|  |  |
| --- | --- |
| **WEIGHT** | **SECTIONS / OBJECTIVES** |
| 10% | **Vertica Architecture**   1. **Identify key features of Vertica**    1. Real-Time Query & Loading    2. Advanced In-Database Analytics    3. Database Designer & Administration Tools    4. Columnar Storage & Execution    5. Scale-Out MPP Architecture    6. Automatic High Availability    7. Optimizer, Execution Engine & Workload Management    8. Native BI, ETL, & Hadoop/MapReduce Integration 2. **Describe high availability capabilities and describe Vertica's transaction model**    1. High availability capabilities:       1. **Distributed Database** - A Vertica database is considered K-safe if any node can fail at any given time without causing the entire database to go down. If a node does crash, when it comes back online and rejoins the database, it will recover any lost objects by querying the functioning nodes.          1. Vertica officially supports K values 0, 1, or 2.          2. A database must have a minimum number of nodes to be able to have a K-safety level greater than zero, as shown in the following table:  |  |  | | --- | --- | | **K-Level** | **Number of Nodes Required** | | 0 | 1+ | | 1 | 3+ | | 2 | 5+ | | K | (K \* 2) + 1 |  * + - 1. The following SQL command can be used to determine the K-safety of a running database:   dbadmin=> SELECT current\_fault\_tolerance FROM system;  current\_fault\_tolerance  -------------------------  1  (1 row)   * + - 1. What does the K stand for in K-safe? As a matter of convention in mathematics, K is used to represent a constant scaling value. What happens when we add nodes to a Vertica cluster? We are "scaling" it horizontally... hence the use of K.     1. **Projections** - To ensure high availability and recovery for database clusters of three or more nodes, Vertica:        1. Replicates small, un-segmented projections        2. Creates buddy projections for large, segmented projections.        3. Vertica duplicates table columns on at least K+1 nodes within the cluster.   1. Transaction model:      1. Vertica supports session-scoped isolation levels set via the SET SESSION CHARACTERISTICS command. What data can be seen by other transactions running concurrently is determined by the Isolation level for a given session.      2. The Vertica query parser understands all four standard SQL isolation levels (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE), however, internally only READ COMMITTED and SERIALIZABLE are used. Vertica will automatically convert READ UNCOMMITTED to READ COMMITTED and REPEATABLE READ to SERIALIZABLE. *Therefore, the isolation level Vertica uses may be more strict than the isolation level you choose.*      3. The isolation level of a transaction cannot be changed until it completes. Vertica internal processes (i.e. Tuple Mover, refresh operations) and DDL operations always execute with a SERIALIZABLE isolation level.      4. Vertica uses the READ COMMITTED isolation level by default.      5. The following table highlights the behaviors of transaction isolation. For specific information see, SERIALIZABLE Isolation and READ COMMITTED Isolation.  |  |  |  |  | | --- | --- | --- | --- | | **Isolation Level** | **Dirty Read** | **Non Repeatable Read** | **Phantom Read** | | READ COMMITTED | Not Possible | Possible | Possible | | SERIALIZABLE | Not Possible | Not Possible | Not Possible |  * + 1. Vertica supports transaction-level and statement-level rollbacks.        1. A transaction-level rollback discards all modifications made by a transaction.        2. A statement-level rollback reverses just the effects made by a particular statement.     2. Vertica supports using savepoints.        1. A savepoint is a special mark inside a transaction that allows all commands run after the savepoint was established to be rolled back, restoring the transaction to its former state in which the savepoint was established.   **Transaction Implementation Details**  Vertica supports conventional SQL transactions with standard ACID properties:   1. ANSI SQL 92 style-implicit transactions. You do not need to run a BEGIN or START TRANSACTION command. 2. No redo/undo log or two-phase commits. 3. The COPY command automatically commits itself and any current transaction (except when loading temporary tables). Vertica recommends that you COMMIT or ROLLBACK the current transaction before you use COPY. 4. **Differentiate between compression and encoding**    1. Compression:       1. The process of transforming data into a compact format that uses fewer bits. Compressed data cannot be directly processed; it must first be decompressed (increasing CPU usage). Vertica uses integer packing for un-encoded integers and LZO for compressible data.       2. The extensive use of compression allows a column store to occupy substantially less storage than a row store. In a column store, every value stored in a column of a projection has the same data type. This greatly facilitates compression, particularly in sorted columns. In a row store, each value of a row can have a different data type, resulting in a much less effective use of compression. The typical compression ratio observed in 2/3.       3. The size of a database is often limited by the availability of storage resources. Typically, when a database exceeds its size limitations, the administrator archives data that is older than a specific historical threshold. Vertica's efficient storage allows the database administrator to keep much more historical data in physical storage. In other words, the archiving threshold can be set to a much earlier date than in a less efficient store.    2. Encoding:       1. The process of converting data into a standard, efficient, Vertica-specific format. In Vertica, encoded data can be processed directly, which distinguishes it from compression. Vertica uses a number of different encoding strategies, depending on column data type, table cardinality, and sort order.       2. The query executor in Vertica operates on the encoded data representation whenever possible to avoid the cost of decoding. It also passes encoded values to other operations, saving memory bandwidth. In contrast, row stores and most other column stores typically decode data elements before performing any operation. 5. **Installation of Vertica**    1. Download and install the Vertica install package (rpm -Uvh *pathname*)    2. Then use install\_vertica script       1. Run as root, or sudo as a user with all privileges.       2. The installation script takes the following as basic parameters:          1. A list of hosts on which to install.          2. The Vertica RPM path and filename.          3. [Optionally] a username for the Vertica administrator. The install script creates a new system user-account. (dbadmin is used if you do not provide a username.)       3. Example: /opt/vertica/sbin/install\_vertica -s host\_list -r rpm\_package -u dba\_username |
| 10% | **Projection Design**   1. **Identify characteristics and determine features of projections used in Vertica**    1. Projections store data in a format that optimizes query execution. They are similar to materialized views in that they store result sets on disk rather than compute them each time they are used in a query. The result sets are automatically refreshed whenever data values are inserted or loaded.    2. Projections compress and encode data to greatly reduce the space required for storing data. Additionally, Vertica operates on the encoded data representation whenever possible to avoid the cost of decoding. This combination of compression and encoding optimizes disk space while maximizing query performance.    3. Projections can be segmented or replicated across database nodes depending on their size. For instance, projections for large tables can be segmented and distributed across all nodes. Un-segmented projections for small tables can be replicated across all nodes in the database.    4. Projections are transparent to end-users of SQL. The Vertica query optimizer automatically picks the best projections to use for any query.    5. Projections also provide high availability and recovery. To ensure high availability and recovery, Vertica duplicates table columns on at least K+1 nodes within the cluster. Thus, if one machine fails in a K-Safe environment, the database continues to operate normally using duplicate data on the remaining nodes. Once the node resumes its normal operation, it automatically recovers its data and lost objects by querying other nodes.    6. Projection Types       1. Super          1. Contains all columns of the individual table          2. Created automatically upon initial table load       2. Pre-Join (See “Describe the characteristics of a pre-join projection” section later in this document       3. Query-Specific          1. Contain a subset of the columns from a table          2. Optimized to support a specific query passed to the Database Designer utility       4. Buddy          1. Contains the same columns and segmentation as another projection          2. Located on multiple nodes in support of HA 2. **Define RLE**    1. Run Length Encoding (RLE) replaces sequences (runs) of identical values with a single pair that contains the value and number of occurrences. Therefore, use it only when the run length is large, such as when sorting low-cardinality columns.    2. Can be used If the average number of repeated rows is less than ten 3. **Describe ORDER BY importance in projection design**    1. The sort order optimizes for a specific query or commonalities in a class of queries based on the query predicate. For example, if the WHERE clause of a query is (x=1 AND y=2) and a projection is sorted on (x, y), the query runs almost instantaneously. It is also useful for sorting a projection to optimize a group by query. Simply match the sort order for the projection to the query group by clause. 4. **Describe the purpose of projection segmentation**    1. Segmentation distributes contiguous pieces of projections, called segments (chunks of data of similar size), for large tables across database nodes. This maximizes database performance by distributing the load.    2. Ensures high availability and recovery through K-Safety.    3. Spreads the query execution workload across multiple nodes.    4. Allows each node to be optimized for different query workloads.    5. Vertica segments large tables, to spread the query execution workload across multiple nodes. 5. **Understanding buddy projections**    1. Buddy projections are copies of segmented projections that are distributed across database nodes    2. Replicated projections serve as buddy projections    3. Vertica ensures that segments that contain the same data are distributed to different nodes. This ensures that if a node goes down, all the data is available on the remaining nodes. 6. **Determining segmentation and partitioning**    1. Vertica does not segment small tables; instead, Vertica replicates small projections, creating a duplicate of each unsegmented projection on each node.    2. Vertica uses hash segmentation to segment large projections.       1. Hash segmentation allows you to segment a projection based on a built-in hash function that provides even distribution of data across multiple nodes, resulting in optimal query execution       2. In a projection, the data to be hashed consists of one or more column values, each having a large number of unique values and an acceptable amount of skew in the value distribution.       3. Primary key columns that meet the criteria could be an excellent choice for hash segmentation.    3. Partitions are defined by the table for fast data purges and query performance. Table partitioning segregates data on each node. You can drop partitions.    4. Segmentation is defined at the projection level. Different projections for the same table have identical partitioning, but can have different segmentation clauses |
