materialized view maintenance (see Section 5.4) and enabling answering queries using materialized views. The key intuition behind QRF is that rewritings are easily expressed as pairs of a pattern matcher, which matches and extracts parts of the query representation (AST or algebra), and a generator that creates the new query representation using the parts extracted by the pattern matcher. The conceptual simplicity of QRF has enabled even interns to develop complex decorrelation rewritings within days. Furthermore, it enabled Redshift to introduce rewritings pertaining to nested and semistructured data processing (see Section 6.4) and sped up the expansion of the materialized views scope.

3 SCALING STORAGE

The storage layer of Amazon Redshift spans from memory, to local storage, to cloud object storage (Amazon S3) and encompasses all the data lifecycle operations (i.e., commit, caching, prefetching, snapshot/restore, replication, and disaster-recovery). Storage has gone through a methodical and carefully deployed transformation to ensure durability, availability, scalability, and performance:

Durability and Availability. The storage layer builds on Amazon S3 and persists all data to Amazon S3 with every commit. Building on Amazon S3 allows Redshift to decouple data from the compute cluster that operates on the data. It also makes the data durable and building an architecture on top of it enhances availability.

Scalability. Using Amazon S3 as the base gives virtually unlimited scale. Redshift Managed Storage (RMS) takes advantage of optimizations such as data block temperature, data block age, and workload patterns to optimize performance and manage data placement across tiers of storage automatically.

Performance. Storage layer extends into memory and algorithmic optimizations. It dynamically prefetches and sizes the in-memory cache, and optimizes the commit protocol to be incremental.

3.1 Redshift Managed Storage

The Redshift managed storage layer (RMS) is designed for a durability of 99.999999999% and 99.99% availability over a given year, across multiple availability zones (AZs). RMS manages both user data as well as transaction metadata. RMS builds on top of the AWS Nitro System, which features high bandwidth networking and performance indistinguishable from bare metal. Compute nodes use large, high performance SSDs as local caches. Redshift leverages workload patterns and techniques such as automatic fine-grained data eviction and intelligent data prefetching, to deliver the performance of local SSD while scaling storage automatically to Amazon S3.

Figure 5 shows the key components of RMS extending from in-memory caches to committed data on Amazon S3. Snapshots of data on Amazon S3 act as logical restore points for the customer. Redshift supports both the restore of a complete cluster, as well as of specific tables, from any available restore point. Amazon S3 is also the data conduit and source of truth for data sharing and machine learning. RMS accelerates data accesses from S3 by using a prefetching scheme that pulls data blocks into memory and caches them to local SSDs. RMS tunes cache replacement to keep the relevant blocks locally available by tracking accesses to every block. This information is also used to help customers decide if scaling up their cluster would be beneficial. RMS makes in-place cluster resizing a pure metadata operation since compute nodes are practically stateless and always have access to the data blocks in RMS. RMS is metadata bound and easy to scale since data can be ingested directly into Amazon S3. The tiered nature of RMS where SSDs act as cache makes swapping out of hardware convenient. RMS-supported Redshift RA3 instances provide up to 16PBs of capacity today. The in-memory disk-cache size can be dynamically changed for balancing performance and memory needs of queries.

A table's data is partitioned into *data slices* and stored as logical chains of blocks. Each data block is described by its block header (e.g., identity, table ownership and slice information) and indexed

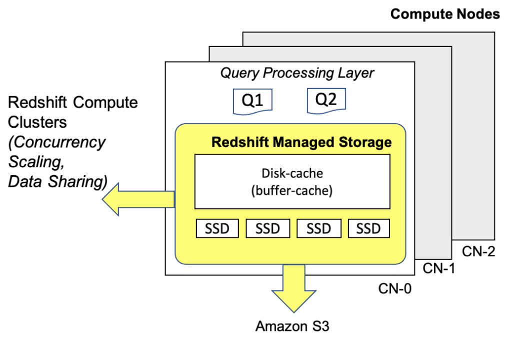


Figure 5: Redshift Managed Storage

via an in-memory construct, called *superblock*. The superblock is an indexing structure similar to many filesystems. Queries reach the relevant data blocks by using zone maps to scan the superblock. The superblock also contains query tracking information for data blocks owned by live queries.

Transactions are synchronously committed to Amazon S3 by RMS. This enables multiple clusters to access live and transactionally consistent data. Writing through to Amazon S3 across different AZs is achieved by batching data writes and hiding latencies under synchronization barriers. State is owned and managed by one cluster, while concurrent readers and writers provide compute scaling on the top of RMS. The concurrent clusters spun up on demand rely on snapshot isolation and prioritized on-demand fetching of data to cater to the query requests. Data deleted from the main cluster gets garbage collected from Amazon S3 once all reader references are cleared. RMS uses a combination of time-to-live and on-demand deletes to make sure data does not leak on transaction rollback.

Since data is always backed in Amazon S3, the loss of local SSDs can be tolerated ensuring data durability. Redshift provides disaster recovery with RPO=0, where a cluster can be relocated to a same or different AZ in the event of cluster loss or data center failure.

