Contents

Chapter 1. About DSE architecture........................................................................................................... 5
Chapter 2. DataStax Enterprise 6.0 FAQ.................................................................................................. 7
Chapter 3. Database architecture............................................................................................................. 9
  Architecture in brief................................................................................................................................. 9
  Internode communications (gossip)......................................................................................................... 12
  Data distribution and replication........................................................................................................... 13
    Data distribution overview................................................................................................................ 13
    Consistent hashing.............................................................................................................................. 15
    Virtual nodes...................................................................................................................................... 17
  Data replication..................................................................................................................................... 19
  Partitioners........................................................................................................................................... 20
  Snitches................................................................................................................................................ 20
  Node repair.......................................................................................................................................... 22
    About repair....................................................................................................................................... 22
    NodeSync: Continuous background repair...................................................................................... 23
    Hinted handoff: repair during write path......................................................................................... 23
    Read Repair: repair during read path.............................................................................................. 26
    Anti-entropy repair............................................................................................................................ 26
Chapter 4. Component architecture........................................................................................................ 30
  DSE Analytics...................................................................................................................................... 30
  DSE Search architecture....................................................................................................................... 30
  DSE Graph architecture....................................................................................................................... 33
    When to use DSE Graph................................................................................................................... 34
    DSE Graph, OLTP, and OLAP............................................................................................................ 35
    Comparing DSE Graph and relational databases........................................................................... 48
    Migrating to DSE Graph from a relational database........................................................................ 53
    Migrating to DSE Graph from Apache Cassandra............................................................................ 53
Chapter 5. Database internals.................................................................................................................. 54
  Storage engine....................................................................................................................................... 54
  About reads and writes.......................................................................................................................... 54
    How is data written? ........................................................................................................................... 54
    How is data maintained?..................................................................................................................... 57
Chapter 1. About the DataStax Enterprise architecture

The Architecture Guide provides information on how the DataStax Enterprise database works. A basic understanding of how DataStax Enterprise works and how it differs from a relational database will save you a lot of time when developing your data models, applications, and operating DataStax Enterprise.

To ensure that you get the best experience in using this document, take a moment to look at the Tips for using DataStax documentation. This page provides information on search, navigational aids, and providing feedback.

To get started, DataStax recommends reading Architecture in brief.

Other important topics include:

- Data distribution and replication
- DSE Graph, DSE Search, DSE Analytics architecture
- How the storage engine works
- How the database reads and writes data

FAQ

How do I interact with DataStax Enterprise?

DataStax Enterprise's architecture allows any authorized user to connect to any node in any datacenter and access data using the Cassandra Query Language (CQL 3.4.5 DSE protocol v2). For ease of use, CQL uses a similar syntax to SQL. The most basic way to interact with DataStax Enterprise (DSE) is using the CQL shell, cqlsh. DataStax Enterprise Studio provides an IDE for syntax validation, type checking, validations specific to the domain, and content assistance for CQL and DSE Graph.

How is DataStax Enterprise different from relational databases?

DataStax Enterprise is a distributed and highly available database that uses peer-to-peer communication. Data modeling in DataStax Enterprise is similar to relational databases while differing in key areas to provide blazingly fast interaction. Relational databases use joins between tables for relationships. DataStax Enterprise uses denormalization to achieve more robust querying.

What is NoSQL?

The NoSQL term originally referred to a new generation of databases that shunned SQL for other interfaces. The term NoSQL has recently become a catch-all term for post-relational "not-only SQL" databases that use a method of storage different from a relational, or SQL, database.

What is Apache Spark™ and how is DSE Analytics different?

Apache Spark is an open source analytics project that provides a fast and general engine for large-scale data processing. DataStax Enterprise integrates Apache Spark real-time and batch analytics processing to more easily manage both database and analytics with a single operational system.

What is Apache Solr™ and how is DSE Search different?

Apache Solr is an open source search project that produces a highly reliable, scalable, and distributed search system that provides search for databases. DSE Search integrates Solr to manage search indexes with a persistent store. DSE Search provide enterprises with the ability to perform text search and text analysis.

What operational tools are included with DataStax Enterprise?

- nodetool
- OpsCenter and Lifecycle Manager (LCM)
- dsetool
- Setting system properties during startup
About the DataStax Enterprise architecture

- cqlsh

What developer tools are available?