| 20% | **Loading Data into Vertica**   1. **Identify benefits of loading data into WOS and directly into ROS**    1. Write Optimized Store (WOS) is a memory-resident data structure for storing INSERT, UPDATE, DELETE, and COPY (without DIRECT hint) actions. The WOS is arranged by projection. To support very fast data load speeds, the WOS stores records without sorting, data compression, or indexing. The WOS organizes data by epoch and holds uncommitted transaction data (**NOTE: The WOS also holds committed data! Data in the WOS is distributed across the cluster nodes via buddy projections! So if one node fails, its in-memory data will be safely available in one the neighboring nodes. This is one of the reasons Vertica is said to be "Highly Available".**) *Data written to the WOS is available immediately to all users.*    2. Read Optimized Store (ROS) is a highly optimized, read-oriented, disk storage structure, organized by projection. The ROS makes heavy use of compression and indexing. You can use the COPY...DIRECT and INSERT (with /\*+direct\*/ hint) statements to load data directly into the ROS. *ROS data is not available immediately as the data first needs to be compressed and indexed.*       1. The ROS container layout can differ across nodes due to data variance.          1. Segmentation can deliver more rows to one node than another.          2. Two loads could fit in the WOS on one node and spill on another.    3. Vertica automatically decides whether the data should be placed in WOS or stored directly in ROS containers based on the amount of data processed by a COPY or INSERT command. Vertica stores large loads directly to disk and stores smaller loads in memory, which it later moves to disk. 2. **Define the actions of the Moveout and Mergeout tasks**    1. Moveout       1. "Flushes" all historical data from the WOS to the ROS.       2. Moves data from memory (WOS) into a new ROS container (on disk) which can be observed in the STORAGE\_CONTAINERS system table    2. Mergeout       1. Consolidates ROS containers when the number of ROS containers increases to a degree that it becomes necessary to merge some of them in order to avoid performance degradation. This process can be thought of as "defragmenting" the ROS.       2. Purges deleted records 3. **Describe how Mergeout purges data marked for deletion**    1. Eligible data is automatically purged when the Tuple Mover performs Mergeout operations.    2. History older than the AHM is eligible to be purged. *The Ancient History Mark (AHM) which is an epoch that represents the time until which history is retained.*       1. Two parameters determine when deleted data is eligible to be purged:          1. **HistoryRetentionTime** - The time in seconds for which delete data is saved (Default 0). *Vertica recommends using this parameter and it takes precedence. Use -1 to disable.*          2. **HistoryRetentionEpochs** - The number of epochs that are saved (Default -1) 4. **Copying data directly to ROS**    1. Use the DIRECT keyword in the COPY statement to bypass loading data into WOS, and instead, load data directly into ROS containers.    2. The DIRECT option is best suited for loading large amounts of data (100MB or more) at a time.    3. Using DIRECT for many loads of smaller data sets results in many ROS containers, which have to be combined later.    4. Moveout operation will not be needed 5. **Identifying characteristics of data file directory**    1. The disk storage location is specified when you install via the **data\_directory** parameter in the install\_vertica sript. The default value is the Database Administrator's default home directory: /home/dbadmin.    2. Data path names must contain only alphanumeric characters and cannot have leading space characters    3. Vertica refuses to overwrite a directory if it appears to be in use by another database. Therefore, if you created a database for evaluation purposes, dropped the database, and want to reuse the database name, make sure that the disk storage location previously used has been completely cleaned up.    4. Each host must have its own catalog and data locations. Hosts cannot share catalog or data locations.    5. Configure the storage so that there is enough I/O bandwidth for each node to access the storage independently. 6. **Understanding both WOS and ROS**    1. The WOS and ROS together are referred to as the “Hybrid Storage Model”    2. The WOS is in memory while the ROS is on disk    3. The WOS is optimized for writing data while the ROS is optimized or reading data 7. **Distinguishing between Moveout and Mergeout actions**    1. The Tuple Mover is responsible for both the Moveout and Mergeout operations    2. The Moveout operation works with both the WOS to ROS    3. The Moveout operation can be tuned with these parameters:       1. **MoveOutInterval** - The number of seconds the Tuple mover waits between checks for new data in the WOS to move to ROS (Default 300). *Reduce the MoveOutInterval if you expect the peak load rate to fill the WOS quickly!*       2. **MoveOutMaxAgeTime** - Forces the WOS to be written to disk at the specified interval (in seconds). (Default 30 minutes (1800 seconds))       3. **MoveOutSizePct** - The percentage of the WOS that can be filled with data before the Tuple Mover performs a Moveout operation. (Default 0)    4. The Mergeout operation only works with the ROS    5. The Mergeout operation can be tuned with the following parameter:       1. **MergeOutInterval** - The number of seconds the Tuple Mover waits between checks for new ROS files to merge out. If ROS containers are added frequently, this value might need to be decreased. (Default 600). *Reduce MergeOutInterval if you anticipate performing many DIRECT loads or inserts.* |
| 16% | **Removing Data Permanently from Vertica and Advanced Projection Design**   1. **Describe the process used to mark records for deletion**    1. Unlike most databases, the DELETE command in Vertica marks rows as deleted so that they remain available to historical queries. These deleted rows are called historical data. Retention of historical data also applies to the UPDATE command, which is actually a combined DELETE and INSERT operation.    2. A DELETE statement marks rows with delete vectors and stores them so data can be rolled back to a previous epoch. The data must eventually be purged before the database can reclaim disk space.    3. The DELETE\_VECTORS system table holds information on deleted rows to speed up the delete process. *Rows with an END\_EPOCH < the AHM are eligible to be purged.*    4. The STORAGE\_CONTAINERS system table contains the column DELETE\_VECTOR\_COUNT which is the number of delete vectors in the storage container.    5. If you delete all rows from a TEMP table (no WHERE clause) no delete vectors are kept as the rows are truly deleted. 2. **Identify the advantages and disadvantages of using delete vectors to identify records marked for deletion**    1. **Advantage**: Historical queries    2. **Disadvantages:**       1. Disk space used for the deleted rows and delete markers       2. A performance penalty for reading and skipping over deleted data 3. **Identify the process for processing a large delete or update**    1. Vertica provides multiple techniques to remove data from the database in bulk.       1. **DROP TABLE** - Permanently removes a table and its definition. Optionally removes associated views and projections as well (with the CASCADE clause)       2. **DELETE FROM TABLE** - Marks rows with delete vectors and stores them so data can be rolled back to a previous epoch. The data must eventually be purged before the database can reclaim disk space.       3. **TRUNCATE TABLE** - Removes all storage and history associated with a table. The table structure is preserved for future use. The results of this command cannot be rolled back.       4. **DROP\_PARTITION** - Removes one partition from a partitioned table. Each partition contains a related subset of data in the table. Partitioned data can be dropped efficiently, and provides query performance benefits.  |  |  |  |  | | --- | --- | --- | --- | | **Syntax** | **Performance** | **Commits Tx** | **Saves History** | | DELETE FROM base\_table | Normal | No | Yes | | DELETE FROM temp\_table | High | No | No | | DELETE FROM base\_table WHERE | Normal | No | Yes | | DELETE FROM temp\_table WHERE | Normal | No | Yes | | DELETE FROM temp\_table WHERE  temp\_table ON COMMIT PRESERVE  ROWS | Normal | No | Yes | | DELETE FROM temp\_table WHERE  temp\_table ON COMMIT DELETE  ROWS | High | Yes | No | | DROP base\_table | High | Yes | No | | TRUNCATE base\_table | High | Yes | No | | TRUNCATE temp\_table | High | Yes | No | | DROP PARTITION | High | Yes | No |  * 1. Query Performance after Large Deletes      1. To eliminate the (un-purged) deleted rows from a result set, a query must do extra processing. It has been observed if 10% or more of the total rows in a table have been deleted, the performance of a query on the table slows down.      2. Recovery performance can also be impacted by large numbers of deleted records   2. Concurrency      1. Deletes and updates take exclusive locks on the table. Hence, only one delete or update transaction on that table can be in progress at a time and only when no loads (or INSERTs) are in progress.      2. Deletes and updates on different tables can be run concurrently   3. Pre-join Projections      1. Avoid pre-joining dimension tables that are frequently updated. Deletes and updates to Pre-join projections cascade to the fact table causing a large delete or update operation.  1. **Distinguish between deleting partitions and deleting records**    1. Partitions       1. Best performance       2. Drops an entire subset of data       3. No delete vectors are created.       