Eon Mode: Bringing the Vertica Columnar Database to the Cloud

Ben Vandiver
Vertica
ben.vandiver@microfocus.com

Shreya Prasad
Vertica
shreya.prasad@microfocus.com

Pratibha Rana
Vertica
pratibha.rana@microfocus.com

Eden Zik
Vertica
eden.zik@microfocus.com

Amin Saeidi
Vertica
amin.saeidi@microfocus.com

Pratyush Parimal
Vertica
pratyush.parimal@microfocus.com

Styliani Pantela
Vertica
styliani.pantela@microfocus.com

Jaimin Dave
Vertica
jaimin.m.dave@microfocus.com

ABSTRACT

The Vertica Analytic Database is a powerful tool for high performance, large scale SQL analytics. Historically, Vertica has managed direct-attached disk for performance and reliability, at a cost of product complexity and scalability. Eon mode is a new architecture for Vertica that places the data on a reliable shared storage, matching the original architecture's performance on existing workloads and supporting new workloads. While the design reuses Vertica's optimizer and execution engine, the metadata, storage, and fault tolerance mechanisms are re-architected to enable and take advantage of shared storage. Running on Amazon EC2 compute and S3 storage, Eon mode demonstrates good performance, superior scalability, and robust operational behavior. With these improvements, Vertica delivers on the promise of cloud economics, consuming only the compute and storage resources needed, while supporting efficient elasticity.

KEYWORDS

Databases, Shared Storage, Cloud, Column Stores

ACM Reference Format:
Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon Mode: Bringing the Vertica Columnar Database to the Cloud. In *Proceedings of ACM SIGMOD/PODS International Conferences on Management of Data (SIGMOD2018).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Vertica database has historically been deployed on-premises with direct attached disk for maximal bandwidth to achieve high query performance. Vertica's simple install and software-only approach has supported cloud deployment as far back as 2009 for Amazon's cloud. As reliable distributed shared storage becomes prevalent and cloud technologies enable consumption-based resource usage, Vertica must adapt to a model that supports the customers of the future. Vertica is not the only database making this pivot: many new and existing databases make the jump to become cloud databases, instead of merely database that run in the cloud. Existing databases often lack the core architecture that matches the cloud, while new query engines lack performance to effectively compete with Vertica.

The huge data volumes produced by modern enterprises require a distributed system comprised of many individual nodes to achieve sufficient performance. A relational database system must be capable of high performance queries, including joins which relate data items in one table to another table. Database joins in a distributed system are dramatically more efficient with pre-placement of the data. With the advent of the cheap shared storage systems that provide near infinite durable and highly available storage, a database employing shared storage as its backing store can address an enterprise domain worth of data, while shedding much responsibility for durability [17]. A key property is elasticity, which allows resource cost to follow consumption. High performance queries on top of a durable shared storage at an appropriate price point is a key business need.

Vertica's new Eon mode integrates a sharding mechanism into Vertica's existing architecture to achieve both elasticity and good query performance. The system is configured with a number of segment shards, where each segment is responsible for a region of a hash space. Each data record's key is hashed and the record is associated with the segment that owns that region of the hash space. Data load splits the data according to the segments and writes the component pieces to a shared storage. A many-to-many mapping from nodes to segment shards indicates which nodes can serve which segments. To complete the picture, each node maintains a cache of recently used data, where the relatively static mapping of nodes to segments ensures that each node's cache keeps a reasonably static subset of the data.

Vertica's existing architecture, referred to as "Enterprise mode" hereafter, provides the base on which Eon mode is built. We contrast the new Eon mode with Enterprise to show our design changes and relate achievements to available behavior. The sharding model supports improvements to Vertica's scalability and operational behavior. Adding additional nodes provides more nodes on which

SIGMOD2018, June 2018, Houston, Texas USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

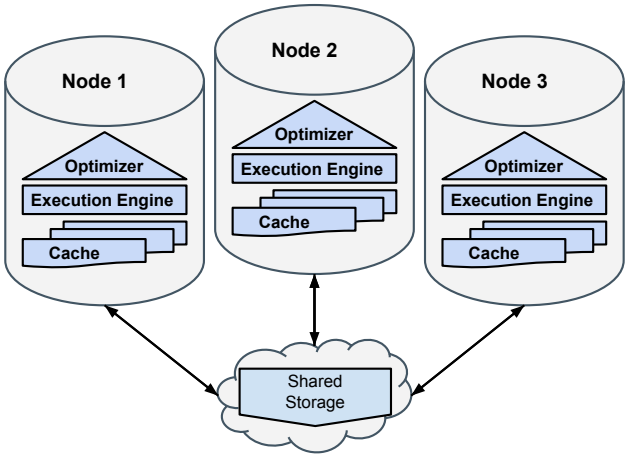


Figure 1: Eon Architecture

to run queries, improving throughput. When nodes go down and recover, they need only fetch an updated copy of the metadata for the shards to which they subscribe and optionally warm their cache from a peer. Many operations in Vertica which were challenging in Enterprise mode with a node down are simple in Eon mode because shards are never down.

In this paper, we present an overview of Vertica to establish context in Section 2, introduce a sharding mechanism for metadata and data in Section 3, and discuss query execution in Section 4. We detail interaction with shared storage in Section 5, articulate advantages in operational behavior in Section 6, comment on implementation in Section 7, and demonstrate performance in Section 8. Finally, we relate Eon mode to existing work in Section 9 and conclude in Section 10.

2 VERTICA OVERVIEW

The core Vertica architecture has been presented before [11], but an overview is provided here for context. Vertica is a column oriented [1, 16] relational SQL database for analytics built on a distributed shared-nothing commodity node platform. Vertica supports the standard SQL declarative query language along with its own proprietary extensions. Vertica's extensions are designed for cases where easily querying timeseries and log style data in SQL was overly cumbersome or impossible. A machine learning package supports high-performance in-database machine learning at scale. Users submit SQL queries using an interactive `vsql` command prompt or via standard JDBC, ODBC, ADO .NET, or Python drivers. Vertica also supports an SDK [18] with hooks for users to extend various parts of the execution engine and bulk load process.

2.1 Physical Design

Vertica supports a variety of mechanisms for improving query performance through good physical design. Vertica physically organizes table data into *projections*, which are sorted, distributed subsets of the attributes of a table. Any number of projections with different sort orders, distributions, and subsets of the table columns are allowed. Because Vertica is a column store and has been optimized for performance, it is not required to have one projection for each predicate present in the query workload. In practice, most customers have one to four projections. Vertica has a Database Designer utility that uses the schema, some sample data, and queries from the workload to automatically determine an optimized set of projections.

Each projection has a specific sort order on which the data is totally sorted as shown in Figure 2. Projections may be thought of as a restricted form of materialized view [2, 15]. They differ from standard materialized views because they are the only physical data structure in Vertica, rather than auxiliary indexes. Sorted data usually results in better compression and thus better I/O performance. Vertica's execution engine can operate directly on encoded data, effectively compressing CPU cycles as well.

In addition to vanilla projections, Vertica supports Live Aggregate Projections which trade-off the ability to maintain precomputed partial aggregate expressions against restrictions on how the base table can be updated. Live aggregates can be used to dramatically speed up query performance for a variety of aggregation, top-K, and distinct operations. Live aggregate projections can even be built with user-defined transform functions supplied by the user via the SDK.