- DataStax Studio
- DSE Graph Loader
- Gremlin console
- Javadoc
- demos

How do I test DataStax Enterprise

- Planning and testing DataStax Enterprise deployments
- cassandra-stress tool

What kind of hardware do I need to run DataStax Enterprise?

See Planning and testing cluster deployments for hardware requirements. The distributed nature of DataStax Enterprise can actively utilize multiple datacenters across several geographic regions, supporting highly available data under even the most trying circumstances.

How do I install DataStax Enterprise?

You can install DataStax Enterprise in several ways, depending on the purpose of the installation, the type of operating system, and the available permissions. See Which install method should I use?.

How do I configure DataStax Enterprise?

The Admin guide and Dev guide provide information about using virtual nodes; setting up security; storing and accessing data exclusively from memory; setting up distributed data replication from remote clusters; running multiple DataStax Enterprise nodes on a single host machine; automating the movement of data across different types of storage media; plus much more.

How do I upgrade DataStax Enterprise?

The Upgrade Guide provides instructions for upgrading DataStax Enterprise. This guide also provides instructions on how to upgrade from Apache Cassandra to DataStax Enterprise.

What drivers work with DataStax Enterprise?

DataStax drivers come in two types: DataStax drivers for DataStax Enterprise 5.0 and later, and DataStax drivers for Apache Cassandra™. The DataStax drivers are enhanced to ease the development of applications powered by DataStax Enterprise. These drivers support full functionality for DSE, including DSE Graph, unified authentication, and geospatial types.
Chapter 2. DataStax Enterprise 6.0 FAQ

How do I interact with DataStax Enterprise?
DataStax Enterprise’s architecture allows any authorized user to connect to any node in any datacenter and access data using the Cassandra Query Language (CQL 3.4 + DSE enhancements). For ease of use, CQL uses a similar syntax to SQL. The most basic way to interact with DataStax Enterprise (DSE) is using the CQL shell, cqlsh. Using cqlsh, you can create keyspaces and tables, insert and query tables, plus much more. Other ways to interact with DSE are:

- DataStax Enterprise Studio provides an IDE for syntax validation, type checking, validations specific to the domain, and content assistance for CQL and DSE Graph.
- The DataStax Spark Cassandra Connector provides integration for Spark and access to the Spark shell. There is also SQL access with Spark’s ODBC/JDBC support.
- For production, DataStax supplies a number of drivers in various programming languages, so that CQL statements can be passed from client to cluster and back. See DataStax drivers.

What is Apache Spark™ and how is DSE Analytics different?
Apache Spark is an open source analytics project that provides a fast and general engine for large-scale data processing. DataStax Enterprise integrates Apache Spark real-time and batch analytics processing to more easily manage both database and analytics with a single operational system.

What is Apache Solr™ and how is DSE Search different?
Apache Solr is an open source search project that produces a highly reliable, scalable, and distributed search system that provides search for databases. DSE Search integrates Solr to manage search indexes with a persistent store. DSE Search provide enterprises with the ability to perform text search and text analysis.

How is DataStax Enterprise different from relational databases?
DataStax Enterprise is a distributed and highly available database that uses peer-to-peer communication. Data modeling in DataStax Enterprise is similar to relational databases while differing in key areas to provide blazingly fast interaction. Relational databases use joins between tables for relationships. DataStax Enterprise uses denormalization to achieve more robust querying.

What is NoSQL?
The NoSQL term originally referred to a new generation of databases that shunned SQL for other interfaces. The term NoSQL has recently become a catch-all term for post-relational "not-only SQL" databases that use a method of storage different from a relational, or SQL, database.

How do I move data to and from DataStax Enterprise?
See Migrating data to DataStax Enterprise. You can use the COPY command to read CSV data to DSE and write CSV data from DSE to a file system. The sstableloader provides the ability to bulk load external data into a cluster. However, before moving data to DSE, you need to consider how your client application will query the tables, and do data modeling first. The paradigm shift between relational databases and NoSQL means that a straight move of data from an RDBMS database to DataStax Enterprise will fail. See Migrating data to DataStax Enterprise.

What operational tools are included with DataStax Enterprise?

- nodetool
- OpsCenter and Lifecycle Manager (LCM)
- dsetool
- Setting system properties during startup
- cqlsh

What developer tools are available?