4. Commits transaction    2. Records       1. Worst performance       2. Need to retain history via delete vectors       3. Does not commit transaction 2. **Identify the advantages of a merge join versus a hash join**    1. Merge       1. Best performance       2. Optimizer may choose if both inputs are **pre-sorted** on the join column       3. Vertica can also performs a **sort-merge** join where the inner input is unsorted but only if the outer input is already sorted on the join keys       4. If the incoming data isn't already sorted correctly, you can facilitate a merge join by adding a projection that is sorted correctly    2. Hash       1. Used for joining large data sets       2. No sort requirement       3. Consumes more memory because a hash table is built with the values in the inner table.       4. Cost of performing a hash join is low if the entire hash table can fit in memory       5. The optimizer chooses a hash join when projections are not sorted on the join columns 3. **Identify the advantages of a group by pipe versus a group by hash**    1. Group by Pipe       1. Requires that inputs be sorted on ALL of the group columns       2. Columns in a query's GROUP BY clause must appear first in the projection's ORDER BY clause       3. The benefit of pre-sorting is that Vertica only needs to retain the data in the current group in memory       4. Generally faster and requires less memory than hash    2. Group by Hash       1. Hash input is not sorted by the group columns, so Vertica builds a hash table on those group columns 4. **Describe the characteristics of a prejoin projection**    1. Contain joins between tables that are connected by PK/FK constraints    2. Have only inner joins between tables on their PK and FK columns; Outer joins are not allowed!    3. A WHERE clause is not permitted    4. Can be described as a “de-normalized” table    5. Provide a significant performance advantage over joining tables at query run-time 5. **Determine methods available to optimize a projection for delete processing**    1. For large bulk deletion, use Partitioned Tables    2. Design projection so that all columns required by the delete or update predicate are present as columns in the projection    3. Design projections so that frequently used delete or update predicate columns appear in the SORT ORDER of all projections    4. Add additional sort columns to the sort order such that each combination of the sort key values uniquely identifies a row or a small set of rows    5. Use the EVALUATE\_DELETE\_PERFORMANCE function to analyze projections for sort order issues |
| 10% | **Cluster Management in Vertica**   1. **Identify the steps used to add nodes to an existing cluster**    1. Backup the database    2. Configure the hosts you want to add to the cluster    3. Add one or more hosts to the cluster       1. Ensure that the newly-added hosts are reachable by the existing nodes in the cluster       2. If you installed Vertica on a single node without specifying the IP address or hostname (you used localhost), it is not possible to expand the cluster. You must reinstall Vertica and specify an IP address or hostname.       3. If your database has more than one node already, you can add a node without stopping the server. However, if you are adding a node to a single-node, non-localhost installation, you must shut down both the database and spread.       4. Run the **update\_vertica** script with a minimum of the –A parameter (where host is the hostname or IP address of the system you are adding to the cluster) and the -r parameter. This will:          1. Install the Vertica RPM on the new host.          2. Perform post-installation checks, including RPM version and N-way network connectivity checks.          3. Modify spread to encompass the larger cluster.          4. Configure the Administration Tools to work with the larger cluster.   **Tips:**   1. You can use hostnames or IP addresses 2. Do not use include spaces in the hostname list provided with -A if you specified more than one host. 3. If a new RPM is specified, Vertica first installs it on the existing cluster hosts before the newly-added hosts. 4. Use the same command line parameters for the database administrator username, password, and directory path you used when you installed the cluster originally. Alternatively, you can create a properties file to save the parameters during install and then re-using it on subsequent install and update operations. 5. If you are installing using sudo, the database administrator user (dbadmin) must already exist on the hosts you are adding and must be configured with passwords and home directory paths identical to the existing hosts. Vertica sets up passwordless ssh from existing hosts to the new hosts, if needed. 6. If you initially used the -T option to configure spread to use direct, point-to-point communication between nodes on the subnet, and you want to continue to do so, use the -T option when you add the new host. Otherwise, the entire cluster is reconfigured to use the default UDP broadcast.    1. Add Node(s) to a Database       1. Management Console          1. Use Manage page          2. Nodes in STANDBY state are eligible for addition. STANDBY nodes are nodes available in the cluster but not yet assigned to the database       2. Administration Tools          1. Advanced Tools Menu/ Cluster Management/Add Host(s)          2. You can chose K-Safety          3. You can rebalance the database now, or create a script to rebalance it later 7. **Describe the benefits of having identically sorted buddy projections**    1. They can significantly improve load, recovery, and site node performance because data doesn’t have to be resorted.   For instance, in a database with a K-safety of one (1) or two (2), buddy projections are used for data recovery. If a node fails, it queries the other nodes to recover data through buddy projections. If a projection's buddies use different sort orders, it takes longer to recover the projection because the data has to be resorted during recovery to match the sort order of the projection.   1. **Identify the steps of online recovery of a failed node** 2. Reboot the machine into runlevel 1, which is a root and console-only mode. Runlevel 1 prevents network connectivity and keeps Vertica from attempting to reconnect to the cluster. 3. In runlevel 1, validate that the hardware has been repaired, the controllers are online, and any RAID recover is able to proceed. Note: You do not need to initialize RAID recover in runlevel 1; simply validate that it can recover. 4. Once the hardware is confirmed consistent, only then reboot to runlevel 3 or higher. 5. At this point, the network activates, and Vertica rejoins the cluster and automatically recovers any missing data. 6. **Define local segmentation capability in Vertica**    1. Relates to Scaling Factor which determines the number of storage containers used to store a projection across the database.       1. Use the SET\_SCALING\_FACTOR function to change your database's scaling factor.   dbadmin=> SELECT SET\_SCALING\_FACTOR(6);  SET\_SCALING\_FACTOR  --------------------  SET  (1 row)   * + 1. The scaling factor can be an integer between 1 and 32.     2. The default scaling factor is "4" for new installs of Vertica and for upgraded installs of Vertica that had local segments disabled.     3. The number of storage containers should be greater than or equal to the number of partitions multiplied by the number of local segments     4. By default, the scaling factor only has an effect when Vertica rebalances the database     5. Check the ELASTIC\_CLUSTER table to determine if elastic clustering is enabled:   dbadmin=> select is\_enabled from ELASTIC\_CLUSTER;  is\_enabled  ------------  t  (1 row)   * 1. Enabling **Local Segmentation** tells Vertica to always segment its data based on the scaling factor, so the data is always broken into containers that are easily moved.      1. Do this if your database is growing rapidly and is constantly busy      2. Having the data segmented in this way dramatically speeds up the process of adding or removing nodes, since the data is always in a state that can be quickly relocated to another node. The rebalancing process that Vertica performs after adding or removing a node just has to decide which storage containers to relocate, instead of first having to first break the data into storage containers.      3. Local data segmentation increases the number of storage containers stored on each node. This is not an issue unless a table contains many partitions. For example, if the table is partitioned by day and contains one or more years. If local data segmentation is enabled, then each of these table partitions is broken into multiple local storage segments, which potentially results in a huge number of files which can lead to ROS "pushback" (Too Many ROS containers). Consider your table partitions and the effect enabling local data segmentation may have before enabling the feature.      4. The number of local segments is determined by multiplying the scaling factor by the number of nodes in the cluster.      5. Best practice is to enable local data segmentation if:         1. The database does not contain tables with hundreds partitions         2. The number of nodes in the database cluster is a power of two         3. Future plan includes the expansion or contraction of the size of the cluster      6. To enable local segmentation, use the ENABLE\_LOCAL\_SEGMENTS function.      7. To disable local segmentation, use the DISABLE\_LOCAL\_SEGMENTATION function  1. **Distinguish between the items in a Vertica cluster**    1. In Vertica, the physical architecture is designed to distribute physical storage and to allow parallel query execution over a potentially large collection of computing resources.    2. The most important terms to understand are host, instance, node, cluster, and database:       1. **Host** - A computer system with a 32-bit (non-production use only) or 64-bit Intel or AMD processor, RAM, hard disk, and TCP/IP network interface (IP address and hostname). Hosts share neither disk space nor main memory with each other.       2. **Instance** - An instance of Vertica consists of the running Vertica process and disk storage (catalog and data) on a host. Only one instance of Vertica can be running on a host at any time.       3. **Node** - A host configured to run an instance of Vertica. It is a member of a database cluster. For a database to have the ability to recover from the failure of a node requires at least three nodes. Vertica recommends that you use a minimum of four nodes.       4. **Cluster** - Refers a collection of hosts (nodes) bound to a database. A cluster is not part of a database definition and does not have a name.       5. **Database** - A cluster of nodes that, when active, can perform distributed data storage and SQL statement execution through administrative, interactive, and programmatic user interfaces. |