At the table level, Vertica supports partitioning the data horizontally, usually by time. Partitioning the data allows for quick file pruning operation when query predicates align with the partitioning expression. For example, partitioning a table by day (*e.g.*, `extract('day' from event_timestamp)`) stores the data such that any given file will contain data from only one day; queries with a predicate on the recent week like `event_timestamp > now() - interval '7 days'` can easily exclude files from older days. Vertica accomplishes this by tracking minimum and maximum values of columns in each storage and using expression analysis to determine if a predicate could ever be true for the given minimum and maximum. Lastly, Vertica supports a mechanism called Flattened Tables that performs arbitrary denormalization using joins at load time while also providing a refresh mechanism for updating the denormalized table columns when the joined dimension table changes.

2.2 Segmentation: Cluster Distribution

Vertica has a distributed storage system that assigns tuples to specific computation nodes. We call this inter-node (splitting tuples among nodes) horizontal partitioning *segmentation* to distinguish it from intra-node (segregating tuples within nodes) partitioning. Segmentation is specified for each projection, which can be (and most often is) different from the sort order. Projection segmentation provides a deterministic mapping of tuple value to node and thus enables many important optimizations. For example, Vertica uses segmentation to perform fully local distributed joins and efficient distributed aggregations, which is particularly effective for the computation of high-cardinality distinct aggregates

Projections can either be *replicated* or *segmented* on the cluster nodes. As the name implies, a replicated projection stores a copy of each tuple on every projection node. Segmented projections store each tuple on exactly one specific projection node. The node on

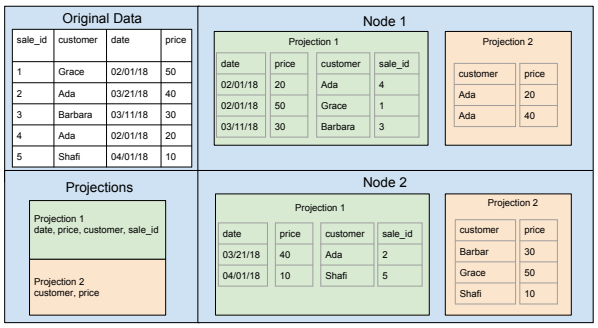


Figure 2: Relationship between tables and projections. The *sales* tables has 2 projections: (1) An all-columns projection, sorted by date, segmented by $HASH(sale\_id)$ and (2) Another projection containing only $(cust, price)$ attributes, sorted by *cust*, segmented by $HASH(cust)$.

which the tuple is stored is determined by a segmentation clause in the projection definition: `CREATE PROJECTION ... SEGMENTED BY HASH(<columns>)` where `<columns>` is an arbitrary list of columns from the projection. A set of one or more columns with high cardinality and relatively even value distribution performs well. Contiguous regions of the hash space are mapped to nodes in the cluster; any tuple whose columns hash to a region will be stored and read from that node. To support fault tolerance, a second "buddy" projection is created that shares the same segmentation expression, but each hash space region is mapped to a different node. Typically, the nodes are conceptually arranged in a logical ring, which is rotated to determine the layout of the buddy, resulting in a layout where adjacent nodes in the ring serve as replicas. When a node in the base projection is down, the optimizer will source the missing data from the appropriate node in the buddy projection.

2.3 Storage

Vertica has a Read Optimized Store (ROS) and a Write Optimized Store (WOS). Data in the ROS is physically stored in multiple *ROS containers* on a standard file system. Each ROS container logically contains some number of complete tuples sorted by the projection's sort order, stored per column. Vertica is a true column store – column data may be independently retrieved as the storage is physically separate. Vertica writes actual column data, followed by a footer with a position index. The position index maps tuple offset in the container to a block in the file, along with block metadata such as minimum value and maximum value to accelerate the execution engine. If the column data is small, Vertica concatenates multiple column files together to reduce the overall file count. Complete tuples are reconstructed by fetching values with the same position from each column file within a ROS container. Once written, ROS files are never modified.

Data in the WOS is solely in memory, where column or row orientation doesn't matter. The WOS's primary purpose is to buffer small data inserts, deletes and updates so that writes to physical structures contain a sufficient numbers of rows to amortize the cost of the writing. Data is not encoded or compressed when it is in the WOS but it is segmented according to the projection's segmentation expression.

The Tuple Mover is a service that performs compactions of the storage using Moveouts and Mergeouts. The tuple mover runs independently on each node as each node's storage and memory situation may vary. Moveout is the operation that converts WOS to ROS, sorting the data and writing it to disk from the in-memory WOS. Mergeout is the operation that compacts ROS containers by merging two or more containers to make a single new container. The input containers are dropped at the end of the mergeout transaction. Mergeout uses an exponentially tiered strata algorithm to select ROS containers to merge so as to only merge each tuple a small fixed number of times. Mergeout may run more aggressively to keep the ROS container count down to constrain metadata size and avoid expensive large fan-in merge operations in the execution engine when a fully sorted stream of tuples is required.

Deletes and updates are implemented with a tombstone-like mechanism called a delete vector that stores the positions of tuples that have been deleted. Delete vectors are additional storage objects created when tuples are deleted and stored using the same format as regular columns. An update is modeled as a delete followed by an insert. Deleted data is purged during mergeout and the number of deleted records on a storage is a factor in its selection for mergeout.

2.4 Catalog Architecture

The Vertica catalog stores and provides access to the metadata of the database. Other databases typically use their own table structures and B-trees for their metadata maintenance. However, Vertica uses a custom mechanism due to its table structures being optimized for billions of rows. In-memory, the catalog uses a multi-version concurrency control mechanism, exposing consistent snapshots to database read operations and copy-on-write semantics for write operations. Transaction commit results in transaction logs appended to a redo log. Transaction logs contain only metadata as the data files are written prior to commit. Transaction logs are broken into multiple files but totally ordered with an incrementing version counter. When the total transaction log size exceeds a threshold, the catalog writes out a checkpoint which reflects the current state of all objects at the time the checkpoint was written. The checkpoint is labeled with the version counter, ensuring that the checkpoint can be ordered relative to the transaction logs. Vertica retains two checkpoints, any prior checkpoints and transaction logs can be deleted. At startup time, the catalog reads the most recent valid checkpoint, then applies any subsequent transaction logs to arrive at the most up to date catalog state.

3 SHARDING

Eon mode introduces a sharding mechanism for metadata management in a distributed system with shared storage.

3.1 Shards and Subscriptions

The catalog is divided into global objects (like tables and users) which are in every node's catalog, and storage objects each of which only a subset of the nodes will serve. In Enterprise, the storage objects are persisted in a node-specific catalog that is managed independently for each node. Each node independently loads and

compacts data so the storage container organization will not match between nodes, even for the "buddy projections" used for replication. Since each node has private disk, synchronization between nodes about storage is unnecessary. However, in Eon mode, the storage is written to a shared storage and is potentially accessible by any node.

Sharding is a new mechanism that guarantees each node can track a subset of the overall storage metadata, all nodes see a consistent view, and the metadata aligns with Vertica's existing projection mechanism to ensure comparable performance. Rather than implicit regions of hash space defined by individual projections, Eon mode explicitly has *segment shards* that logically contain any metadata object referring to storage of tuples that hash to a specific region (see Figure 3). All storage metadata for a segmented projection is associated with segment shards. The number of segment shards is fixed at database creation. Replicated projections have their storage metadata associated with a *replica shard*.

A node that is *subscribed* to a shard will store and serve the metadata and data associated with the shard. Node subscriptions control how processing is distributed across the cluster and can be created or removed while the cluster is running. A node usually subscribes to more than one shard and shards normally have multiple subscribers. For cluster availability, there must be at least one subscribing node for each shard. To enable node fault tolerance, there must be more than one subscriber to each shard.