3.2 Decoupling Metadata from Data

Decoupling metadata from data enables Elastic Resize and Cross-Instance Restore. Both features shuffle metadata from one cluster configuration to another and thus separating metadata from data can lead to an efficient implementation. Elastic Resize allows customers to reconfigure their cluster in minutes by adding nodes to get better performance and more storage for demanding workloads or by removing nodes to save cost. Cross-Instance Restore allows users to restore snapshots taken from a cluster of one instance type to a cluster of different instance type or different number of nodes.

Redshift implements the features mentioned above as follows. First, it ensures that a copy of data is in Amazon S3. This allows the recovery from truly rare events such as multiple hardware failures. Before any reconfiguration, Redshift takes account of the data in the cluster and generates a plan of how to reconfigure with minimal data movement that also results in a balanced cluster. Redshift records counts and checksums on the data before reconfiguration and validates correctness after completion. In case of restore, Redshift records counts of number of tables, blocks, rows, bytes used, and data distribution, along with a snapshot. It validates the counts and checksums after restore before accepting new queries.

Cross-Instance Restore and resize leverage the Elastic Resize technology to provide migration in minutes. Both Elastic Resize and Cross-Instance Restore are heavily used features, where customers use them for reconfiguration over 15,000 times a month. The failure rates are less than 0.0001%. Finally, the elasticity gained by reconfiguring metadata enabled the transition to RMS.

3.3 Expand Beyond Local Capacity

Redshift enhances scalability by using Amazon S3 to expand storage capacity of a cluster and utilizing the local memory and SSD as caches. Many changes have been made to enable this transition: upgrading superblock to support larger capacities, modifying local layout to support more metadata, modifying how snapshots are taken, transforming how to rehydrate and evict data, etc. Due to space constraints, the section focuses on two components: tiered-storage cache and dynamic buffer cache.

The tiered-storage cache keeps track of the number of accesses of data blocks so that each cluster maintains its working set locally. It builds a two-level clock-based cache replacement policy to track data blocks stored in local disk for each compute node. Cache policy places a cold data block B (i.e., accessed for the first time by customer query) in the low-level clock cache and increases B's reference count on every access. When B becomes hot (i.e., accessed multiple times), cache policy promotes it to the high-level clock cache. During eviction, the reference count of each block pointed by clock pointer is decremented. When the reference count of B is zero, B is either demoted from high-level clock to low-level clock or evicted from the cache.

RMS uses the tiered-storage cache to drive rehydration (i.e., what data to cache on local SSDs) after a cluster reconfiguration (e.g., Elastic Resize, cluster restore, hardware failures). In all these scenarios, the compute nodes rehydrate their local disks with the data blocks that have highest possibility to be accessed by customer queries. With this optimization, customer's queries achieve more than 80% local disk hit rate at 20% rehydration completion.

Finally, to boost performance, Redshift utilizes a dynamic disk-cache on top of tiered-storage cache to maintain the hottest blocks in memory. In addition, the disk-cache keeps other blocks created by queries such as new data blocks and query-specific temporary blocks. Disk-cache automatically scales up when memory is available and proactively scales down as the system is near memory exhaustion. These changes lead up to a 30% performance improvement in benchmarks as well as customer workloads.

3.4 Incremental Commits

To use Amazon S3 as primary storage requires incremental commits to reduce data footprint and cost. RMS only captures the exact data changes since last commit and updates the commit log accordingly. Persistent data structures are also updated incrementally. Redshift's log-based commit protocol, which replaced the earlier redirect-on-write protocol, decouples the in-memory structure from the persisted structure, where the persisted superblock simply records a log of changes. The log-based commit improves commit performance by 40% by converting a series of random I/Os into a few sequential log appends. Since RMS provides durable and highly available data access across multiple AZs and regions, the metadata can be shared and replayed on globally distributed compute. This log-structured metadata reduces the cost of features like concurrency scaling and data sharing; both of these features access transactionally consistent data by applying the log onto their local superblock.

3.5 Concurrency Control

Redshift implements Multi-version Concurrency Control (MVCC) where readers neither block nor are blocked and writers may only be blocked by other concurrent writers. Each transaction sees a consistent snapshot of the database established by all committed transactions prior to its start. Redshift enforces serializable isolation, thus avoiding data anomalies such as lost updates and read-write skews [3, 16]. With that, Redshift provides industry-leading performance without trading off data correctness, and our customers do not need to analyze whether a workload should run on lower transactional isolation levels.

Our legacy implementation used a graph-based mechanism to track dependencies between transactions to avoid cycles and enforce serializability. This required tracking individual state of each transaction well after they were committed, until all other concurrent transactions committed as well. We recently adopted a new design based on a Serial Safety Net (SSN) as a certifier on top of Snapshot Isolation [20]. This heuristic-based design allows us to guarantee strict serializability in a more memory-efficient manner, using only summary information from prior committed transactions. As analyzed in [20], the SSN algorithm is an improvement over comparable algorithms such as Serializable Snapshot Isolation (SSI) [16]. Optimizations and enhancements were added to the base SSN algorithm, primarily to be backwards compatible with our legacy certifier. Such enhancements include aborting certain transactions at the time the operation is executed, rather than performing the calculations at commit time as the original SSN design does. There is a significant reduction in resource utilization compared to the legacy design. In particular, memory footprint of this component was reduced by as much as 8GB, depending on workload.