| 10% | **Backup/Restore and Resource Management in Vertica**   1. **Identify situations when a backup is recommended**    1. Before you upgrade Vertica to another release.    2. Before you drop a partition.    3. After you load a large volume of data.    4. If the epoch in the latest snapshot is earlier than the current ancient history mark (AHM).    5. Before and after you add, remove, or replace nodes in your database cluster.    6. After recovering a cluster from a crash. 2. **Identify features of the Vertica file used for backup and restore**    1. The script is vbr.py (written in Python)       1. Allows you to copy data to:          1. A local directory on the nodes in the cluster          2. One or more hosts outside of the cluster          3. A different Vertica cluster (effectively cloning your database)       2. Helps to automate backing up your database (via runtime parameters and cron)       3. Create full database backups and incremental database backups       4. Create object level backup (snapshots)    2. Important terminology:  |  |  | | --- | --- | | **TERM** | **DEFINTION** | | Snapshots | A consistent image of all objects and data in the database at the time the snapshot is taken. Object-level snapshots consist of a subset of database objects, selected by the user. They include other objects in the dependency graph, and associated data in the database at the time the object-level snapshot is taken.  You can refer to snapshots by a user-defined descriptive name, such as FullDBSnap, Schema1Snap, Table1Snap, and so on. | | Archive | A number of same-name snapshots, past and present. Each archive can have a different retention policy. If TSnap names a snapshot of table T, and you take the snapshot daily, keeping 7 snapshots in the archive would let you revert back to any of the last week's snapshots of table T. | | Backup Host | The directory location on a backup host where snapshots are saved. This location can comprise multiple snapshots, including associated archives. All snapshots in the same backup location share data files (through hard links). The snapshots are also compatible, meaning that after restoring a full database snapshot, any object snapshot from the same backup location is eligible to be restored. | | Object-level snapshot | One or more schemas or tables, or group of such objects, saved on a backup host. The conglomerate parts of the object-level snapshot do not contain the entire database.  In earlier Vertica versions, object-level snapshots could not exist because a snapshot always contained the entire database. | | Incremental backups | A successive backup consisting only of new or changed data. | | Selected objects | The objects chosen to be part of an object-level snapshot. For example, if tables T1 and T2 are backed up in a snapshot, these objects comprise the selected objects. | | Dependent objects | Objects that should be part of an object-level snapshot due to their dependency. For example, a table with a foreign key can exist on its own, but must be backed up with the primary key table, due to table constraints. Projections anchored on a table in the selected objects are also dependent objects. | | Principal objects | The objects on which both selected and dependent objects depend. For instance, each table and projection has an owner. The owner is a principal object. |  1. **Define the Resource Manager's role in query processing**    1. The Resource Manager (RM) provides options and controls for resolving tension created when attempting to provide concurrent queries the maximum amount of resources (thereby getting fastest run time for that query), while ensuring that every query eventually gets serviced and that true system limits are respected at all times.    2. When a query is submitted to the database, the following series of events occur:       1. The query is parsed, optimized to determine an execution plan, and distributed to the participating nodes.       2. The Resource Manager is invoked on each node to estimate resources required to run the query and compare that with the resources currently in use. One of the following will occur:          1. If the memory required by the query alone would exceed the machine's physical memory, the query is rejected - it cannot possibly run. Outside of significantly under-provisioned nodes, this case is very unlikely.          2. If the resource requirements are not currently available, the query is queued. The query will remain on the queue until either sufficient resources are freed up and the query runs or the query times out and is rejected.          3. Otherwise the query is allowed to run.       3. The query starts running when all participating nodes allow it to run.    3. Once the query is running, the Resource Manager further manages resource allocation using RUNTIMEPRIORITY and RUNTIMEPRIORITYTHRESHOLD parameters for the resource pool.       1. RUNTIMEPRIORITY - Use this parameter to set the priority for running queries in this resource pool. Any query with a duration that exceeds the value in the RUNTIMEPRIORITYTHRESHOLD property will be assigned the run-time priority you specify here.          1. If you want to ensure that short queries will always run at a high priority, set the RUNTIMEPRIORITY parameter to MEDIUM or LOW and set the RUNTIMEPRIORITYTHRESHOLD to a small number that will accommodate your short queries.          2. If you want all queries in the resource pool to always run a specified value, set the RUNTIMEPRIORITY parameter to HIGH, MEDIUM, or LOW and also set the RUNTIMEPRIORITYTHRESHOLD to 0. Setting RUNTIMEPRIORITYTHRESHOLD to 0 effectively turns off the RUNTIMEPRIORITYTHRESHOLD feature. All queries will run with the RUNTIMEPRIORITY of the resource pool.       2. **RUNTIMEPRIORITYTHRESHOLD** - Use this parameter to specify the duration (in seconds) of queries that should always run with HIGH run-time priority. Because all queries begin running with a RUNTIMEPRIORITY of HIGH, queries that finish within the specified threshold will run at a HIGH priority; all other queries will be assigned the runtime priority assigned to the resource pool. To disable this feature, set the RUNTIMEPRIORITYTHRESHOLD to 0. 2. **Describe the differences between maxconcurrency and plannedconcurrency**    1. **MAXCONCURRENCY** - Use this parameter if you want to impose a hard limit on the number of concurrent requests that are allowed to run against any pool, including the GENERAL pool. Instead of limiting this at the pool level, it is also possible to limit at the connection level **using MaxClientSessions**       1. **MaxClientSessions - Determines the maximum number of client sessions that** can be run on the database. The default value includes 5 additional administrative logins. Setting this parameter to 0 is useful for preventing new client sessions from being opened while you are shutting down the database.    2. **PLANNEDCONCURRENCY** - This parameter specifies the typical number of queries running concurrently in the system.       1. Set PLANNEDCONCURRENCY to AUTO to specify that Vertica should calculate this number. Vertica takes the lower of these two values:          1. Number of cores          2. Memory/2GB       2. The minimum value is 4       3. For the TM pool, the PLANNEDCONCURRENCY parameter must be proportional to the size of the RAM, the CPU, and the storage subsystem. Depending on the storage type, if you increase PLANNEDCONCURRENCY for the Tuple Mover threads, you might create storage I/O bottleneck. Monitor the storage subsystem; if it becomes saturated with long I/O queues, more than two I/O queues, and long latency in read and write, adjust the PLANNEDCONCURRENCY parameter to keep the storage subsystem resources below saturation level. In addition, you might need to:          1. Partition storage data files          2. Adjust block-size optimization on storage subsystems such as RAID 5 or RAID 10          3. Identify the optimal number of disks in the RAID array.       4. Consider the tradeoff between giving each query its maximum amount of resources and allowing many concurrent queries to run in a reasonable amount of time       5. This parameter can be used in combination with MEMORYSIZE to tune the memory used by a query down to a specific size.       6. For clusters where the number of cores differs on different nodes, AUTO can apply differently on each node. Distributed queries run like the minimal effective planned concurrency. Single-node queries run with the planned concurrency of the initiator.       7. If you created or upgraded your database in 4.0 or 4.1, the PLANNEDCONCURRENCY setting on the GENERAL pool defaults to a too-small value for machines with large numbers of cores. To adjust to a more appropriate value: ALTER RESOURCE POOL general PLANNEDCONCURRENCY <#cores>; 3. Using query profiles and resource pools    1. Query Profiles       1. Profiling data lets you examine where time is spent during query execution       2. Unlike the EXPLAIN plan, where the cost and row counts are estimates, counters and plan from profiled data reflect what really happened and lets you consider some of the following:          1. Query plan quality issues          2. Projection design issues          3. If the query is network bound          4. How much memory each operator allocated          5. If a query rewrite might speed up the query          6. How many threads are executing for each operator          7. How the data has flowed through each operator at different points in time over the life of the query       3. Profiling data is available for:          1. Any query that has been explicitly profiled          2. Any query that is currently executing       4. To determine how the database is performing, you can profile the following areas:          1. Session—Provides basic session parameters and lock time out data          2. Query—Provides general information about queries that ran, such as the query strings used and the duration of queries          3. Execution Engine—Provides detailed information about the execution run of each query       5. To determine if profiling is enabled, use the following command:   dbadmin=> SELECT SHOW\_PROFILING\_CONFIG();  SHOW\_PROFILING\_CONFIG  -----------------------------------------  Session Profiling: Local off, Global off  EE Profiling: Local off, Global on  Query Profiling: Local off, Global off  (1 row)   * + 1. To enable profiling for the current session, issue the ENABLE\_PROFILING() command: SELECT ENABLE\_PROFILING('type\_of\_profiling');   Where type\_of\_profiling is one of the following:   * Session - Establishes profiling for sessions. * Query - Establishes profiling for queries. * ee - Establishes profiling for query execution runs.   + 1. To disable profiling for the current session, issue the ENABLE\_PROFILING() command: SELECT DISABLE\_PROFILING('type\_of\_profiling');     2. To enable profiling for all sessions on all nodes, enter the following command: SELECT SET\_CONFIG\_PARAMETER('global\_profiling\_type', 1);   Where global\_profiling\_type is one of the following:   * GlobalSessionProfiling - Establishes profiling for sessions. * GlobalQueryProfiling - Establishes profiling for queries. * GlobalEEProfiling- Establishes profiling for query execution runs.   + 1. To disable profiling for all sessions on all nodes, enter the following command: SELECT SET\_CONFIG\_PARAMETER('global\_profiling\_type', 0);     2. To profile a single statement add the PROFILE keyword to the beginning of the statement   dbadmin=> PROFILE SELECT \* FROM dual;  NOTICE 4788: Statement is being profiled  HINT: Select \* from v\_monitor.execution\_engine\_profiles where transaction\_id=45035996277823360 and statement\_id=17;  NOTICE 3557: Initiator memory for query: [on pool sysquery: 4103 KB, minimum: 4103 KB]  NOTICE 5077: Total memory required by query: [4103 KB]  dummy  -------  X  (1 row)   * + 1. Real-time profiling counters are available for all currently-executing statements—including internal operations such as mergeout, recovery, and refresh—but only while the statements are executing.        1. EXECUTION\_ENGINE\_PROFILES system table contains counters, i.e.           1. execution time (μs)           2. rows produced           3. total merge phases           4. completed merge phases           5. current size of temp files (bytes)     2. Three tables contain profile data:  |  |  | | --- | --- | | **TABLE** | **DESCRIPTION** | | EXECUTION\_ENGINE\_PROFILES | Provides profiling information for query execution runs. | | QUERY\_PROFILES | Provides profiling information for queries. | | SESSION\_PROFILES | Provides profiling information for sessions. |  * + 1. To clear profiling data use the command:   SELECT CLEAR\_PROFILING('type\_of\_profiling');   * + 1. Use query labeling to pass a user-defined label to a Vertica query as a hint that takes the following form: /\*+label(label-name)\*/. The labels are available in the profile tables via the identifier column.   1. Resource Pools      1. A resource pool comprises a pre-allocated subset of the system resources, with an associated queue      2. Resource pool parameters  |  |  |  | | --- | --- | --- | | **Parameter** | **Data Type** | **Description** | | MEMORYSIZE | VARCHAR | Value of the amount of memory allocated to the resource pool. | | MAXMEMORYSIZE | VARCHAR | Value assigned as the maximum size the resource pool could grow by borrowing memory from the GENERAL pool. | | EXECUTIONPARALLELISM | INTEGER | [Default: AUTO] Limits the number of threads used to process any single query issued in this resource pool.  When set to AUTO, Vertica sets this value based on the number of cores, available memory, and amount of data in the system. Unless data is limited, or the amount of data is very small, Vertica sets this value to the number of cores on the node.  Reducing this value increases the throughput of short queries issued in the pool, especially if the queries are executed concurrently.  If you choose to set this parameter manually, set it to a value between 1 and the number of cores. | | PRIORITY | INTEGER | Value of PRIORITY parameter specified when defining the pool. | | RUNTIMEPRIORITY |  | Value that indicates the amount of run-time resources (CPU, I/O bandwidtch) the Resource Manager should dedicate to running queries in the resource pool. Valid values are:   * HIGH * MEDIUM (Default) * LOW   These values are relative to each other. Queries with a HIGH run-time priority are given more CPU and I/O resources than those with a MEDIUM or LOW run-time priority. | | RUNTIMEPRIORITYTHRESHOLD | INTEGER | Value that specifies the time limit (in seconds) by which a query must finish before the Resource Manager assigns to it the RUNTIMEPRIORITY of the resource pool. All queries begin running at a HIGH priority. When a query's duration exceeds this threshold, it is assigned the RUNTIMEPRIORITY of the resource pool. Default is 2. | | QUEUETIMEOUT | INTEGER | Value in seconds no greater than 31556926 (just over one year) of QUEUETIMEOUT parameter specified when defining the pool. Represents the maximum amount of time the request is allowed to wait for resources to become available before being rejected. Default 300. | | PLANNEDCONCURRENCY | INTEGER | Value of PLANNEDCONCURRENCY parameter specified when defining the pool, which represents the number of concurrent queries that are normally expected to be running against the resource pool. | | MAXCONCURRENCY | INTEGER | Value of MAXCONCURRENCY parameter specified when defining the pool, which represents the maximum number of concurrent execution slots available to the resource pool. | | RUNTIMECAP | INTERVAL | [Default: NONE] Sets the maximum amount of time any query on the pool can execute. Set RUNTIMECAP using interval, such as '1 minute' or '100 seconds' (see Interval Values (page 37) for details). This value cannot exceed one year. Setting this value to NONE specifies that there is no time limit on queries running on the pool. If the user or session also has a RUNTIMECAP, the shorter limit applies. | | SINGLEINITIATOR | BOOLEAN | Value that indicates whether all requests using this pool are issued against the same initiator node or whether multiple initiator nodes can be used; for instance in a round-robin configuration. [Default false] This parameter is included for backwards compatibility only. Do not change the value. |  * + 1. Built-in (internal) resource pools  |  |  | | --- | --- | | **POOL** | **DESCRIPTION** | | GENERAL | A special, catch-all pool used to answer requests that have no specific resource pool associated with them. Any memory left over after memory has been allocated to all other pools is automatically allocated to the GENERAL pool. The MEMORYSIZE parameter of the GENERAL pool is undefined (variable), however, the GENERAL pool must be at least 1GB in size and cannot be smaller than 25% of the memory in the system.  The MAXMEMORYSIZE parameter of the GENERAL pool has special meaning; when set as a % value it represents the percent of total physical RAM on the machine that the Resource Manager can use for queries. By default, it is set to 95%. The GENERAL.MAXMEMORYSIZE governs the total amount of RAM that the Resource Manager can use for queries, regardless of whether it is set to a percent or to a specific value (for example, '10GB')  Any user-defined pool can ―borrow‖ memory from the GENERAL pool to satisfy requests that need extra memory until the MAXMEMORYSIZE parameter of that pool is reached. If the pool is configured to have MEMORYSIZE equal to MAXMEMORYSIZE, it cannot borrow any memory from the GENERAL pool and is said to be a standalone resource pool. When multiple pools request memory from the GENERAL pool, they are granted access to general pool memory according to their priority setting. In this manner, the GENERAL pool provides some elasticity to account for point-in-time deviations from normal usage of individual resource pools. | | SYSQUERY | The pool that runs queries against ***system monitoring and catalog tables*** (page 941). The SYSQUERY pool reserves resources for system table queries so that they are never blocked by contention for available resources. | | SYSDATA | The pool reserved for temporary storage of intermediate results of queries against system monitoring and catalog tables (page 941). If the SYSDATA pool size is too low, Vertica cannot execute queries for large system tables or during high concurrent access to system tables. | | WOSDATA | The Write Optimized Store (WOS) resource pool. Data loads to the WOS automatically spill to the ROS once it exceeds a certain amount of WOS usage; the PLANNEDCONCURRENCY parameter of the WOS is used to determine this spill threshold. For instance, if PLANNEDCONCURRENCY of the WOSDATA pool is set to 4, once a load has occupied one quarter of the WOS, it spills to the ROS. | | TM | The Tuple Mover (TM) pool. You can use the MAXCONCURRENCY parameter for the TM pool to allow more than one concurrent TM operation to occur. | | RECOVERY | The pool used by queries issued when recovering another node of the database. The MAXCONCURRENCY parameter is used to determine how many concurrent recovery threads to use. You can use the PLANNEDCONCURRENCY parameter (by default, set to twice the MAXCONCURRENCY) to tune how to apportion memory to recovery queries. | | REFRESH | The pool used by queries issued by the ***PROJECTION\_REFRESHES*** (page 1052) operations. Refresh does not currently use multiple concurrent threads; thus, changes to the MAXCONCURRENCY values have no effect. | | DBD | The Database Designer pool, used to control resource usage for the DBD internal processing. Since the Database Designer is such a resource-intensive process, the DBD pool is configured with a zero (0) second QUEUETIMEOUT value. Whenever resources are under pressure, this timeout setting causes the DBD to time out immediately, and not be queued to run later. The Database Designer then requests the user to run the designer later, when resources are more available. Vertica recommends that you do not reconfigure this pool. |  * + 1. System tables with RP info:        1. RESOURCE\_POOL\_DEFAULTS - Provides information about the default values for resource pools        2. RESOURCE\_POOLS - Displays information about the parameters specified for the resource pool when it was created        3. RESOURCE\_POOL\_STATUS - Provides configuration settings of the various resource pools in the system, including internal pools. Use the IS\_INTERNAL column to identify internal pools        4. RESOURCE\_ACQUISITIONS - Retains information about resources (memory, open file handles, threads) acquired by each running request for each resource pool in the system        5. RESOURCE\_QUEUES - Provides information about requests pending for various resource pools        6. RESOURCE\_REJECTIONS - Monitors requests for resources that are rejected by the Resource Manager        7. RESOURCE\_REJECTION\_DETAILS - Records an entry for each resource request that Vertica denies. This is useful for determining if there are resource space issues, as well as which users/pools encounter problems     2. User defined resource pools        1. Use CREATE RESOURCE POOL *pool-name* command with parameters discussed above        2. User defined resource pools can be created and assigned to users or user profiles. |