3.2 Transaction processing

When a transaction commits, any storage metadata associated with a shard must have been sent to every subscriber of the shard. Nodes can each create metadata as part of a transaction. For example, a bulk load could create ROS containers on every node in the cluster. Vertica eagerly redistributes metadata within a transaction to better handle node failures that occur prior to commit. The shard metadata deltas are piggybacked on existing messages where possible to avoid additional message overhead. At commit time, the transaction validates the invariant that all nodes have metadata for all their subscribed shards, confirming that no additional subscription has "snuck in" to invalidate the transaction. If the invariant doesn't hold, the transaction rolls back.

3.3 Subscription Process

Vertica executes a sequence of metadata and data operations to subscribe a node to a shard (as shown in Figure 4). A node indicates it wishes to subscribe by creating a subscription in the `PENDING` state. A subscription service wakes up, picks a source node that already subscribes to the shard, and transfers metadata to bring the new subscriber up to date. The transfer process proceeds in a succession of rounds, transferring checkpoint and/or transaction logs from source to destination until the node is caught up. If the missing metadata is sufficiently small, the service takes a lock that blocks transaction commit, transfers the remainder logs, marks the subscription to `PASSIVE`, and commits. Once in the `PASSIVE` state, the node can participate in commits and could be promoted to `ACTIVE` if all other subscribers fail. A cache warming service wakes up, picks a source node that already subscribes to the shard, and warms the cache through a process described in Section 5.2.

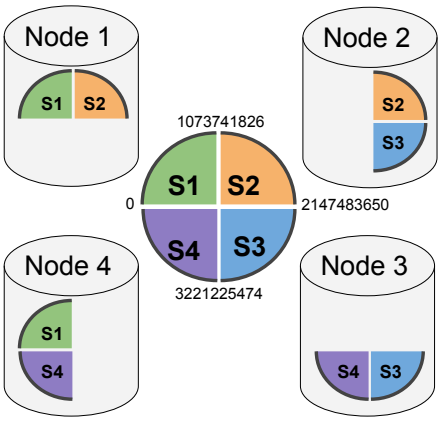


Figure 3: 32-bit hash space is segmented into four shards S1, S2, S3, S4. Each node subscribes to a subset of shards.

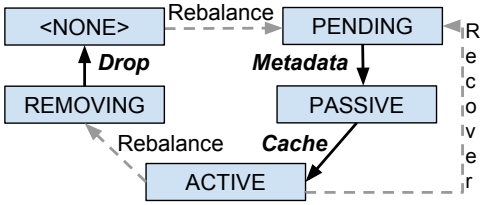


Figure 4: State transitions for a shard subscription. Solid black arrows denote data and metadata operations, whereas the other arrows are proactive or reaction organizational changes.

Once the cache is warm, the subscription transitions to the `ACTIVE` state and begins serving queries. Not all new subscribers will care about cache warming and thus will skip directly from `PASSIVE` to `ACTIVE`.

The subscription process also handles node down and recovery. When a node goes down and recovers, it returns with subscriptions that are stale. Upon invitation back into the cluster, a transaction is committed that transitions all of the `ACTIVE` subscriptions for the recovering node to `PENDING`, effectively forcing a re-subscription. The re-subscription process proceeds similarly to subscription, except the metadata transfer and cache warm steps can be incremental. Upon completion, the recovered node's subscriptions are once again `ACTIVE` and the node will begin to serve queries.

When a node unsubscribes from a shard, it also follows a collection of steps. First, the subscription transitions to `REMOVING` to declare the intent to remove the subscription. However, the subscription cannot be dropped until sufficient other subscribers exist to ensure the shard remains fault tolerant. For example, in scenario where only one node subscribes to a shard, moving a subscription between two nodes will require subscription to occur before the existing subscription can be dropped. While in the `REMOVING` state, the node continues to serve queries. Once a sufficient number of

other subscribers exist in the ACTIVE state, the node drops the relevant metadata for the shard, purges the associated data from the cache, and drops the subscription.

3.4 Cluster Invariants

For an Eon mode cluster to be viable, at least one node must have an ACTIVE subscription for each shard. Furthermore, each ACTIVE subscriber must have an identical view of the shard and all nodes must have an identical view of the global catalog objects. For simplicity, the catalog maintains a global catalog version number which increments with each transaction commit. To form a cluster, Vertica needs a quorum of nodes, all the shards to be represented by nodes with subscriptions that were ACTIVE when the nodes went down, and that each contributory node has the same (highest) global catalog version. Nodes whose version is behind will be repaired after the cluster forms with the re-subscription process described above. If the cluster cannot agree on a highest version between nodes with the appropriate subscriptions, a truncation mechanism discards transactions, rewinding until a consistent version can be established on the cluster. If sufficient nodes fail such that the constraints are violated during cluster operation, the cluster will shutdown automatically to avoid divergence or wrong answers.

3.5 Revive

Durability guarantees in Eon are stronger than a traditional Vertica deployment, leveraging shared storage as a durable but slow persistence layer for both data and metadata. While all data is uploaded to shared storage before commit, metadata is persisted asynchronously to avoid sacrificing commit performance. Each node writes transaction logs to local storage, then independently uploads them to shared storage on a regular, configurable interval. During a normal shutdown, any remaining logs are uploaded to ensure shared storage has a complete record. Process termination results in reading the local transaction logs and no loss of transactions. Individual instance loss results in rebuilding metadata from a peer and no loss of transactions. Catastrophic loss of many instances requires constructing a consistent version from the uploaded logs, where each node may or may not have uploaded logs for a particular transaction. The operation that starts a cluster from shared storage is called *revive*. Paired with the catalog upload or sync operation, they have the following objectives: discard as few transactions as practical, restore to a consistent snapshot for which all the data and metadata files are present, and ensure the version sequence of the catalog is consistent even through revive operations that lose transactions.

Rather than burden the revive operation with the responsibility of selecting the transactions to discard, a running cluster regularly updates the *truncation version* to reflect the durability point. Each node maintains a *sync interval* that reflects the range of versions to which it could successfully revive based on uploaded checkpoints and transaction logs. An elected leader writes down a consensus *truncation version* that is the set cover of the metadata with respect to each shard as shown in Figure 5. The truncation version is the minimum across shards of the upper bound of sync interval for each subscribing node. The consensus version serves as a "high watermark" for all cluster metadata - a version consistent with respect to all shards, to which the cluster can be revived. Nodes uploading transactions increase the upper bound of the sync interval and deleting stale checkpoints increases the lower bound of the sync interval. Deleting checkpoints and transaction logs after the truncation version is not allowed. Once the truncation version is computed, it is persisted to a file in shared storage called `cluster_info.json`. In addition to the truncation version, the file also contains a timestamp, node and database information, and a lease time.

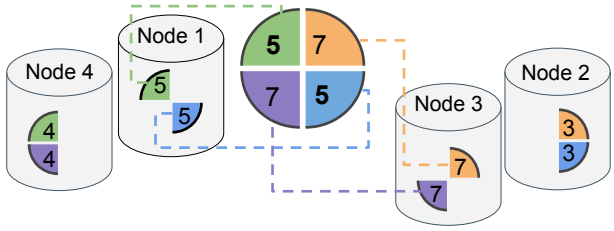


Figure 5: Computing a consensuses version of a 4 node, 4 shard cluster with respect to each shard. In this example, consensus version is 5.