4 SCALING COMPUTE

Every week Redshift processes billions of queries that serve a diverse set of workloads that have varying performance requirements. ETL workloads have strict latency SLAs that downstream reporting depends on. Interactive dashboards on the other hand have high concurrency requirements and are extremely sensitive to response times. Recently, Redshift onboarded a customer workload where the 90th percentile response time requirement was <1s. Redshift workload patterns have evolved and our customers choose one or more of the following compute scaling options to get the best price/performance to serve their needs.

4.1 Cluster Size Scaling

Elastic Resize allows customers to quickly add or remove compute nodes from their cluster depending on their current compute needs.

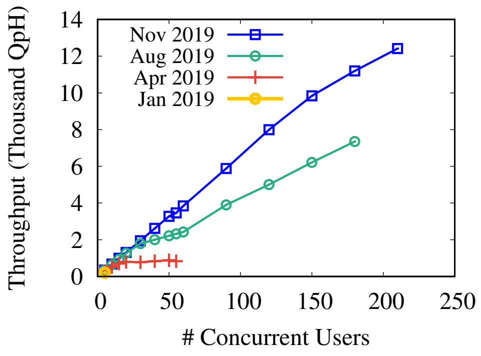


Figure 6: Concurrency improvements

It is a light-weight metadata operation that does not require reshuffling the underlying data distribution. Instead, Elastic Resize re-organizes the data partition assignment to ensure that after the resize, all compute nodes are balanced with respect to the number of data partitions. After the resize, the data partitions that have been moved are re-hydrated from S3 in the background; prioritizing on-demand requests and hot data. Because of the data partition reassignment, the number of data partitions per node after an Elastic Resize differs from those of a Redshift cluster that has not been resized. In order to provide consistent query performance after Elastic Resize, Redshift decouples compute parallelism from data partitions. When there is more compute parallelism than data partitions, multiple compute processes are able to share work from an individual data partition. When there are more data partitions than compute parallelism, individual compute processes are able to work on multiple data partitions. Redshift achieves this by generating shareable work units [13, 17] when scanning Redshift tables and employing a compute-node centric view of query operators, where all query processes of a compute node collaborate on processing all data partitions on that node.

4.2 Concurrency Scaling

Concurrency Scaling allows Redshift to dynamically scale-out when users need more concurrency than what a single Redshift cluster can offer. As the number of concurrent queries increases, Concurrency Scaling transparently handles the increase in the workload. With Concurrency Scaling customers maintain a single cluster endpoint to which they submit their queries. When the assigned compute resources are fully utilized and new queries start queuing, Redshift automatically attaches additional Concurrency Scaling compute clusters and routes the queued queries to them. Concurrency Scaling clusters re-hydrate data from RMS. Figure 6 shows the concurrency improvements Redshift achieved over a period of one year, in terms of query throughput versus number of concurrent clients. The workload used is 3TB TPC-DS. Redshift achieves linear query throughput for hundreds of concurrent clients.

4.3 Compute Isolation

Redshift allows customers to securely and easily share live data across different Redshift compute clusters and AWS accounts. This enables different compute clusters to operate on a single source of data and eliminates the complexity of pipelines that maintain copies of data. Data can be shared at many levels, such as schemas, tables, views, and user-defined functions. In order to access a producer's data, a producer cluster must first create a data share and then grant access to a consumer. Redshift manages the resulting metadata and IAM policies which facilitate authentication and authorization of shares between producers and consumers. There is no restriction on the number of consumer clusters a data share can have.

When a consumer cluster queries a shared object, one or more metadata requests are issued. A metadata request is only possible after a consumer cluster has been authorized to access a data share. Each metadata request flows through a directory service and proxy layer which form a networking mesh between producers and consumers of data shares. The proxy performs authentication and authorization of requests at low latencies and routes consumer metadata requests to the appropriate producer, which can serve the requests even if it is paused. After a consumer cluster receives the metadata, it reads the required data blocks from RMS and executes the query. The data blocks are cached locally on the consumer cluster. If a subsequent query accesses the same data blocks, those reads are served locally as long as the blocks have not been evicted from the consumer cluster.

5 AUTOMATED TUNING AND OPERATIONS

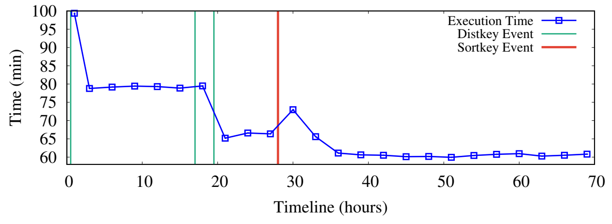
One of Redshift's key premises since its launch was the case for simplicity [10]. From the early days, Redshift simplified many aspects of traditional data warehousing, including cluster maintenance, patching, monitoring, resize, backups and encryption.