| 10% | **Monitoring and Troubleshooting Vertica**   1. **Define the use of Management Console in monitoring Vertica**    1. MC provides ability to monitor:       1. Multiple database cluster states       2. Cluster-wide alerts and messages       3. Database and agent log entries       4. Database activity in relation to physical resource usage       5. Current database size usage statistics, as well as statistics over time       6. User activity on MC          1. User log-on/log-off activities          2. Database creation          3. Database connection through the console interface          4. Start/stop a database          5. Remove a database from the console view          6. Drop a database          7. Database rebalance across the cluster          8. License activity views on a database, as well as new license uploads          9. Workload Analyzer views on a database          10. Database password changes          11. Database settings changes (individual settings are tracked in the audit record)          12. Syncing the database with the cluster (who clicked Sync on grid view)          13. Query detail viewings of a database          14. Node changes (add, start, stop, replace)          15. User management (add, edit, enable, disable, delete)          16. LDAP authentication (enable/disable)          17. Management Console setting changes (individual settings are tracked in the audit record)          18. SSL certificate uploads          19. Message deletion and number deleted          20. Console restart from the browser interface          21. Factory reset from the browser interface       7. Issues related to the MC server       8. Error handling and feedback          1. Low disk space          2. Read-only file system          3. Loss of K-safety          4. Current fault tolerance at critical level          5. Too many ROS containers          6. WOS overflow          7. Change in node state          8. Recovery error          9. Recovery failure          10. Recovery lock error          11. Recovery projection retrieval error          12. Refresh error          13. Refresh lock error          14. Workload Analyzer operations          15. Tuple Mover error          16. Timer service task error          17. Last Good Epoch (LGE) lag          18. License size compliance          19. License term compliance       9. Key performance indicators (KPI) that report on the cluster's overall health          1. **Node state**—(default view) shows node status (up, down, k-safety critical) by color; you can filter which nodes appear on the page by sliding the health filter from left to right          2. **CPU Utilization**—average CPU utilization          3. **Memory Utilization**—average percent RAM used          4. **Storage Utilization**—average percent storage used       10. Activity flags that report on important events in time           1. Workload Analyzer has occurred           2. Rebalance Operation has occurred    2. Over Page       1. **Database** - Mirrors information shown on the Databases and Clusters page, such as database name, the database status (up, down, initializing, validating, recovering, critical), the number of nodes in the cluster and their status.       2. **Cluster** - Statistics for cluster-wide information, such as cluster size (the number of hosts), operating system type and revision (as found in /etc/issue), Vertica version, and charts of CPU utilization, network traffic, and memory usage over the last 24 hours. If you click the Details link in the top right of the cube, the Manage page for the database displays.       3. **Performance** - Latest Workload Analyzer tuning recommendations and cost. The WLA task begins 60 seconds after Management Console starts and then once per day unless you click Analyze, which immediately runs the Workload Analyzer. Use the arrows at bottom right of the cube to navigate through multiple pages of tuning recommendations. Alerts. Summarized list of message count by type/severity, such as Emergency, Alert, Critical, Error, Warning, Notice, and Info. Two columns appear for each severity: Today and All. The Details link displays the Message Center, which is pre-filtered by the database ID.       4. **Jobs** - A high overview of MC-managed database activity in graphical format with respect to 'now' (when the last refresh occurred) and over the last 24 hours. Click the Details link to open the Activity page, which displays a graph of different job types; for example, user versus system queries, as well as CPU and memory usage for those jobs. You change the dates or you can click the defaults for 1 hour, 1 day, 1 week.       5. **License** - High-level snapshot of the current database license. This cube includes information about the license total size and how much is in use, the grace period, expiration date, and license type. The Details page shows even more information in a graphical format, which lets you see at a glance if your database is approaching its limit.    3. Messages and icons are color coded to represent the level of severity. For example:       1. Red - Emergency, Alert, and Critical       2. Orange - Error       3. Yellow - Warning       4. Monitoring Vertica       5. Green - Notice       6. Blue - Informational       7. Purple/Black/Clear - Debug 2. **Determine methods to troubleshoot spread**    1. Spread daemon needs to run on each node    2. Check if daemon is running using command: ps -ef | grep -i spread    3. If spread is not running, restart it as root using the command: /etc/init.d/spreadd restart    4. If spread is running check its status using this command as root: /etc/init.d/spreadd status    5. Can also use the admintools command line check\_spread tool:   bash-3.2$ admintools -t check\_spread  Checking multicasting layer  Success  Spread check succeeded   * 1. Use ifconfig command to check current IP addresses of the hosts and verify they match the values that are stored in the spread configuration file: /opt/vertica/config/vspread.conf   2. If spread fails to start, check log files for permission and syntax problems:      1. /tmp/spread\*.log      2. /var/log/spreadd.log   3. Check if keepalived daemon is running:   bash-3.2$ ps ax | grep keepalived  13884 pts/0 S+ 0:00 grep keepalived   * 1. The spread daemon relies on all of the nodes in the having their clocks synchronized for timing purposes. If your nodes do not have NTP running, the installation can fail with a spread configuration error as well as other potential errors   2. If an embedded Vertica database fails and the connection is lost, MC agents will check if spread is running. If running, MC attempts to recover VERTICA as a background process. MC does not have privileges to restart spread, so if spread has stopped running, a dialog box notifies you to manually restart spread.  1. **Describe how to disallow user connections, while preserving dbadmin connectivity**    1. Set the **MaxClientSessions** database parameter to 0. The database will always allow 5 admin account logins. The default value for **MaxClientSessions** is 50.   SELECT SET\_CONFIG\_PARAMETER ('MaxClientSessions', 0);   1. **Defining, using and logging into Management Console**    1. Information about MC settings, which you set during MC configuration, is stored in /opt/vertica/config/console.properties on the MC server    2. The default port MC accepts connections on is 5450.    3. Connect to MC using URL: https://hostname:5450/ or https://xx.xx.xx.xx:5450/    4. MC superuser administrator is a Linux user account. 2. **Administering a cluster using Management Console**    1. **Create a New Cluster**       1. Created the private key file and copied it to your local machine (so you the severs can use password-less SSH)       2. Connect to Management Console and log in as an MC administrator.       3. On the Home page, click the Databases and Clusters task.       4. Click the green plus sign and select Create Cluster.       5. The Create Cluster wizard opens. Provide the following then click Next          1. Cluster name - This is a label for the cluster          2. Vertica Admin User - This is the user that is created on each of the nodes when they are installed. Typically this is 'dbadmin'. The user has access to Vertica and is also an OS user on the host.          3. Provide a password for the Vertica Admin User. This password is set for each of the nodes when Vertica is installed.          4. Vertica Admin Path: Provide a path where the catalog files are stored. By default this is /home/dbadmin unless you changed the setting on the MC.       6. Click Next and specify the private key file and host information:          1. Click Browse and navigate to the private key file (vid\_rsa) that you created earlier (i)          2. Include the host IP addresses. Here you have three options:             1. Specify later (but include number of nodes). This option allows you to specify the number of nodes, but not the specific IPs. You can specify the specific IPs before you validate hosts.             2. Import IP addresses from local file - You can specify the hosts in a CSV file using either IP addresses or host names.             3. Enter a range of IP addresses - You can specify a range of IPs to use for new nodes. For example 192.168.1.10 to 192.168.1.30. The range of IPs must be on the same subnet.       7. Click Next and select the software and license:          1. Vertica Software -If one or more Vertica rpm(s) have been uploaded, then you can select one from the list, otherwise select the Upload a new local vertica binary file radio button and browse to a Vertica rpm file on your local system.          2. Vertica License - Click Browse and navigate to a local copy of your Vertica license.       8. Click Next. The Create cluster page opens. If you did not specify the IP addresses then you must select each host icon and provide an IP. Enter the IP in the box and click Apply for each host.       9. The hosts are now ready for Host Validation and Cluster Creation.    2. **Validate Hosts**       1. Host validation is the process where the MC runs tests against each host in a proposed cluster.       2. How to validate hosts:          1. Connect to Management Console and log in as an MC administrator. On the MC Home page, click the Databases and Clusters task.          2. In the list of databases and clusters, select the cluster on which you have recently run the cluster installation wizard (Creating... appears under the cluster) and click View.          3. Validate one or several hosts:             1. To validate a single host, click the host icon, then click Validate Host in the upper right corner of the screen.             2. To validate All hosts at the same time, click All in the Node List, then click Validate Host in the upper right corner of the screen.             3. To validate more than one host, but not all of them, CTRL-click the host numbers in the node list, then click Validate Host in the upper right corner of the screen.       3. Wait while validation proceeds. The validation step takes several minutes to complete. The tests run in parallel for each host, so the number of hosts does not necessarily increase the amount of time it takes to validate all the hosts if you validate them at the same time. Hosts validation results in one of three possible states:          1. Green check mark - The host is valid and this host can be included in the cluster.          2. Orange triangle - The host can be added to the cluster, but warnings were generated. Click the tests(s) in the host validation window to see details about the warnings.          