Revive is the process by which a cluster is started from shared storage and it occurs in several stages. First, the nodes are commissioned with empty local storage. Next, all nodes individually download their the catalog from shared storage. All nodes then read the `cluster_info.json` file from shared storage, and extract the truncation version and lease time. If the lease time has not yet expired, revive aborts since it is likely that another cluster is already running on the shared storage location. Each node reads its catalog, truncates all commits subsequent to the truncation version, and writes a new checkpoint. Finally, the cluster starts at the new version.

The revive mechanism is augmented with an *incarnation id* to ensure each revive operation is atomic and to avoid duplication in the version space. After truncation, the cluster can commit a version with the same version number as prior to truncation but with different contents. The *incarnation ID* is a 128 bit UUID [12] which changes each time the cluster is revived. Metadata files uploaded to shared storage are qualified with the incarnation id, ensuring that each revived cluster writes to a distinct location. When sufficient metadata has been uploaded from the newly revived cluster, a version of the `cluster_info.json` file is uploaded with the new incarnation id of the cluster. A subsequent revive reads the file to determine which incarnation it is reviving from, effectively making the write of the `cluster_info.json` the commit point for revive.

Cluster formation reuses the revive mechanism when the cluster crashes mid commit and some nodes restart with different catalog versions. The cluster former notices the discrepancy based on invite messages and instructs the cluster to perform a truncation operation to the best catalog version. The cluster follows the same mechanism as revive, moving to a new incarnation id, and eventually uploading a new `cluster_info.json` file.

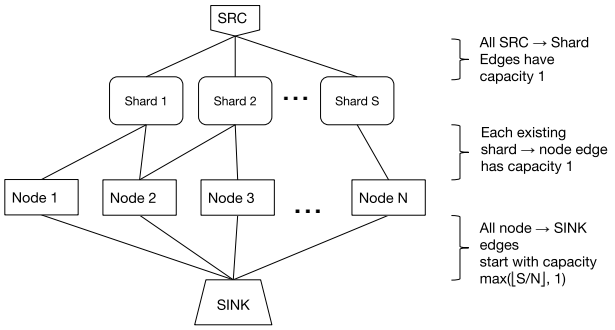


Figure 6: Graph expressing constraints whose max-flow describes an assignment of subscribing nodes to shards.

4 QUERY EXECUTION

Vertica uses a slightly different process to plan queries in Eon mode to incorporate the sharding mechanism and remote storage. Instead of using a fixed segmentation distribution of the hash space to each node, Eon mode uses the sharding mechanism to dynamically select a covering set of subscriptions over the shard space to serve a query. When the Vertica optimizer selects a projection, the layout for the projection is determined by the participating subscriptions for the session as described in Section 4.1. Eon runs Vertica's standard cost-based distributed optimizer, generating query plans equivalent to Enterprise mode. Only nodes that the session selects to serve a shard participate in query execution. When an executor node receives a query plan, it attaches storage for the shards the session has instructed for it to serve. Storage containers are partitioned by shard: each contains rows whose hash values map to a single shard's hash range.

By segmenting the data by key value, operations such as join and group-by that need all records with a particular key find them on the same node. For example, a query that joins table T1 on column "a" with T2 on column "b" can be executed without a reshuffle if T1 is segmented by column "a" and T2 by column "b" identical values will be hashed to same value, be stored in the same shard, and served by the same node. Similarly if T1 is segmented by column "a", a query that groups by column "a" does not need a reshuffle to compute the value of each "a" group. For predictable queries, such as those run by dashboarding applications, proper segmentation choices for each table can result in fast query performance that avoids any network bottlenecks.

4.1 Participating Subscription Selection

We model the shard to node allocation problem as a graph flow problem by carefully constructing a graph that reflects the configuration of the database (shown in Figure 6). The graph has a source vertex which has an edge to each shard vertex. Each shard vertex has an edge to a node vertex if the node can serve that shard. Finally, each node vertex has an edge to the sink vertex. In this way, the graph encodes the constraints for which mappings are possible. A max flow on the graph will route flow over the shard to node edges, where the used edges indicate the mapping the process has selected for the query.

A balanced distribution of shards to nodes is obtained through adjusting the capacity of the edges in the graph. The edges from source to shard all have capacity 1 as the solution must involve all shards. This establishes the desired max flow as the number of shards. Edges between shard and node vertices also have capacity 1, consuming all flow from a shard vertex. Edges from node vertices to the sink vertex begin with capacity $max(\frac{S}{N}, 1)$. By assigning even outflow to each node vertex, a max flow will push flow evenly over the shard to node edges, resulting in a balanced distribution.

The graph can be very asymmetric if the node to shard edges are unbalanced, leading to a max flow that is less than the number of shards, and resulting in an incomplete assignment of shards to nodes. For example, with N nodes and S shards, if only one node serves every shard, then the graph will assign only one shard mapping when run. We address this issue by running successive rounds of max flow, leaving the existing flow intact while incrementally increasing the capacity of the node vertex to SINK edges. When the flow finally reaches the number of shards, all the shards have been assigned with minimal skew.

The max flow algorithm is typically deterministic: given the same input graph it will generate the same shard to node mapping. To promote even usage of each shard to node mapping, we vary the order the graph edges are created, so as to vary the output. The result is a more even distribution of nodes selected to serve shards, increasing query throughput because the same nodes are not "full" serving the same shards for all queries.

Additionally, we can prioritize some nodes over others by incrementally adding the edges from node vertices to the SINK. The graph starts with edges only from priority nodes to the SINK. If max flow does not successfully deliver all potential flow to the SINK, add the next set of edges from lower priority node vertices to the SINK and re-run the max flow algorithm. For example, the starting graph includes only nodes on the same physical rack, encouraging an assignment that avoids sending network data across bandwidth-constrained links.

When the system decides which nodes will be capable of serving which shards, we intentionally distribute shards to nodes such that subgroups of nodes can serve all shards. For example, the set of nodes on each physical rack are initialized to be capable of serving the full set of shards. Thus we can partition the cluster into distinct non-overlapping subsets and enforce separation by using the prioritization mechanism described above.

4.2 Elastic Throughput Scaling

Duplicating responsibility for each segment across multiple nodes improves query throughput. By running each query on the subset of the nodes, adding additional nodes results in more concurrent queries on the system as a whole. A simple case is where there are twice as many nodes as segments, effectively producing two clusters which can run queries independently. Even with non-integral multiples of the segment count of nodes, linear scale-out can be achieved when each node can concurrently run queries equal to the number of shards. For a database with S shards, N nodes, and E execution slots per node, a running query requires S of the total $N * E$ slots. If $S < E$, then adding individual nodes will result in linear scale-out performance, otherwise batches of nodes will be

required and performance improvement will look more like a step function. A proper load balancing mechanism as described above is necessary to distribute the workload over the cluster. See Section 8 for performance results that demonstrate scalability.

4.3 Subcluster Workload Isolation

As alluded to above, the subscription mechanism can be employed to run workloads on specific subsets of the cluster nodes. The administrator can designate a collection of nodes as a subcluster and the subscription rebalance mechanism will ensure that every shard has a node subscriber in the subcluster. When a client connects to a node in the subcluster, nodes in the subcluster are prioritized by the participating nodes algorithm, resulting in queries that execute on just those nodes. The workload does not escape to include any node from the rest of the cluster unless there have been sufficient node failures within the subcluster to require outside assistance to serve queries. Strong node-based workload isolation improves support for multi-tenancy and can be used to insulate query from data load or finely-tuned reporting from ad-hoc queries.