However, there were still some routine maintenance tasks and performance knobs whose fine-tuning required the expertise of a database administrator. For instance, customers had to schedule maintenance tasks (e.g., vacuum) and decide on performance parameters such as distribution keys. To alleviate this pain, Redshift invested heavily in maintenance automation and machine learning based autonomics.

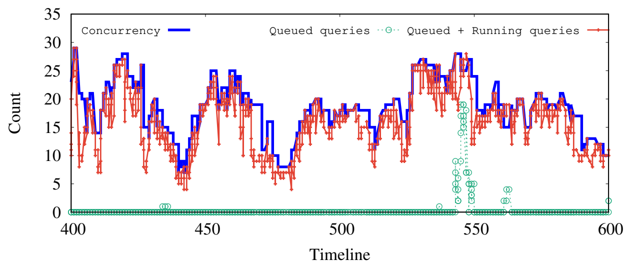
Today, Redshift runs common maintenance tasks like vacuum, analyze or the refresh of materialized views in the background without any performance impact to customer workloads. Automatic workload management dynamically chooses query concurrency and memory assignment based on workload characteristics. Furthermore, Redshift monitors and analyzes customer workloads and identifies opportunities for performance improvement, e.g., by automatically applying distribution and sort key recommendations. Finally, Redshift employs state of the art forecasting techniques to make additional nodes available as soon as possible for node failures, cluster resumption and concurrency scaling, thus, further improving query latency and reducing down time.

5.1 Automatic Table Optimizations

Table properties like distribution and sort keys allow Redshift customers to optimize the performance of their workloads. Distribution keys facilitate efficient collocated joins and aggregates; they increase parallelism and minimize data reshuffling over the network when data distribution required by the query matches the physical distribution of the underlying tables. Sort keys facilitate optimizations like zone maps and improve the efficiency of sort-based operations like merge join. Choosing appropriate distribution and



(a) ATO performance improvements



(b) AutoWLM on real customer workload

Figure 7: Automated Tuning Examples

sort keys is not always easy and often requires detailed workload knowledge. Also, evolving workloads may require reconfiguration of the physical layout.

Redshift has now fully automated the selection and the application process of distribution and sort keys through *Automatic Table Optimization* (ATO). ATO analyzes the cluster workload to generate distribution and sort key recommendations and provides tools for their seamless application on user tables. ATO periodically collects query execution metadata like the optimized query plans, cardinalities and predicate selectivities to generate recommendations. In addition, it estimates the expected benefit of each recommendation and only surfaces highly beneficial recommendations.

The distribution key advisor focuses on minimizing the overall network cost for a given workload. Distribution keys cannot be chosen in isolation but require a holistic look at all tables that participate in the workload. Thus, ATO builds a weighted join graph from all joins in the workload, and then selects distribution keys that minimize the total network distribution cost [15].

Similarly, the sort key advisor focuses on reducing the the data volume that needs to be retrieved from disk. Given the query workload, ATO analyzes the selectivity of all range restricted scan operations and recommends sort keys that improve the effectiveness of zone map filtering, i.e., pruning of data blocks.

To apply the recommendations, Redshift offers two options to the customers. First, through the console the users can inspect and manually apply recommendations through simple DDLs. Second, automatic background workers periodically apply beneficial recommendations without affecting the performance of a customers' workload; the workers run when the cluster is not busy and apply recommendations incrementally, backing off whenever the load on the cluster increases.

Figure 7(a) illustrates the effectiveness of ATO on a 5-node *RA3.16xlarge* instance derived from an out-of-box TPC-H 30TB dataset without distribution or sort keys in any of the tables. The TPC-H benchmark workload runs on this instance every 30 minutes and we measure the end-to-end runtime. Over time, more and more optimizations are automatically applied reducing the total workload runtime. After all recommendations have been applied, the workload runtime is reduced by 23% (excluding the first execution that is higher due to compilation).

5.2 Automatic Workload Management

Admitting queries for execution has a wide range of implications for concurrently executing queries. Admitting too few, causes high latency for queued queries and poor resource utilization (e.g., CPU, I/O or memory) for executing ones. Admitting too many, reduces the number of queued queries but has a negative effect on resource utilization as resources become over-saturated. For instance, too little memory per query results in more queries spilling to disk, which has a negative effect on query latency. Redshift is used for a wide range of rapidly changing workloads with different resource needs. To improve response times and throughput, Redshift employs machine learning techniques that predict query resource requirements and queuing theory models that adjust the number of concurrently executing queries.

Redshift's Automatic Workload Manager (AutoWLM) is responsible for admission control, scheduling and resource allocation. When a query arrives, AutoWLM converts its execution plan and optimizer-derived statistics into a feature vector, which is evaluated against machine learning models to estimate metrics like execution time, memory consumption and compilation time. Based on these characteristics, the query finds its place in the execution queue. Redshift uses execution time prediction to schedule short queries ahead of long ones. A query may proceed to execution if its estimated memory footprint can be allocated from the query memory pool. As more queries are admitted for execution, AutoWLM monitors the utilization of cluster's resources using a feedback mechanism based on queuing theory. When utilization is too high, AutoWLM throttles the concurrency level to prevent increase in query latency due to over-saturated query resources.