3. Red X - The host is not valid. Click the tests(s) in the host validation window that have red X's to see details about the errors. The errors must be corrected and the host must be revalidated, or the host must be removed, before the cluster can be created. To remove the host, highlight the host icon or the IP address in the Node List and click Remove Host near the upper right corner of the screen.       4. All hosts must be valid before you can create the cluster. Once all hosts are valid a Create Cluster button appears near the top right corner of the screen.       5. Click Create Cluster to install Vertica on each host and assemble the nodes into a cluster. The process takes a few minutes as the software is copied to each host and installed. This process is done in parallel.       6. When the process has completed The Success window opens. You can optionally create a database on the new cluster at this time by clicking Create Database, or click Done to create the database at a later time.    3. **Creating a database on a cluster**       1. If you are already on the Databases and Clusters page, skip to the next step; otherwise connect to MC and sign in as an MC administrator. On the Home page, click the Databases and Clusters task.       2. If there is a database running on the cluster on which you want to add a new database, stop the running database by selecting the database and clicking Stop. Wait for the running database to have a status of Stopped. If no databases exist on the cluster then continue to the next step. Note: Although you can have multiple databases in Vertica, only one database can be running at a time on a cluster.       3. Click the cluster on which you want to create the new database and click Create Database.       4. The Create Database wizard opens. Provide the following information:          1. Database name and password. See Creating a database name and password for rules.          2. Optionally click Advanced to open the advance settings and change the port and catalog, data, and temporary data paths. By default the MC server port is 5450 and paths are /home/dbadmin, or whatever you defined for the paths when you ran the cluster creation wizard. Do not use the default agent port 5444 as a new setting for the MC server port.       5. Click Continue. The Database Configuration window opens with the options you provided and a graphical representation of the nodes appears on the page. By default, all nodes are selected to be part of this database (denoted by a green check mark). You can optionally click each node and clear Include host in new database to exclude that node from the database. Excluded nodes are gray. If you change your mind, click the node and select the Include check box.       6. Click Create in the Database Configuration window to create the database on the nodes. The creation process takes a few moments and then the database is started and a Success message appears.       7. Click OK to close the success message. The Database Manager page opens and displays the database nodes. Nodes not included in the database are gray. |
| 14% | **Analytics and Performance Tuning**   1. **Understanding analytics syntax**    1. Analytic functions return aggregate results but they do not group the result set. They return the group value multiple times, once per record.    2. General syntax:   ANALYTIC\_FUNCTION( argument-1, ..., argument-n )  OVER( [ window\_partition\_clause ]  [ window\_order\_clause ]  [ window\_frame\_clause ] )   |  |  | | --- | --- | | **Syntactic Construct** | **Definition** | | ANALYTIC\_FUNCTION() | Vertica provides a number of analytic functions that allow advanced data manipulation and analysis. Each of these functions takes one or more arguments. | | OVER(...) | Specifies partitioning, ordering, and window framing for the function—important elements that determine what data the analytic function takes as input with respect to the current row. The OVER() clause is evaluated after the FROM, WHERE, GROUP BY, and HAVING clauses. The SQL OVER() clause must follow the analytic function. | | window\_partition\_clause | Groups the rows in the input table by a given list of columns or expressions.  The window\_partition\_clause is optional; if you omit it, the rows are not grouped, and the analytic function applies to all rows in the input set as a single partition. | | window\_order\_clause | Sorts the rows specified by the OVER() operator and supplies the ordered set of rows to the analytic function. If the partition clause is present, the window\_order\_clause applies within each partition.  The order clause is optional. If you do not use it, the selection set is not sorted. | | window\_frame\_clause | Used by only some analytic functions. If you include the frame clause in the OVER() statement, which specifies the beginning and end of the window relative to the current row, the analytic function applies only to a subset of the rows defined by the partition clause. This subset changes as the rows in the partition change (called a moving window). |  * 1. Analytic functions require the OVER() clause. However, depending on the function, the window\_frame\_clause and window\_order\_clause might not apply. For example, when used with analytic aggregate functions like SUM(x), you can use the OVER() clause without supplying any of the windowing clauses; in this case, the aggregate returns the same aggregated value for each row of the result set.   2. Analytic functions are allowed only in the SELECT and ORDER BY clauses.   3. Analytic functions can be used in a subquery or in the parent query but cannot be nested; for example, the following query is not allowed:   SELECT MEDIAN(RANK() OVER(ORDER BY sal) OVER());   * 1. Analytic functions WHERE, GROUP BY and HAVING operators are technically not part of the analytic function; however, they determine on which rows the analytic functions operate.   2. window\_partition\_clause      1. Optional      2. Divides the rows in the input based on user-provided expressions      3. Similar to the GROUP BY clause except that it returns only one result row per input row      4. If omitted, all input rows are treated as a single partition      5. Syntax:   OVER ( PARTITION BY expression [ , ... ] )   |  |  | | --- | --- | | **Parameter** | **Definition** | | expression | Expression on which to to sort the partition on. May involve columns, constants or an arbitrary expression formed on columns. |  1. window\_order\_clause    1. Sorts the rows specified by the OVER() clause and specifies whether data is sorted in ascending or descending order as well as the placement of null values    2. Syntax:   OVER ( ORDER BY expression [ { ASC | DESC } ]  ... [ NULLS { FIRST | LAST | AUTO } ] [, expression ...] )   |  |  | | --- | --- | | **Parameter** | **Definition** | | expression | Expression on which to sort the partition, which may involve columns, constants, or an arbitrary expression formed on columns. | | ASC | DESC | Specifies the ordering sequence as ascending (default) or descending. | | NULLS { FIRST | LAST | AUTO } | Indicates the position of nulls in the ordered sequence as either first or last. The order makes nulls compare either high or low with respect to non-null values.  If the sequence is specified as ascending order, ASC NULLS FIRST implies that nulls are smaller than other non-null values. ASC NULLS LAST implies that nulls are larger than non-null values. The opposite is true for descending order. If you specify NULLS AUTO, Vertica chooses the most efficient placement of nulls (for example, either NULLS FIRST or NULLS LAST) based on your query. The default is ASC NULLS LAST and DESC NULLS FIRST. |  1. window\_frame\_clause    1. A unique construct known as moving window.    2. It defines which values in the partition are evaluated relative to the current row (the next row for which the analytic function computes results) in terms of       1. Logical intervals (such as time) using the RANGE keyword       2. A physical number of rows before and/or after the current row using the ROWS keyword    3. Syntax:   { ROWS | RANGE }  {  {  BETWEEN  { UNBOUNDED PRECEDING  | CURRENT ROW  | *constant-value* { PRECEDING | FOLLOWING }  }  AND  { UNBOUNDED FOLLOWING  | CURRENT ROW  | *constant-value* { PRECEDING | FOLLOWING }  }  }  |  {  { UNBOUNDED PRECEDING  | CURRENT ROW  | *constant-value* PRECEDING  }  }  }   |  |  | | --- | --- | | **Parameter** | **Definition** | | ROWS | RANGE | The ROWS and RANGE keywords define the window frame type.  ROWS specifies a window as a physical offset and defines the window's start and end point by the number of rows before or after the current row. The value can be INTEGER data type only.  RANGE specifies the window as a logical offset, such as time. The range value must match the window\_order\_clause data type, which can be NUMERIC, DATE/TIME, FLOAT or INTEGER.  **Note**: The value returned by an analytic function with a logical offset is always deterministic. However, the value returned by an analytic function with a physical offset could produce nondeterministic results unless the ordering expression results in a unique ordering. You might have to specify multiple columns in the window\_order\_clause to achieve this unique ordering. | | BETWEEN ... AND | Specifies a start point and end point for the window. The first expression (before AND) defines the start point and the second expression (after AND) defines the end point.  **Note**: If you use the keyword BETWEEN, you must also use AND. | | UNBOUNDED PRECEDING | Specifies a start point and end point for the window. The first expression (before AND) defines the start point and the second expression (after AND) defines the end point.  **Note**: If you use the keyword BETWEEN, you must also use AND. | | UNBOUNDED PRECEDING | Within a partition, indicates that the window frame starts at the first row of the partition. This start-point specification cannot be used as an end-point specification, and the default is RANGE UNBOUNDED PRECEDING AND CURRENT ROW | | UNBOUNDED FOLLOWING | Within a partition, indicates that the window frame ends at the last row of the partition. This end-point specification cannot be used as a start-point specification. | | CURRENT ROW | As a start point, CURRENT ROW specifies that the window begins at the current row or value, depending on whether you have specified ROW or RANGE, respectively. In this case, the end point cannot be *constant-value* PRECEDING.  As an end point, CURRENT ROW specifies that the window ends at the current row or value, depending on whether you have specified ROW or RANGE, respectively. In this case the start point cannot be *constant-value* FOLLOWING. | | *constant-value* {  PRECEDING | FOLLOWING } | For RANGE or ROW:   * If *constant-value* FOLLOWING is the start point, the end point must be constant-value FOLLOWING. * If *constant-value* PRECEDING is the end point, the start point must be constant-value PRECEDING. * If you specify a logical window that is defined by a time interval in NUMERIC format, you might need to use conversion functions.   