4.4 Crunch Scaling

While Elastic Throughput Scaling is effective at increasing throughput, it does not improve the running time of an individual query. Workloads can contain a mix of short requests and longer requests that would benefit from additional computational power. The simplest mechanism is to run with more shards than nodes; elastically adding nodes will spread the shards out across more nodes and distribute query processing workload. When the node count exceeds the shard count, a new mechanism is required. Increasing the shard count requires splitting a segment shard, an expensive operation since all the data must be split and rewritten. Instead, two or more nodes can collectively serve a segment shard for the same query by applying a new hash segmentation predicate to each row as it is read to determine which node will process the row. By applying selective predicates first, the hashing burden can be reduced, but in the worst case each node reads the entire data-set for the shard.

Alternatively, the containers can be physically split between the nodes resulting in good I/O performance at the cost of skew vulnerability and loss of the segmentation property. Each node sharing a segment scans a distinct subset of the containers, or regions within a container. If the query has a selective predicate, a lucky node could quickly filter all the rows from the container(s) assigned to it, leaving another node with all the work. The scenario is more likely when a node has part of the sorted container and the query containers a predicate on the sort column(s). A work-stealing mechanism would mitigate the scenario. More importantly, the data is no longer segmented such that a node has all the rows whose segmentation columns match. Local joins and aggregates are no longer possible, the data must be shuffled within the nodes sharing a shard. With container split, each row is read once across the cluster, but the processing overhead is higher. Choosing between hash filter and container split depends on the query, making it a likely candidate for using Vertica's cost-based optimizer.

4.5 Data Load and Data Definition operations

Similarly to queries, a data modification statement (DML) like `INSERT`, `UPDATE`, `DELETE`, `MERGE`, or data (re)definition operations such as Tuple Mover or partition management operations such as copy, move partitions will run according to the selected mapping of nodes to shards. The plan will execute on the participating nodes, which compute the output data files or delete vectors for each shard. An executor which is responsible for multiple shards will locally split the output data into separate streams for each shard, resulting in storage containers that contain data for exactly one shard. Vertica never modifies existing files, instead creating new files for data or for delete marks. The files are first written to the local disk, then uploaded to shared storage. Replicated projections use just a single participating node as the writer. The metadata for the new files is generated on the participating nodes and then distributed to other subscribing nodes. The commit point for the statement occurs when upload to the shared storage completes. For a committed transaction all the data has been successfully uploaded to shared storage; failure of the nodes involved cannot result in missing files on the shared storage.

While planning these operations or at commit point, if the session sees concurrent subscription changes so that a participating node is no longer subscribed to the shard it wrote the data into, the transaction is rolled back to ensure correctness.

5 STORAGE

Eon relies on a shared storage to persist data and metadata across a cluster, and certain properties are required of such shared storage:

(1) **Durable** - once a write has succeeded, failures in the storage subsystem are highly unlikely to result in data loss.
(2) **Available** - reads and writes are highly likely to succeed, even in the presence of failures in the storage subsystem.
(3) **Globally addressable** - any data can be read from any compute node using the same path.
(4) **Elastic** - capacity increases on demand, to a size limited by purchasing power.

Additionally, shared storage has different properties than local storage:

(1) Latency of read and write access to shared storage is higher than local storage
(2) Remote - compute tasks cannot be scheduled co-resident with data (*e.g.*, S3 or SAN)
(3) Accessing shared storage carries a cost - either in consumption of a limited shared resource or in actual dollars.
(4) Shared storage may lack POSIX features (*e.g.*, file rename or append)

5.1 Data Layout

An Enterprise mode Vertica database writes data to a direct attached disk that is not shared with other nodes. Each node writes files in a separate namespace, thus ensuring no filename collisions between nodes. Vertica employs a two tier directory structure to avoid overloading the filesystem with too many files in a directory. The sam mechanism is used for column data and delete vectors. A simple naming mechanism like using the metadata object identifier (OID)

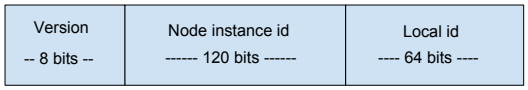


Figure 7: Storage Identifier Format used to construct globally unique filenames for Vertica storage

would be sufficient to uniquely name a file so the execution engine can open it given metadata information. However, operations like backup and restore, replication between clusters, or node recovery benefit from containers whose names are globally unique. Without a globally unique name for a given object, repeated copies between clusters, potentially bidirectional, would require keeping persistent mappings and incur significantly increased complexity.

Vertica adopts an approach that uses a globally unique storage identifier (SID) to identify files. The SID is a combination of the node instance id (120 bit random number generated when the Vertica process starts) and the local id (64 bit catalog OID associated with the storage object when it is created) as shown in Figure 7. The node instance identifier is strongly random (from `/dev/random`) and provides the core uniqueness property, whereas the OID component is a simple counter. Each node can create SIDs without communicating with the other nodes. Tying the node instance id to the lifetime of the Vertica process ensures that for a cluster whose catalog and data are cloned, each of the two subsequent clusters will still generate SIDs that are unique from each other.

In Eon mode, globally unique SIDs ensures that all nodes can write files into a single shared storage namespace without fear of collision. Vertica writes files to a flat namespace without subdividing them by node or table. Storage objects are not owned by any particular node since many nodes can subscribe to a single shard. Vertica supports operations like `copy_table` and `swap_partition` which can reference the same storage in multiple tables, so storage is not tied to a specific table. Determining when a file can be deleted is a complex operation and is discussed in Section 6.5.

Eon mode does not support the WOS; all modification operations are required to persist to disk. With the WOS, data could be lost if nodes crash. Asymmetric memory consumption can cause a node to spill to disk where a peer did not, creating opportunity for node storage to diverge. Most Vertica users do not derive significant benefit from the WOS, but pay a significant penalty in complexity and recovery mechanism. If the existing ROS mechanism is insufficient to real-world low latency write use cases, a new mechanism different from the WOS will be required.

5.2 Cache

Running every query directly against the data in shared storage would result in poor performance and subject the shared storage to heavy load. Vertica Eon mode introduces a cache to avoid reading from the shared storage for frequently used data (See Figure 1).

The cache is a disk cache for caching entire data files from the shared storage. Vertica never modifies storage files once they are written, so the cache only needs to handle add and drop, and never invalidate. The cache eviction policy is a simple least-recently-used

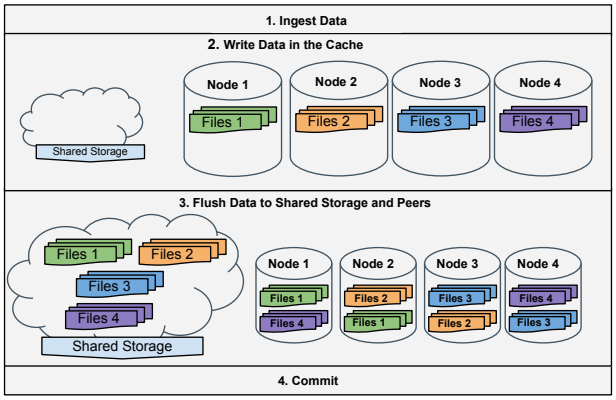


Figure 8: Data Load Workflow. Files are cached by peers and persisted to shared storage prior to commit.

(LRU) mechanism, assuming that past access is a good predictor of future need. LRU has been shown to be an effective page replacement algorithm [7]. Users can express shaping policies like "don't use the cache for this query" or eventually policies like "cache recent partitions of table `T`" or "never cache table `T2`." Shaping policies support mixed workload, ensuring that large batch historical queries do not evict items important to serving low latency dashboard queries. Similarly while loading archive data, write though the cache can be turned off for the same reasons. If needed the cache can be cleared completely.