Figure 7(b) illustrates AutoWLM in action on a real customer workload. AutoWLM is able to adjust the concurrency level in tandem with the number of query arrivals leading to minimum queuing and execution time. At time 545, AutoWLM detects that workload at current concurrency level is leading to IO/CPU saturation and therefore, it reduces concurrency level. This leads to increase in queuing because newly arrived queries are not allowed to execute. To avoid such queueing, customers can either opt for concurrency scaling (Section 4.2) or define *query priorities* to allow prioritization of more crucial queries.

During admission control, AutoWLM employs a weighted round-robin scheme for scheduling higher priority queries more often than low priority ones. In addition, higher priority queries get a bigger share of hardware resources. Redshift divides CPU and I/O in exponentially decreasing chunks for decreasing priority level when queries with different priorities are running concurrently. This accelerates higher priority queries exponentially as compared to lower priority ones. If a higher priority query arrives after a lower priority query started executing, AutoWLM preempts (i.e., aborts and restarts) the lower priority query to make space. In case of several low priority queries, AutoWLM preempts the query that is furthest from completion, using the query's estimated execution time. To prevent starvation of lower priority queries, a query's probability of being preempted is reduced with each preemption. Even so, if too many queries are preempted, throughput suffers. To remedy this, AutoWLM prevents preemption if wasted work ratio (i.e., time lost due to preemption over total time) breaches a threshold. As a result of query priorities, when cluster resources are exhausted, mostly lower priority queries would queue to let higher priority workloads meet their SLAs.

5.3 Query Predictor Framework

AutoWLM relies on machine learning models to predict the memory consumption and the execution time of a query. These models are maintained by Redshift's Query Predictor Framework. The framework runs within each Redshift cluster. It collects training data, trains an XGBOOST model and permits inference whenever required. This allows Redshift to learn from the workload and self-tune accordingly to improve performance. Having a predictor on the cluster itself helps to quickly react to changing workloads, which would not be possible if the model was trained off-cluster and only used on the cluster for inference. Code compilation sub-system also makes use of the query predictor framework to pick between optimized and debug compilation and improve the overall query response time.

5.4 Materialized Views

In a data warehouse environment, applications often need to perform complex queries on large tables. Processing these queries can be expensive in terms of system resources as well as the time it takes to compute the results. Materialized views (MVs) are especially useful for speeding up queries that are predictable and repeated. Redshift automates the efficient maintenance and use of MVs in three ways. First, it incrementally maintains filter, projection, grouping and join in materialized views to reflect changes on base tables. Since Redshift thrives on batch operations, MVs are maintained in a deferred fashion, so that the transactional workload is not slowed down.

Second, Redshift can automate the timing of the maintenance. In particular, Redshift detects which MVs are outdated and maintains a priority queue to choose which MVs to maintain in the background. The prioritization of refreshes is based on combining (1) the utility of a materialized view in the query workload and (2) the cost of refreshing the materialized view. The goal is to maximize the overall performance benefit of materialized views. For 95% of MVs, Redshift brings the views up-to-date within 15 minutes of a base table change.

Third, Redshift users can directly query an MV but they can also rely on Redshift's sophisticated MV-based autorewriting to rewrite queries over base tables to use the best eligible materialized views to optimally answer the query. MV-based autorewriting is cost based and proceeds only if the rewritten query is estimated to be faster than the original query. For aggregated MVs, autorewritten queries are up to 2x faster for 50% of the clusters and up to 5x faster for 25% of clusters. Both MV incremental maintenance and autorewriting are internally using Redshift's novel DSL-based query rewriting framework, which enables the Redshift team to keep expanding the SQL scope of incremental view maintenance and autorewriting.

5.5 Smart Warmpools, Gray Failure Detection and Auto-Remediation

At the scale at which Redshift operates, hardware failures are the norm and operational health is of the utmost importance. Over the years, the Redshift team has developed elaborate monitoring, telemetry and auto-remediation mechanisms.

Redshift uses a smart warmpool architecture, which enables prompt replacements of faulty nodes, rapid resumption of paused clusters, automatic concurrency scaling, failover, and many other critical operations. Warmpools are a group of EC2 instances that have been pre-installed with software and networking configurations. Redshift maintains a distinct warmpool in each AWS availability zone for each region. In order to guarantee optimal inter-node communication, clusters are configured with compute nodes from the same availability zones.

Keeping all of the aforementioned operations low latency requires a high hit rate when a node is acquired from the warmpool. To guarantee high hit rate, Redshift built a machine learning model to forecast how many EC2 instances are required for a given warmpool at any time. This system dynamically adjusts warmpools in each region and availability zone to save on infrastructure cost without sacrificing latency.

While fail-stop failures are relatively easy to detect, the gray failures are way more challenging [11]. For gray failures, Redshift has developed outlier detection algorithms that identify with confidence sub-performing components (e.g., slow disks, NICs, etc.) and automatically trigger the corresponding remediation actions.

6 USING THE BEST TOOL FOR THE JOB

AWS offers multiple purpose-built services, i.e., services that excel in their objective. These purpose-built services include the scalable object storage service Amazon S3, transactional database services (e.g., DynamoDB and Aurora), and the ML services of Amazon Sagemaker. AWS and Redshift make it easy for their users to use the best service for each job and seamlessly integrate with Redshift. This section describes the major integration points of Redshift.