- DataStax Studio
DataStax Enterprise 6.0 FAQ

- DSE Graph Loader
- Gremlin console
- Javadoc
- demos

How do I test DataStax Enterprise

- Planning and testing DataStax Enterprise deployments
- cassandra-stress tool

What kind of hardware do I need to run DataStax Enterprise?

See Planning and testing cluster deployments for hardware requirements. The distributed nature of DataStax Enterprise can actively utilize multiple datacenters across several geographic regions, supporting highly available data under even the most trying circumstances.

How do I install DataStax Enterprise?

You can install DataStax Enterprise in several ways, depending on the purpose of the installation, the type of operating system, and the available permissions. See Which install method should I use?.

How do I configure DataStax Enterprise?

The Admin guide and Dev guide provide information about using virtual nodes; setting up security; storing and accessing data exclusively from memory; setting up distributed data replication from remote clusters; running multiple DataStax Enterprise nodes on a single host machine; automating the movement of data across different types of storage media; plus much more.

How do I upgrade DataStax Enterprise?

The Upgrade Guide provides instructions for upgrading DataStax Enterprise. This guide also provides instructions on how to upgrade from Apache Cassandra to DataStax Enterprise.

What drivers work with DataStax Enterprise?

DataStax drivers come in two types: DataStax drivers for DataStax Enterprise 5.0 and later, and DataStax drivers for Apache Cassandra™. The DataStax drivers are enhanced to ease the development of applications powered by DataStax Enterprise. These drivers support full functionality for DSE, including DSE Graph, unified authentication, and geospatial types.

<table>
<thead>
<tr>
<th>DataStax drivers for DSE 5.0 and later</th>
<th>DataStax drivers for Apache Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ driver</td>
<td>C/C++ driver</td>
</tr>
<tr>
<td>C# driver</td>
<td>C# driver</td>
</tr>
<tr>
<td>Java driver</td>
<td>Java driver</td>
</tr>
<tr>
<td>Node.js driver</td>
<td>Node.js driver</td>
</tr>
<tr>
<td>Python driver</td>
<td>PHP driver</td>
</tr>
<tr>
<td>Ruby driver</td>
<td>Python driver</td>
</tr>
<tr>
<td></td>
<td>Ruby driver</td>
</tr>
</tbody>
</table>
Chapter 3. Understanding the database architecture

DataStax strongly recommends taking time to read these topics. Understanding how the database works and how it differs from a relational database will save you a lot of time when setting up and operating DataStax Enterprise in a production environment.

Architecture in brief

This page provides essential information for understanding and using DataStax Enterprise. Because DataStax Enterprise differs from a relational database, you will save yourself a lot of time by reading and understanding each section on this page.

How DataStax Enterprise works

DataStax Enterprise, powered by the best distribution of Apache Cassandra™, seamlessly integrates your code, allowing applications to utilize a breadth of techniques to produce mobile apps and online applications. DSE is designed to handle big data workloads across multiple nodes with no single point of failure. DSE architecture is based on the understanding that system and hardware failures can and do occur.

DSE addresses the problem of failures by employing a peer-to-peer distributed system across homogeneous nodes where data is distributed among all nodes in the cluster. Each node frequently exchanges state information about itself and other nodes across the cluster using peer-to-peer gossip communication protocol. A sequentially written commit log on each node captures write activity to ensure data durability. Data is then indexed and written to an in-memory structure, called a memtable, which resembles a write-back cache.

Each time the memory structure is full, the data is written to disk in an SSTables data file. All writes are automatically partitioned and replicated throughout the cluster. DSE periodically consolidates SSTables using compaction, which discards obsolete data marked for deletion with a tombstone. A tombstone is a marker in a row that indicates a column will be deleted. During compaction, marked columns are deleted. To ensure all data across the cluster stays consistent, various repair mechanisms are employed.

The DSE database is a partitioned row store database, where rows are organized into tables with a (required) primary key. The database's architecture allows any authorized user to connect to any node in any datacenter and access data using the CQL language. For ease of use, CQL uses a similar syntax to SQL and works with table data. Developers can access CQL through CQL shell (cqlsh) commands, DataStax Studio, and by using drivers for application languages. Typically, a cluster has one keyspace per application composed of many different tables.