The cache is write-through since newly added files are likely to be referenced by queries. At load time files are written to the cache, uploaded to the shared storage and sent to all the nodes that subscribe to the shard in which the data is being loaded. This mechanism of sending data to peers at load time results in much better node down performance since the cache of the peers who take over for the down node is already warm. The file compaction mechanism (mergeout) puts its output files into the cache and also uploads them to the shared storage.

When a node subscribes to a shard, it warms up its cache to resemble the cache of its peer. The node attempts to select a peer from the same subcluster, if any, to ensure the cache matches the workload the node will experience. The subscriber supplies the peer with a capacity target and the peer supplies a list of most-recently-used files that fit within the budget. The subscriber can then either fetch the files from shared storage or from the peer itself. Given a reasonable cache size, peer to peer cache warming provides a very similar looking cache on the new node and helps in mitigating any performance hiccups.

5.3 Filesystem Access

Vertica filesystem access by the execution engine is done via an API that provides the abstraction to access filesystems with different internal implementations. The API is dubbed the user-defined filesystem (UDFS) API, despite not being currently released to users. Vertica currently supports three filesystems: POSIX, HDFS, and S3. In theory, any one of these filesystems can serve as a storage for table data, temp data, or metadata. It is the subject of future work to

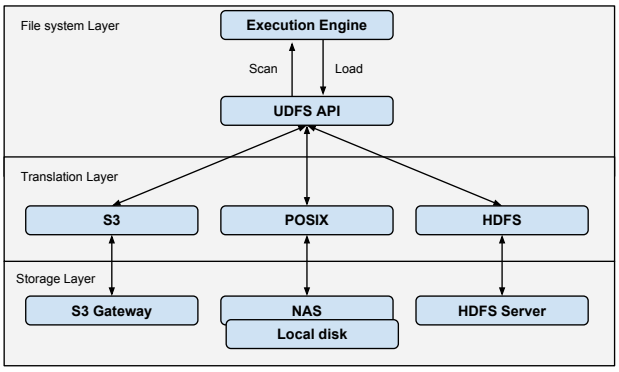


Figure 9: UDFS API Diagram

open this API and let users build their own UDFS implementation to run Eon mode on the shared storage of their choice.

Given Eon mode's focus on the cloud, S3 [4] is a practical choice that meets the desired properties for shared storage. S3 is an object store with several key semantic differences from a POSIX linux filesystem. Objects do not support the full range of POSIX operations (e.g. rename, append). Directories do not exist and path globbing functions differently. Operations that would rarely fail in a real filesystem do fail occasionally on S3. S3 also exposes a different consistency model from a regular filesystem in different situations. S3 requires different tuning to achieve good performance. Finally, integration with S3's security model is critical for a secure deployment.

S3 objects are immutable. That means appending x to an object y is not possible. In those cases, an entirely new object must be created with contents of y and x, which may be costly if the original object is large. While Vertica works on a similar principle of immutable files, the load process itself sometimes opens and closes the files. By staging the files first to the cache, Vertica can work on a more functional filesystem and upload the final artifacts. The load process has been improved and relies less on rename and append, so writes are cheaper when a cache is not present.

Vertica observes broader failures with S3 than with local filesystems. Any filesystem access can (and will) fail. For instance, a write operation could fail because S3 credentials were not set up properly or the permissions changed mid-operation. Sometimes S3 generates internal errors outside of user control. A properly balanced retry loop is required when errors happen or the S3 system throttles access. Users expect their queries to be cancelable, so Vertica cannot hang waiting for S3 to respond.

Another caveat with reading and writing objects to S3 is consistency guarantees that can vary based on use case. For instance, in some cases, one might want to check if a file exists on S3, and create it only if it does not. S3 provides read-after-write consistency for writes to new objects, however, if one checks the existence of a file with a HEAD request before writing, the read-after-write then becomes eventually consistent. Vertica requires strong consistency. To avoid observing eventual consistency issues, Vertica does not check object existence with HEAD requests, and instead uses the "list" API with an object prefix. Overwriting S3 objects is also eventually consistent, but as mentioned above, since Vertica never modifies written objects, the scenario never arises.

Achieving good performance with S3 requires larger request sizes than local disk to better amortize the cost of accessing the service. The directory fan out tree used for storing files requires a hash-based prefix scheme instead of the simpler prefix scheme to avoid hotspotting a single S3 server to read or write recent data. Finally, requests cost money, so minimizing the request amount results in a lower cost to operate.

Vertica pursues a secure by default model, using IAM authentication to avoid storing keys in the database, HTTPS to communicate with S3, and providing support for bucket encryption.

6 OPERATIONAL BEHAVIOR

Eon mode exhibits better operational behavior than Enterprise mode, with improvements in most core components of database operation.

6.1 Node Down and Recovery

Duplicating responsibility for each shard across multiple nodes improves availability. When a node fails, other nodes can immediately serve any segments it was responsible for without needing a repair operation. The subscription assignment mechanism ensures that each shard has at least two subscribers, an analog of Enterprise's K-safety mechanism. Unlike Enterprise which relies on a "buddy projection" mechanism, the global query plan does not change when a node is down, merely a different node serves the underlying data. See Section 8 for performance results that demonstrate improved node down performance.

When a node rejoins the cluster, it must re-subscribe to the shards it previously was subscribed to. Re-subscription is less resource intensive than subscription: the node can fetch incremental shard diffs and cache-warming a lukewarm cache requires transferring fewer files. Failure to resubscribe is a critical failure that probably indicates some issue with the host; the node goes down to ensure visibility to the administrator. When re-subscription is complete, the node is once again a full participant in query processing and client activity. By contrast, Enterprise requires that each table and projection be repaired with an operation that involves table locks on every table in the database. Since the storage layout is not identical between Enterprise node serving as replicas, the data must be logically transferred. In Eon mode, nodes perform a byte-based file copy to warm up the cache of the resubscribing node, instead of an executed query plan in Enterprise. Worst case recovery performance is proportional to the size of the cache in Eon, whereas Enterprise recovery is proportional to the entire data-set stored on an Enterprise node.

6.2 Compaction with the Tuple Mover

Vertica Eon mode uses the tuple mover from Enterprise mode with some modifications. It does not run moveout operation as write-optimized-storage (WOS) is disabled in this mode. However, the mergeout operation is still required to maintain performance as the number of ROS containers grows over time. In Enterprise mode, each node runs mergeout independently and replicated data will be redundantly merged by multiple nodes. In Eon mode, a subscriber is

deemed the *mergeout coordinator* and selects the content of mergeout jobs. A single coordinator is selected to ensure that conflicting mergeout jobs are not executed concurrently. Should the mergeout coordinator for a shard fail, the cluster runs a transaction to select a new coordinator, taking care to keep the workload balanced. The mergeout coordinator can run the mergeout jobs itself, with the job commit informing the other subscribers of the result of the mergeout operation. Alternatively, the coordinator can farm out the jobs to the other subscribers, effectively scaling the mergeout bandwidth with cluster size. The coordinators can also be assigned to specific subcluster, allowing compaction work to be isolated from other workload.

6.3 Schema Evolution

The `ADD Column` DDL typically has a default value that can be either derived or constant. If no default is specified NULL is used as the default. The transaction generates new ROS containers and the relevant ROS metadata. In Enterprise mode, the metadata for the ROS container is a local catalog object and exists within the context of the transaction without being published to other nodes. However, in Eon mode the ROS container is a global object that gets published to other subscribers upon creation and so does the newly added column. Publishing the ROS container requires having published its column specific metadata. Publishing the column specific metadata requires modifying the table object that is associated with it. Publishing that requires holding the global catalog lock. Holding the lock while generating ROS containers increases contention and should be kept to a minimum. That leads to a chicken and egg problem that we solve by updating the Vertica concurrency model.