6.1 Data in Open File Formats in Amazon S3

In addition to the data in RMS, Redshift also has the ability to access data in open file formats in Amazon S3 via a feature called Spectrum [5]. Redshift Spectrum facilitates exabyte scale analytics of data lakes and is extremely cost effective with pay-as-you-go billing based on amount of data scanned. Spectrum provides massive scale-out processing, performing scans and aggregations of data in Parquet, Text, ORC and AVRO formats. Amazon maintains a fleet of multi-tenant Spectrum nodes and leverages 1:10 fan-out ratio from Redshift compute slice to Spectrum instance. These nodes are acquired during query execution and released subsequently.

In order to leverage Spectrum, Redshift customers register their external tables in either Hive Metastore, AWS Glue or AWS Lake Formation catalog. During query planning, Spectrum tables are localized into temporary tables to internally represent the external

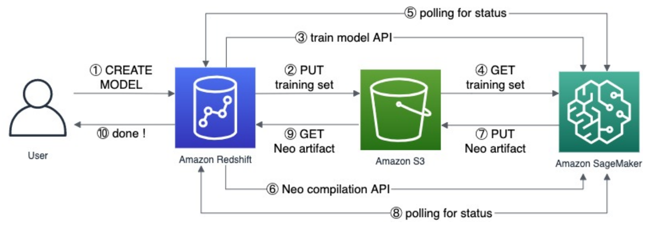


Figure 8: Redshift ML: Create model

table. Subsequently, queries are rewritten and isolated to Spectrum sub-queries in order to pushdown filters and aggregation. Either through S3 listing or from manifests belonging to partitions, the leader node generates scan ranges. Along with the serialized plan, scan ranges are sent over to compute nodes. An asynchronous Thrift request is made to the Spectrum instance with a presigned S3 URL to retrieve S3 objects. To speed-up repetitive Spectrum queries, Redshift externalized a result cache and also added support for materialized views over external tables with complete refresh.

6.2 Redshift ML with Amazon Sagemaker

Redshift ML makes it easy for data analysts to train machine learning models and perform prediction (inference) using SQL. For example, a user can train a churn detection model with customer retention data present in Redshift and then predict using that model so that the marketing team can offer incentives to customers at risk of churning. Internally, Redshift ML uses Amazon SageMaker, a fully managed machine learning service. After a model has been trained, Redshift makes available a SQL function that performs inference and can be used directly in SQL queries.

Rather than reinventing the wheel, Redshift ML complements the Sagemaker offering by leveraging the strengths of the two services. Redshift ML brings the model to the data rather than vice versa. This simplifies the machine learning pipelines while enabling at-scale, cost-efficient, in-database prediction.

Figure 8 illustrates the pipeline to create an ML model in Redshift. When a customer initializes the process, Redshift ML uses sampling to unload the proper amount of data from the Redshift cluster to an S3 folder. Subsequently, in the Redshift AUTO training mode, a Sagemaker Autopilot job is initiated under-the-hood. It discovers the best combination of preprocessor, model and hyperparameters. In order to perform ML inference locally using this model, Redshift invokes the Amazon Sagemaker Neo service to compile the model. Neo transforms the classic machine learning models from the Scikit-Learn library into inference code.

Neo abstracts the multiple steps (preprocessor, inference algorithm and post-processor) into a pipeline for executing the full sequence. Redshift localizes the compiled artifacts and registers an inference function corresponding to the created model. Upon invocation of the inference function, Redshift generates C++ code that loads and invokes the localized models. All this activity (discovery of best preprocessors, algorithm and hyperaparameter tuning and localization of inference) happens automatically. Thus, the user obtains the benefits of the purpose-built ML tools, while staying in Redshift. Redshift ML can also operate in an AUTO OFF mode, where the user takes control of preprocessing and algorithm/model choice. To begin with, Redshift ML has introduced XGBoost and Multi-layer perceptron in the AUTO OFF path.

Staying true to utilizing the best tool for the job, Redshift can also delegate inference to Sagemaker. This ability is needed for Vertical AI (e.g., sentiment analysis) and opens the gate towards the use of specialized GPU hardware when need be.

6.3 OLTP Sources with Federated Query and Glue Elastic Views

AWS offers purpose-built OLTP-oriented database services, such as DynamoDB [7], Aurora Postgres, and Aurora MySQL [19]. AWS users have been getting top OLTP performance through these services but often need to analyze the collected data with Redshift. Redshift facilitates both the in-place querying of data found in the OLTP services, using Redshift's Federated Query, as well as the seamless copying and synchronization of data to Redshift, using Glue Elastic Views.

A popular approach for users to ingest and/or query live data from their OLTP database services is through Redshift Federated Query, where Redshift makes direct connections to the customer's Postgres or MySQL instances and fetches live data. Federated Query enables real-time reporting and analysis and simplified ETL processing, without the need to extract data from the OLTP service to Amazon S3 and load it afterwards.