Client read or write requests can be sent to any node in the cluster. When a client connects to a node with a request, that node serves as the coordinator node for that particular client operation. The coordinator acts as a proxy between the client application and the nodes that own the data being requested. The coordinator determines which nodes in the ring should get the request based on how the cluster is configured.

Key structures

Node
Where you store your data. It is the basic database infrastructure component.

Cluster
A group of distributed nodes for storing data. A cluster can have a single node, single datacenter, or multiple datacenters.

Datacenter
A group of related nodes configured together within a cluster for replication purposes. A datacenter can be a physical datacenter or virtual datacenter. Using separate datacenters prevents transactions from being impacted by other workloads and lowers latency. Depending on the replication factor, data can be written to multiple datacenters. Datacenters must never span physical locations.

Each datacenter usually contains only one node type. The node types are:
Understanding the database architecture

- **Transactional**: Previously referred to as a Cassandra node.
- **DSE Graph**: A graph database for managing, analyzing, and searching highly-connected data.
- **DSE Analytics**: Integration with Apache Spark.
- **DSE Search**: Integration with Apache Solr.
- **DSE SearchAnalytics**: DSE Search queries within DSE Analytics jobs.

**Replication**

The process of storing copies of data on multiple nodes. Replication ensures reliability and fault tolerance. The number of copies is set by the replication factor.

**Commit log**

All data is written first to the commit log for durability. After all its data has been flushed to SSTables, it can be archived, deleted, or recycled.

**SSTable**

A sorted string table (SSTable) is an immutable data file to which the database writes memtables periodically. SSTables are append only and stored on disk sequentially and maintained for each database table.

**tombstone**

A marker in a row that indicates a column will be deleted. During compaction, marked columns are deleted.

**CQL Table**

A collection of ordered columns fetched by table row. A table consists of columns and has a primary key.

**Key components for configuring DataStax Enterprise**

**Gossip**

A peer-to-peer communication protocol to discover and share location and state information about the other nodes in a DSE cluster. Gossip information is persisted locally by each node to use immediately when a node restarts.

**Partitioner**

A partitioner distributes data evenly across the nodes in the cluster for load balancing. Specifically, a partitioner determines which node receives the first replica of a piece of data, and how to distribute other replicas across other nodes in the cluster. Each row of data is uniquely identified by a primary key, which may be the same as its partition key, but which may also include other clustering columns. A partitioner is a hash function that derives a token from the primary key of a row. The partitioner uses the token value to determine which nodes in the cluster receive the replicas of that row. The Murmur3Partitioner is the default partitioning strategy for new DSE clusters and the right choice for new clusters in almost all cases.

**Replication factor**

Replication is the process of storing copies of data on multiple nodes. Replication ensures reliability and fault tolerance. The number of copies is set by the replication factor. A replication factor of 1 means that there is only one copy of each row on one node. A replication factor of 2 means two copies of each row, where each copy is on a different node. All replicas are equally important; there is no primary or master replica. You define the replication factor for each datacenter. Generally, set the replication strategy greater than one, but no more than the number of nodes in the cluster.

**Replica placement strategy**

A replication strategy determines which nodes to place replicas on. The first replica of data is simply the first copy; it is not unique in any sense. The NetworkTopologyStrategy is highly recommended for most deployments because it is much easier to expand to multiple datacenters when required by future expansion. When creating a keyspace, you must define the replica placement strategy and the number of replicas you want.

**Snitch**

A snitch maps from the IP addresses of nodes to physical and virtual locations, such as racks and datacenters. Snitches inform the database about the network topology so that requests are routed efficiently and allows the database to distribute replicas by grouping machines into datacenters and racks.
You must configure a **snitch** when you create a cluster. All snitches use a dynamic snitch layer, which monitors performance and chooses the best replica for reading. The **dynamic snitch** is enabled by default and recommended for use in most deployments. Configure dynamic snitch thresholds for each node in the cassandra.yaml configuration file.

The default **DseSimpleSnitch** does not recognize datacenter or rack information. Use it for single-datacenter deployments or single-zone in public clouds. The **GossipingPropertyFileSnitch** is recommended for production. It defines a node's datacenter and rack and uses gossip for propagating this information to other nodes.