The new model is Optimistic Concurrency Control (OCC) [10]. Modifications to metadata happen offline and up front without requiring a global catalog lock. Throughout the transaction, a write set is maintained that keeps track of all the global catalog objects that have been modified. Then the ROS containers are generated and published to other subscribers within the transaction context. Only then is the global catalog lock is acquired and the write set is validated. The validation happens by comparing the version tracked in the write set with the latest version of the object. If the versions match the validation succeeds the transaction commits, otherwise it rolls back. The new paradigm leads to optimized concurrency and reduced lock contention.

6.4 Elasticity

The node-to-segment mapping can be rapidly adjusted because all of the data is stored in the shared storage, not directly on the nodes themselves. Nodes can easily be added to the system by adjusting the mapping to allocate a subset of the segments to the new node, potentially removing responsibility for such from some pre-existing nodes. Queries can immediately use the new nodes as no expensive redistribution mechanism over all records is required. Filling a cold cache takes work proportional to the active working set, not the entire dataset that could conceivably be served by the node. Removing a node is as simple as ensuring any segment served by the node be removed is also served by another node.

6.5 Deleting Files

Since files are never modified, the key decision is when to delete a file. The goal is to never delete a file that is still in use but to eventually delete all files that are no longer needed. In Enterprise mode, Vertica maintains a reference count of every file, considering the file for deletion when the count hits zero. The counter tracks both catalog references such as storage containers in a table as well as query references from running queries. In Enterprise mode, each file is owned by a single, so each node is responsible for deleting its own files. In Eon mode, files are not owned by a specific node and hence local reference counting is insufficient and cluster-wide operations are expensive. Since shared storage is typically cheap, Eon mode can afford less expensive algorithms that are less efficient at reclamation. Eon mode employs an augmented reference counting mechanism for online cleanup and a global enumeration mechanism as a fallback. A node may only delete a file when it is subscribed to the shard containing the storage and a quorum of nodes are present.

When the reference count hits zero, an Eon mode database might need to retain the file for two reasons. The first reason is that the query reference count of the file might be non-zero on another node in the cluster, since not all queries run on all nodes. The file can be removed from the node's cache immediately when the local reference count hits zero. Rather than maintain a global query reference count, each node gossips the minimum catalog version of its running queries, taking care to ensure the reported value is monotonically increasing. When the cluster's minimum query version exceeds the version at which the catalog reference count hit zero, the node knows that no query on the cluster references the file and it may be safely deleted. Alternatively, a constant time delay on file delete is a simple mechanism that prevents issues if queries run in less time than the delay.

The second reason a file may need to be preserved past zero reference count is that the catalog transaction containing the storage drop may not have been persisted to shared storage yet. Recall that transaction commit writes to local disk with subsequent asynchronous upload to shared storage, so a loss of all the node local disk can undo a transaction. Files can be deleted when the truncation version passes the drop version.

A file on shared storage can be leaked if the node responsible for handling it crashes mid-operation. For example, a file can be leaked if a node crashes after creating the file but before any other node is informed of the it's existence. Other scenarios involve moving subscriptions between nodes with concurrent node crashes. To clean up leaked files, the nodes aggregate a complete list of referenced files from all node's reference counters, compare with a list of existing files on the shared storage, and delete any unknown files. To handle concurrently created files, the operation ignores storage with a prefix of any currently running node instance id. While expensive, the operation is not common, as it is manually run when nodes crash.

7 IMPLEMENTATION

Eon mode will ship in a Vertica release in the near future and has been in public beta since Vertica 9.0 released in October 2017. Numerous customers have tried the beta and have seen performance

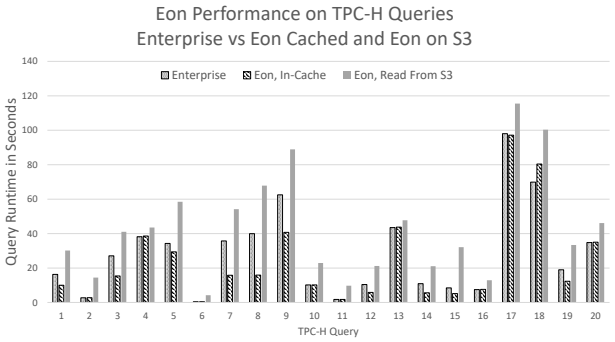


Figure 10: Performance of Eon compared to Enterprise, showing in-cache performance and reading from S3. TPC-H scale factor 200 on 4 node c3.2xlarge. Enterprise runs against EBS volumes, Eon instance storage.

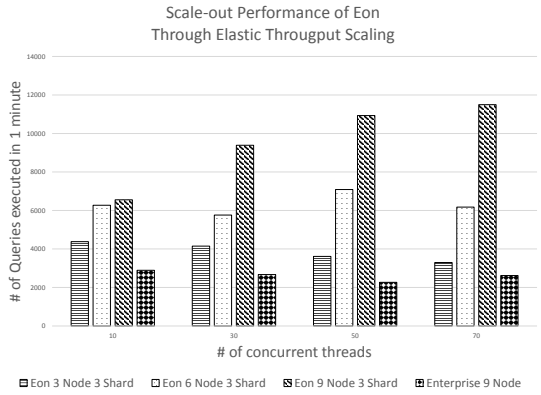
and scalability that led them to try to put it in production prior to the actual release. The infrastructure cost benefits of separated storage and compute are significant, making it much more cost effective to load massive amounts of data into Vertica.

Implementing Eon mode required rewriting several core components of a mature database while meeting the performance and stability standards associated with the Vertica brand. While many elements of Vertica's original design align well with Eon, the process was not without its challenges. Prototyping, phased design, and a robust stress testing framework were all key to a successful implementation.

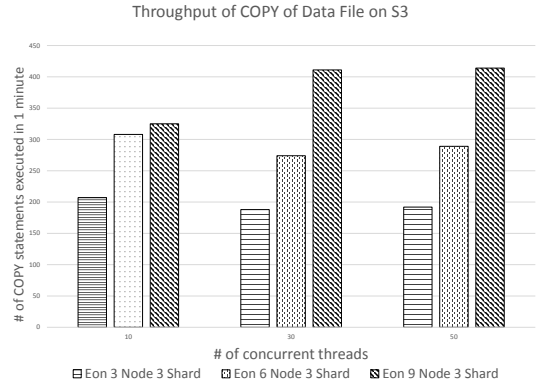
8 PERFORMANCE EVALUATION

The promise of Eon mode is to provide solid base-line performance, scale performance as nodes are added and removed, and demonstrate good operational behavior. For Vertica Enterprise users, good base-line performance means performing as well as Enterprise despite running in a shared storage environment. We ran the TPC-H queries queries against Enterprise and Eon and the results are in Figure 10. The experiment was run in AWS on 4 c3.2xlarge instances on TPC-H data loaded at scale factor 200. Enterprise runs in AWS on Elastic Block Storage (EBS) to ensure that node data persists over instance loss. Eon runs with the cache on instance storage because loss of cache data does not result in lack of durability. Eon mode matches or outperforms Enterprise on most queries. In-cache performance is a reasonable comparison because many deployments will be sized to fit the working set into the cache on the nodes. Against non-cached data, performance is significantly impacted, but response times are still reasonable.