The ability to access both Redshift tables and Postgres or MySQL tables in a single query is effective. It allows users to have a fresh "union all" view of their data, including Spatial data [4], as part of their business intelligence (BI) and reporting applications. In addition, Redshift Federated Query includes a query optimization component to determine the most efficient way to execute a federated query, sending subqueries with filters and aggregations into the OLTP source database to speed up query performance and reduce data transfer over the network. Note that these subqueries often need to be augmented/transformed to ensure they are semantically correct, as different DBMS systems may have slightly different query operator semantics. The query optimizer also considers table and column statistics available on the OLTP store for query planning purposes.

The integration of Glue Elastic Views (GEV) with Redshift facilitates and accelerates the ingestion of data from OLTP sources into Redshift. GEV enables the definition of views over AWS sources (presently DynamoDB and Amazon Open Search, with additional sources to be added later). The views are defined in PartiQL, a language backwards-compatible with SQL, which can operate on both schemaful and schemaless sources (such as DynamoDB). GEV offers a journal of changes to the view, i.e., a stream of insert and delete changes. Consequently, the user can define Redshift materialized views that reflect the data of the GEV views; that is, Redshift materialized views that upon refresh, consume the stream of inserts and deletes received from the journal. The user may go a step beyond materializing the full GEV view to perform additional on-the-fly projection, transformation, filtering and aggregation on it.

The GEV view transforms the source data in ways that meet business logic needs (e.g. filter out sensitive rows and attributes) but also in ways that resolve the type mismatches and data organization mismatches between the source and the Redshift target. For example, the view will cast the data into Redshift-compatible types, one or more views may normalize nested data, while other views may catch cases where unexpected attributes or types appear.

In addition, GEV resolves the mismatch between the different ingestion strengths of the sources and the Redshift target. Users perform multiple small transactions on DynamoDB but it is not viable to have one insert (or delete) transaction at Redshift for each DynamoDB transaction. Redshift's strength is in ingesting big batches of data. This mismatch between DynamoDB and Redshift is solved by GEV essentially playing the role of a buffer, while Redshift is drawing batches of data (insert/delete changes from the journal) for its materialized view at the proper time: Relatively infrequently, so that significantly large batches can be accumulated for throughput optimization, but also frequently enough so that queries on the materialized view see fresh snapshots.

6.4 Redshift's SUPER Schemaless Processing

Redshift offers yet another efficient, flexible and easy-to-use ingestion path with the launch of the SUPER semistructured type. The SUPER type can contain schemaless, nested data. Technically, a value of SUPER type generally consists of Redshift string and number scalars, arrays and structs (also known as tuples and objects). No schema is imposed on the value. For example, it may be an array whose first element is a string, while its second element is a double.

A first use case for SUPER is the low latency and flexible insertion of JSON data. A Redshift INSERT supports the rapid parsing of JSON and storing it as a SUPER value. These insert transactions can operate up to five times faster than performing the same insertions into tables that have shredded the attributes of SUPER into conventional columns. For example, suppose that the incoming JSON is of the form {"a":.., "b":.., ...}. The user can accelerate the insert performance by storing the incoming JSON into a table TJ with a single SUPER column S, instead of storing it into a conventional table TR with columns 'a', 'b' and so on. When there are hundreds of attributes in the JSON, the performance advantage of SUPER data type becomes substantial, since write amplification is avoided.

Also, the SUPER data type does not require a regular schema and thus it takes no effort to bring in schemaless data for ELT processing: The user need not introspect and clean up the incoming JSON before storing it. For instance, suppose an incoming JSON contains an attribute "c" that has string values and numeric values. If "c" was declared as either varchar or decimal, data ingestion would fail. In contrast, using the SUPER data type, all JSON data is stored during ingestion without loss of information. Later, the user can utilize the PartiQL extension of SQL to analyze the information.

After the user has stored the semistructured data into a SUPER data value, they can query it without imposing a schema. Redshift's dynamic typing provides the capability to discover deeply nested data without the need to impose a schema before querying. Dynamic typing enables filtering, join and aggregation even if their arguments do not have a uniform, single type.

Finally, after schemaless and semistructured data land into SUPER, the user can create PartiQL materialized views to introspect the data and shred them into materialized views. Redshift PartiQL does not fail when functions are fed arguments of the wrong type; it merely nullifies the result of the particular function invocation. This aligns with its role as the language for cleaning up the semistructured data. The materialized views with the shredded data lend performance and usability advantages to the classic analytics cases. When performing analytics on the shredded data, the columnar organization of Amazon Redshift materialized views provides better performance. Furthermore, users and business intelligence (BI) tools that require a conventional schema for ingested data can use views (either materialized or virtual) as the conventional schema presentation of the data.

6.5 Redshift with Lambda

Redshift supports the creation of user-defined functions (UDFs) that are backed by AWS Lambda code. This allows customers to integrate with external components outside of Redshift and enables use cases like i) data enrichment from external data stores or external APIs, ii) data masking and tokenization with external providers, iii) migrating legacy UDFs written in C, C++ or Java. Redshift Lambda UDFs are designed to perform efficiently and securely. Each data slice in the Redshift cluster batches the relevant tuples and invokes Lambda function in parallel. The data transfer happens over a separate isolated network, inaccessible by clients.