**cassandra.yaml configuration file**

The main configuration file for setting the initialization properties for a cluster, caching parameters for tables, properties for tuning and resource utilization, timeout settings, client connections, backups, and security. By default, a node is configured to store the data it manages in a directory set in the cassandra.yaml file.

In a production cluster deployment, DataStax recommends changing the commitlog directory to a different disk drive from the data file directories.

**dse.yaml configuration file**

The configuration file for DSE Advanced Security, DSE Search, DSE Graph, and DSE Analytics.

**System keyspace table properties**

You set storage configuration attributes on a per-keyspace or per-table basis programmatically or using a client application, such as CQL shell (cqlsh).

**Loading and unloading data**

Use the DataStax Bulk Loader tool to efficiently load and unload DSE data. The tool migrates data into DSE from another DSE or Apache Cassandra™ cluster.

- Unloads data from any Cassandra 2.1 or later data source
- Loads data into DSE 5.0 or later
- Supports CSV and JSON formats

**Basic concepts for data modeling**

**Design the data model**

The design of the data model is based on the queries you want to perform, not on modeling entities and relationships like you do for relational databases.

**Keyspace**

The outermost grouping of data, similar to a schema in a relational database. All tables belong to a keyspace. A keyspace is the defining container for replication.

**Table**

A table stores data based on a primary key, which consists of a partition key and optional clustering columns. Materialized views can also be added for high cardinality data.

- A partition key defines the node on which the data is stored, and divides data into logical groups. Define partition keys that evenly distribute the data and also satisfy specific queries. Query and write requests across multiple partitions should be avoided if possible.
- A clustering column defines the sort order of rows within a partition. When defining a clustering column, consider the purpose of the data. For example, retrieving the most recent transactions, sorted by date, in descending order.
- Materialized views are tables built from another table's data with a new primary key and new properties. Queries are optimized by the primary key definition. Data in the materialized view is automatically updated by changes to the source table.

In earlier versions of DataStax Enterprise and Apache Cassandra™, a column family was synonymous in many respects, to a table.
Understanding the database architecture

More information on data modeling

- Data modeling concepts in the CQL documentation.
- Using CQL in the CQL documentation.
- Creating a materialized view in the CQL documentation.
- Getting Started with Time Series Data Modeling white paper.
- Getting Started with User Profile Data Modeling white paper.
- Become a Super Modeler webinar.
- The Data Model is Dead, Long Live the Data Model webinar.

Internode communications (gossip)

DataStax Enterprise uses a protocol called gossip to discover location and state information about the other nodes participating in a cluster.

What is gossip?

Gossip is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know about. The gossip process runs every second and exchanges state messages with up to three other nodes in the cluster. The nodes exchange information about themselves and about the other nodes that they have gossiped about, so all nodes quickly learn about all other nodes in the cluster. A gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.

Preventing problems in gossip communications

To prevent problems in gossip communications, be sure to use the same list of seed nodes for all nodes in a cluster. This is most critical the first time a node starts up. By default, a node remembers other nodes it has gossiped with between subsequent restarts. The seed node designation has no purpose other than bootstrapping the gossip process for new nodes joining the cluster. Seed nodes are not a single point of failure, nor do they have any other special purpose in cluster operations beyond the bootstrapping of nodes.

Making every node a seed node is not recommended because of increased maintenance and reduced gossip performance. Gossip optimization is not critical, but it is recommended to use a small seed list (approximately three nodes per datacenter).

About failure detection and recovery

Failure detection is a method for locally determining from gossip state and history when a node in the system is down or has come back up. The DataStax Enterprise database uses this information to avoid routing client requests to unreachable nodes whenever possible. (The database can also avoid routing to poorly performing nodes, through the dynamic snitch.)

<table>
<thead>
<tr>
<th>Row</th>
<th>Partition key</th>
<th>Clustering column</th>
<th>Column</th>
<th>CQL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SSN</td>
<td>Name</td>
<td>Age</td>
<td>F</td>
</tr>
<tr>
<td>555-55-5555</td>
<td>Min</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>444-44-4444</td>
<td>Pazival</td>
<td>17</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>222-22-2222</td>
<td>Dorcus</td>
<td>38</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

create table people (ssn text, name text, age int, gender text, primary key(ssn,name)) ;
The gossip process tracks state from other nodes both directly (nodes gossiping directly to it) and indirectly (nodes communicated about secondhand, third-hand, and so on). Rather than using a fixed threshold for marking failing nodes, the database uses an accrual detection mechanism to calculate a per-node threshold. The threshold takes into account network performance, workload, and historical conditions. During gossip exchanges, every node maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster.