If you specified ROWS:   * *constant-value* is a physical offset. It must be a constant or expression and must evaluate to an INTEGER data type value. * If *constant-value* is part of the start point, it must evaluate to a row before the end point.   If you specified RANGE:   * *constant-value* is a logical offset. It must be a constant or expression that evaluates to a positive numeric value or an INTERVAL literal. * If *constant-value* evaluates to a NUMERIC value, the ORDER BY column type must be a NUMERIC data type.. * If the *constant-value* evaluates to an INTERVAL DAY TO SECOND subtype, the ORDER BY column type can only be TIMESTAMP, TIME, DATE, or INTERVAL DAY TO SECOND. * If the *constant-value* evaluates to an INTERVAL YEAR TO MONTH, the ORDER BY column type can only be TIMESTAMP, DATE, or INTERVAL YEAR TO MONTH. * You can specify only one expression in the window\_order\_clause. |  1. **Using Event Based Windows, Time Series, Event Series Join, and Pattern Matching**    1. Event Based Windows       1. Can be used to break time series data into windows that border on significant events within the data       2. Two event based functions available:          1. CONDITIONAL\_CHANGE\_EVENT()             1. Assigns an event window number to each row             2. Starts with 0             3. Increments by 1 when argument expression on the current row differs from the previous row             4. Similar to the ROW\_NUMBER analytic function             5. Example:   SELECT ts,  link,  real\_cnt,  CONDITIONAL\_CHANGE\_EVENT(real\_cnt) OVER(ORDER BY ts)  FROM real\_web\_hits  WHERE ts BETWEEN '2013-02-08 12:00:00' AND '2013-02-08 12:30:00'  ORDER BY ts;   * + - 1. CONDITIONAL\_TRUE\_EVENT()          1. Assigns an event window number to each row          2. Starts with 0          3. Increments by 1 when the result of the Boolean argument expression evaluates true          4. Example:   SELECT ts,  link,  real\_cnt,  CONDITIONAL\_TRUE\_EVENT(real\_cnt > 50) OVER(ORDER BY ts)  FROM real\_web\_hits  WHERE ts BETWEEN '2013-02-08 12:00:00' AND '2013-02-08 12:30:00'  ORDER BY ts;   * + 1. Sessionization, a special case of event-based windows, is a feature often used to analyze click streams, such as identifying web browsing sessions from recorded web clicks.        1. The sessionization computation attempts to identify Web browsing sessions from the recorded clicks by grouping the clicks from each user based on the time-intervals between the clicks. If two clicks from the same user are made too far apart in time, as defined by a time-out threshold, the clicks are treated as though they are from two different browsing sessions.        2. Example:   SELECT userId, timestamp,  CONDITIONAL\_TRUE\_EVENT(timestamp - LAG(timestamp) > '30 seconds')  OVER(PARTITION BY userId ORDER BY timestamp) AS session  FROM WebClicks;   * 1. Time Series      1. Time series analytics evaluate the values of a given set of variables over time and group those values into a window (based on a time interval) for analysis and aggregation.      2. One of the columns in a set of time series data must include time      3. **Gap-filling and interpolation (GFI)** functionality can fill in missing data points and add new (missing) data points within a range of known data points to the output using time series aggregate functions and the SQL **TIMESERIES** clause.      4. There are two type of interpolation         1. **Constant Interpolation** - Interpolation scheme in which Vertica computes a new value based on the previous input records. This is the default scheme.         2. **Linear Interpolation** - Interpolation scheme in which Vertica interpolates values in a linear slope based on the specified time slice      5. **Timeseries Aggregate (TSA) functions** evaluate the values of a given set of variables over time and group those values into a window for analysis and aggregation.         1. **TS\_FIRST\_VALUE** - Returns the value at the start of the time slice, where an interpolation scheme is applied if the timeslice is missing, in which case the value is determined by the values corresponding to the previous (and next) timeslices based on the interpolation scheme of constant or linear.         2. **TS\_LAST\_VALUE** - Returns the value at the end of the time slice, where an interpolation scheme is applied if the timeslice is missing, in which case the value is determined by the values corresponding to the previous (and next) timeslices based on the interpolation scheme of constant or linear.   2. Event Series Join      1. Vertica SQL extension that enables the analysis of two series when their measurement intervals don‘t align precisely, such as with mismatched timestamps      2. Compare values from the two series directly, rather than having to normalize the series to the same measurement interval      3. An extension of outer joins, but instead of padding the non-preserved side with NULL values when there is no match, the event series join pads the non-preserved side values that it interpolates from the previous value      4. Uses the INTERPOLATE predicate as part of the ON clause   3. Pattern Matching  1. **Using explain plans and query profiles**    1. Explain Plans (Query Plans)       1. Query sequence indicating data flow during query execution       2. Verify that query is executing optimally and as expected          1. Projections are being used for a query          2. Vertica method being used for join       3. Returns the query plan execution strategy to standard output.       4. Can be generated for SELECT, INSERT and UPDATE statements       5. Cost and row counts are estimates       6. Precede the SQL statement with the EXPLAIN key word to see the plan:   dbadmin=> EXPLAIN SELECT \* FROM dual;  QUERY PLAN  ---------------------------------------------------------------------------------  QUERY PLAN DESCRIPTION:  ------------------------------  EXPLAIN SELECT \* FROM dual;  Access Path:  +-STORAGE ACCESS for dual [Cost: 663, Rows: 10K (NO STATISTICS)] (PATH ID: 1)  | Projection: v\_catalog.dual\_p  | Materialize: dual.dummy  | Execute on: Query Initiator  ------------------------------  -----------------------------------------------  PLAN: BASE QUERY PLAN (GraphViz Format)  -----------------------------------------------  digraph G {  graph [rankdir=BT, label = "BASE QUERY PLAN\nQuery: EXPLAIN SELECT \* FROM dual;\n\nAll Nodes Vector: \n\n node[0]=v\_intersect\_node0001 (initiator) Up\n  node[1]=v\_intersect\_node0002 (executor) Up\n node[2]=v\_intersect\_node0003 (executor) Up\n", labelloc=t, labeljust=l ordering=out]  0[label = "Root \nOutBlk=[UncTuple]", color = "green", shape = "house"];  1[label = "NewEENode \nOutBlk=[UncTuple]", color = "green", shape = "box"];  2[label = "StorageUnionStep: dual\_p\nUnc: Varchar(128)", color = "purple", shape = "box"];  3[label = "ScanStep: dual\_p\ndummy\nUnc: Varchar(128)", color = "brown", shape = "box"];  1->0 [label = "V[0]",color = "black"];  2->1 [label = "0",color = "blue"];  3->2 [label = "0",color = "blue"];  }  (27 rows)   * 1. Query Profiles      1. Shows where time is being spent during query execution      2. Identifies resource pools being utilized via the **RESOURCE\_ACQUISITIONS** table      3. Indicates memory required to process the query      4. The counters and plans from profiled data reflect what really happened during a query      5. Used to examine:  1. Query plan quality issues 2. Projection design issues 3. If the query is network bound 4. How much memory each operator allocated 5. If a query rewrite might speed up the query 6. How many threads are executing for each operator 7. How the data has flowed through each operator at different points in time over the life of the query    * 1. Real-time profiling is always "on". This is for currently executing queries      2. We can explicitly profile queries in the following areas:         1. **Session**—Provides basic session parameters and lock time out data         2. **Query**—Provides general information about queries that ran, such as the query strings used and the duration of queries         3. **Execution Engine**—Provides detailed information about the execution run of each query      3. Use the ENABLE\_PROFILING() function to enable profiling for the current session and DISABLE\_PROFILING() to disable it. Pass in the parameter ‘session’, ‘query’ or ‘ee’      4. To enable profiling for all sessions on all nodes, call the SET\_CONFIG\_PARAMETER() function. Set one of the configurations setting to 1, ‘GlobalSessionProfiling’, ‘GlobalQueryProfiling’ or ‘GlobalEEProfiling’. Set them to 0 to disable      5. Use the CLEAR\_PROFILING() function to clear profiling data. Pass in the parameter ‘session’, ‘query’ or ‘ee’.      6. The V\_MONITOR.EXECUTION\_ENGINE\_PROFILES system table contains available profiling counters for internal operations, as well as user statements         1. You need to know at least the transaction id which can be found in the Vertica log file         2. When you query the table use the TO\_HEX() function on the transaction\_id column when comparing it to the value taken from the log file      7. Query the V\_MONITOR.QUERY\_PLAN\_PROFILES system table to monitor real-time flow of data through a query plan      8. To profile a single statement add the PROFILE keyword to the beginning of the statement. It’ll return a hint query that can be used to find relevant data in the V\_MONITOR.EXECUTION\_ENGINE\_PROFILES system table      9. The QUERY\_PROFILES table provides profiling information for queries      10. The SESSION\_PROFILES table provides profiling information for sessions      11. Profiling data can also show data skews if some nodes are processing more data than others. The rows produced counter in the EXECUTION\_ENGINE\_PROFILES table shows how many rows have been processed by each of the operators. Comparing the rows produced across all nodes for a given operator would reveal if there is a data skew issue.      12. Profiling counters are available in the EXECUTION\_ENGINE\_PROFILES table until the storage quota has been exceeded after query execution has completed if any of the following conditions are true:          1. The query was run via the PROFILE <query> command          2. Systemwide profiling has been enabled through the ENABLE\_PROFILING() function          3. The query took longer than two seconds to run      13. We can label queries so that they are easier to find in the profiling tables. The syntax for query labeling is /\*+label(*label-name*)\*/. The identifier column can be queried for the label. 8. Analyzing an explain plan    1. If you see large amounts of time spent in GroupByHash operators, you might want to revisit the projection designs so that GroupByPipeline can be used. Use the Database Designer for these optimizations. |