7 CONCLUSION

In summary, Amazon Redshift has consistently grown its industry-leading performance and scalability with multiple innovations, such as Managed Storage, Concurrency Scaling, Data Sharing and AQUA. At the same time, Amazon Redshift has grown on ease-of-use: Automated workload management, automated table optimization and automated query rewriting using materialized views enable superior out-of-the-box query performance. Furthermore, Redshift can now interface with additional data (semistructured, spatial) and multiple purpose-built AWS services.

With a differentiating execution core, ability to scale to tens of PBs of data and thousands of concurrent users, ML-based automations that make it easy to use, and tight integration with the wide AWS ecosystem, Amazon Redshift is a best-of-class solution for cloud data warehousing; the innovation on the product continues at accelerated pace.

ACKNOWLEDGEMENTS

A product is deeply shaped by its customers and the people that work on it. We thank the tens of thousands of Redshift customers whose continuous feedback and high standards guided our innovations to solving problems that matter. We have been fortunate to have a great customer base with a relentless demand for an ever improving product.

We have been lucky to have an amazing team working with us on this journey. We thank Charlie Bell, Andy Caldwell, Vuk Ercegovac, Martin Grund, Raju Gulabani, Anurag Gupta, James Hamilton, Ryan Johnson, Menelaos Karavelas, Eugene Kawamoto, Sriram Krishnamurthy, Bruce McGaughy, Yannis Papakonstantinou, Athanasios Papathanasiou, Michalis Petropoulos, Vijayan Prabhakaran, Entong Shen, Vidhya Srinivasan, Stefano Stefani, KamCheung Ting, John Tobler and the entire Redshift team for the impactful moments we had together during the course of this evolution. Nate Binkert and Britt Johnson, your words of wisdom continue to guide us.

REFERENCES

[1] Cloud data warehouse benchmark. https://github.com/awslabs/amazon-redshift-utils/tree/master/src/CloudDataWarehouseBenchmark/Cloud-DWB-Derived-from-TPCDS. Accessed: 2021-11-22.

[2] Partiql query language. https://partiql.org/. Accessed: 2021-11-22.

[3] J. Aguilar-Saborit, R. Ramakrishnan, K. Srinivasan, K. Bocksrocker, I. Alagiannis, M. Sankara, M. Shafiei, J. Blakeley, G. Dasarathy, S. Dash, L. Davidovic, M. Damjanic, S. Djunic, N. Djurkic, C. Feddersen, C. Galindo-Legaria, A. Halverson, M. Kovacevic, N. Kicovic, G. Lukic, D. Maksimovic, A. Manic, N. Markovic, B. Mihic, U. Milic, M. Milojevic, T. Nayak, M. Potocnik, M. Radic, B. Radivojevic, S. Rangarajan, M. Ruzic, M. Simic, M. Sosic, I. Stanko, M. Stikic, S. Stanojkov, V. Stefanovic, M. Sukovic, A. Tomic, D. Tomic, S. Toscano, D. Trifunovic, V. Vasic, T. Verona, A. Vujic, N. Vujic, M. Vukovic, and M. Zivanovic. POLARIS: The distributed SQL engine in Azure Synapse. *PVLDB*, 13(12), 2020.

[4] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Luszczak, M. Switakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. Delta Lake: High-performance ACID table storage over cloud object stores. *PVLDB*, 13(12), 2020.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 1970.

[7] N. Boric, H. Gildhoff, M. Karavelas, I. Pandis, and I. Tsalouchidou. Unified spatial analytics from heterogeneous sources with Amazon Redshift. In *SIGMOD*, 2020.

[8] M. Cai, M. Grund, A. Gupta, F. Nagel, I. Pandis, Y. Papakonstantinou, and M. Petropoulos. Integrated querying of SQL database data and S3 data in Amazon Redshift. *IEEE Data Eng. Bull.*, 41(2), 2018.

[9] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake Elastic Data Warehouse. In *SIGMOD*, 2016.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[11] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1), 1994.

[12] R. Greer. Daytona and the fourth-generation language cymbal. In *SIGMOD*, 1999.

[13] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *SIGMOD*, 2015.

[14] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' Heel of cloud-scale systems. In *HotOS*, 2017.

[15] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[16] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[17] S. Palkar, J. J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. P. Amarasinghe, S. Madden, and M. Zaharia. Evaluating end-to-end optimization for data analytics applications in Weld. *PVLDB*, 11(9), 2018.

[18] P. Parchas, Y. Naamad, P. V. Bouwel, C. Faloutsos, and M. Petropoulos. Fast and effective distribution-key recommendation for Amazon Redshift. *PVLDB*, 13(11), 2020.

[19] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *PVLDB*, 5(12), 2012.

[20] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11), 2013.

[21] K. Sato. An inside look at Google BigQuery. Technical report, Google. https://cloud.google.com/files/BigQueryTechnicalWP.pdf.

[22] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD*, 2017.

[23] T. Wang, R. Johnson, A. Fekete, and I. Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal*, 26(4), 2017.