Eon's Elastic Throughput Scaling optimization provides additional throughput for short queries when the cluster scales out as shown in Figure 11a. The experiment was run on c3.2xlarge instances against an in-cache data set stored on instance storage. The query is a customer-supplied short query comprised of multiple joins and aggregations that usually runs in about 100 milliseconds. Growing the cluster from a 3 node to a 9 node cluster while keeping the segment shard count at 3 shows a nearly linear speedup. Enterprise only supports effectively a 9 node 9 segment shard cluster and



(a) Customer query on in-cache data comparing Eon and Enterprise mode



(b) Copy of 50MB File on S3.

Figure 11: Scale-out Performance of Eon through Elastic Throughput Scaling.

exhibits performance degradation because the additional compute resources are not worth the overhead of assembling them.

Eon demonstrates improved performance on many concurrent small loads as shown in Figure 11b. In the experiment, each bulk load or COPY statement loads 50MB of input data. Many tables being loaded concurrently with a small batch size produces this type of load; the scenario is typical of an internet of things workload.

A critical operational element is system performance when nodes fail. The sharding mechanism of Eon results in a non-cliff performance scale down when a node is killed as shown in Figure 12. The query is a TPC-H query that typically runs in 6 seconds, containing multiple aggregations and a group by. A 4 node 3 shard setup shows smooth performance regression when one node is killed. As in the earlier experiment, Enterprise only supports effectively shard count

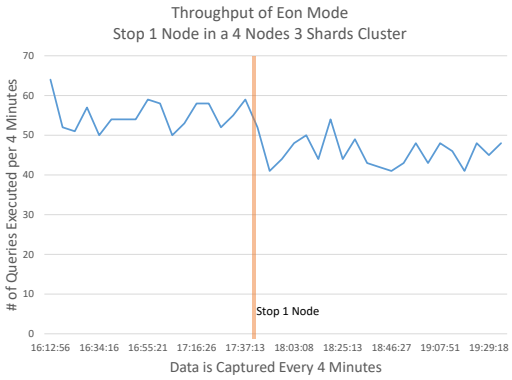


Figure 12: Throughput, Eon Mode, 4 nodes, Kill 1 node.

equals node count behavior and thus suffers higher performance degradation.

Historically, elasticity on petabyte scale databases was approached with trepidation. Elasticity in Eon mode is a function of cache size since the majority of the time is spent moving data. A typical customer deployment took less than 30 minutes to elastic the cluster up while concurrently running a full workload. Without cache fill, the process takes minutes. Performance comparisons with Enterprise are unfair as Enterprise must redistribute the entire data set.

9 RELATED WORK

Existing data layout mechanisms usually fall into two camps: the data-value-agnostic and the fixed layout. An agnostic mechanism is one like round-robin: data records stored together on a node have no relation to each other, and thus query processing requires a shuffle for any join or collation operation. A fixed layout can place related records on the same node to enable efficient query processing, but is inelastic because adjusting the node set requires expensive reshuffling of all the stored data.

Amazon RedShift [3] is a shared-nothing cloud database offered as a service on AWS. Much like Enterprise Vertica, it relies on a fixed layout and therefore node set adjustments requires expensive data redistribution. In contrast, the Snowflake Data Warehouse [6] resembles Eon Mode in that it too decouples storage from compute and allows for storage scale up and down without data movement. Snowflake's Query Optimizer assigns input file sets to worker nodes using consistent hashing over table file names. Future queries accessing the same table file will do this on the same worker node. Vertica's sharding model supports co-segmented tables, enabling faster joins by avoiding unnecessary data shuffles.

Another highly scalable cloud system is HBase [13]. Its model works by distributing tables when they become too large by performing auto-sharding. HBase has regions and maintains a mapping between regions and nodes, which is kept in a system table called META. Clients can go directly to the region server to retrieve the value of their key. Similarly MESA [9], which is a highly scalable analytic data warehousing system that stores critical measurement data related to Google's Internet advertising business, shards its data by table.

10 CONCLUSIONS

Eon mode provides great performance, enables Vertica's separation of compute and storage, and supports cloud economics. When running in-cache, Eon outperforms Enterprise mode, demonstrating support for Vertica's existing workloads. Elastic throughput scaling ensures that Vertica supports scale-out for operational dashboard queries that provide significant value for many organizations. Transparent access to non-cached data with acceptable performance makes a more massive data lake strategy practical. The operational benefits of improved fault tolerance and elasticity ensure that organizations spend more time extracting value and less on database administration.

With support for shared storage, the idea of two or more databases sharing the same metadata and data files is practical and compelling. Database sharing will provide strong fault and workload isolation, align spending with business unit resource consumption, and decrease the organizational and monetary cost of exploratory data science projects. Eon mode provides a solid substrate on which a sharing solution can be built.

By leveraging the UDFS API, Eon mode can support additional shared storage products such as Azure Blob storage [5], Google cloud storage [8], HDFS [14], Ceph [19], and so on. These storage solutions are a mix of cloud and on-premises, enabling deployment of Eon mode anywhere an organization requires. We look forward to the journey.

11 ACKNOWLEDGMENTS

Eon mode would not have been possible without the support of the entire Vertica development team. Misha Davidson green-lit the project and Nga Tran provided additional engineering resources. Jason Slaunwhite ran the initial development effort. Vertica's Pittsburgh crew (Stephen Walkauskas, John Heffner, Tharanga Gamaethige, and others) made many important design suggestions. Our QA team (Michelle Qian, Carl Gerstle, Fang Xing, Feng Tian, Packard Gibson, Mark Hayden, and others) regularly found design and implementation flaws. Lisa Donaghue and Casey Starnes provided comprehensive documentation. Finally, we would like to thank our customers without whom Vertica would not exist.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280.

[2] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. 466–475. https://doi.org/10.1109/ICDE.2007.367892

[3] Amazon. 2018. Amazon Redshift. (2018). https://aws.amazon.com/redshift/

[4] Amazon. 2018. Amazon Simple Storage Service Documentation. (2018). https://aws.amazon.com/documentation/s3/

[5] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas.

2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 143–157. https://doi.org/10.1145/2043556.2043571

[6] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[7] Asit Dan and Don Towsley. 1990. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '90)*. ACM, New York, NY, USA, 143–152. https://doi.org/10.1145/98457.98525

[8] Google. 2012. Google Cloud Storage. (2012). https://cloud.google.com/whitepapers/

[9] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. 2014. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. In *VLDB*.

[10] H.T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. In *ACM Transactions on Database Systems, Vol. 6, No. 2.* 213–226.

[11] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[12] Paul J Leach, Michael Mealling, and Rich Salz. 2005. A universally unique identifier (uuid) urn namespace. (2005).

[13] Kevin O'Dell and Jean-Marc Spaggiari. 2016. *Architecting HBase Applications, A Guidebook for Successful Development and Design.* O Reilly Media, Reading, MA.

[14] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[15] Martin Staudt and Matthias Jarke. 1996. Incremental Maintenance of Externally Materialized Views. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 75–86. http://dl.acm.org/citation.cfm?id=645922.673479

[16] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, 553–564. http://dl.acm.org/citation.cfm?id=1083592.1083658

[17] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1041–1052.

[18] Vertica. 2017. Vertica SDKs. (2017). https://my.vertica.com/docs/8.1.x/HTML/index.htm#Authoring/SupportedPlatforms/HPVerticaSDK.htm/

[19] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 307–320. http://dl.acm.org/citation.cfm?id=1298455.1298485