To adjust the sensitivity of the failure detector, configure the `phi_convict_threshold` property. Lower values increase the likelihood that an unresponsive node will be marked as down. Use the default value for most situations, but increase it to 10 or 12 for Amazon EC2 (due to frequently encountered network congestion). In unstable network environments (EC2 at times), raising the value to 10 or 12 helps prevent false failures. Values higher than 12 and lower than 5 are not recommended.

Node failures can result from various causes such as hardware failures and network outages. Node outages are often transient but can last for extended periods. Because a node outage rarely signifies a permanent departure from the cluster, it does not automatically result in permanent removal of the node from the ring. Other nodes will periodically try to re-establish contact with failed nodes to see if they are back up. To permanently change a node's membership in a cluster, you must explicitly add or remove nodes from a cluster.

When a node comes back online after an outage, it may have missed writes for the replica data it maintains. Repair mechanisms exist to recover missed data, such as hinted handoffs and manual repair with `nodetool repair`. The length of the outage determines which repair mechanism is used to make the data consistent.

### Data distribution and replication

In DataStax Enterprise, data distribution and replication go together. Data is organized by table and identified by a primary key, which determines which node the data is stored on. Replicas are copies of rows, which are stored on multiple nodes to ensure reliability and fault tolerance. When data is first written, it is also referred to as a replica. All replicas are equally important; there is no primary or master replica.

Features affecting replication include:

- **Virtual nodes** assign data ownership to physical machines.
- **Partitioners** distribute the data across the cluster.
- **Replication strategy** determines the replicas for each row of data.
- **Snitches** define the topology information that the replication strategy uses to place replicas.

### Data distribution overview

In DataStax Enterprise (DSE), the total amount of data managed by the cluster is represented as a ring. The ring is divided into ranges equal to the number of nodes, with each node being responsible for one or more ranges of the data. Before a node can join the ring, it must be assigned a token. The token value determines the node's position in the ring and its range of data. Table data is partitioned across the nodes based on the row key. To determine the node where the first replica of a row will live, DSE locates the node with a token value greater than that of the row key. Each node is responsible for the region of the ring between the node itself and its predecessor, excluding the predecessor's token value. With the nodes sorted in token order, the last node is considered the predecessor of the first node.

For example, consider a simple four-node cluster where all of the row keys managed by the cluster were numbers in the range of 0 to 100. Each node is assigned a token that represents a point in this range. In this simple example, the token values are 0, 25, 50, and 75. The first node with token 0 is responsible for the wrapping range (76-0). The node with the lowest token also accepts row keys less than the lowest token and more than the highest token.
Figure 2: Four-node cluster

Data range 4
(76+ wrapping range)

Data range 1
(1-25)

Data range 3
(51-75)

Data range 2
(26-50)
DSE provides several types of partitioners, which should be assigned to your cluster. An initial_token value is assigned to each node so that each node is responsible for roughly an equal amount of data.

**Consistent hashing**

Consistent hashing allows data distribution across a cluster to minimize reorganization when nodes are added or removed. Consistent hashing partitions data based on the partition key. For an explanation of partition keys and primary keys, see the Data modeling example.

For example, if you have the following data:

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>car</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>jim</td>
<td>36</td>
<td>camaro</td>
<td>M</td>
</tr>
<tr>
<td>carol</td>
<td>37</td>
<td>bmw</td>
<td>F</td>
</tr>
<tr>
<td>johnny</td>
<td>12</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>suzy</td>
<td>10</td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>

The database assigns a hash value to each partition key:

<table>
<thead>
<tr>
<th>Partition key</th>
<th>Murmur3 hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>jim</td>
<td>-2245462676723223822</td>
</tr>
<tr>
<td>carol</td>
<td>7723358927203680754</td>
</tr>
<tr>
<td>johnny</td>
<td>-6723372854036780675</td>
</tr>
<tr>
<td>suzy</td>
<td>1168604627387940318</td>
</tr>
</tbody>
</table>

Each node in the cluster is responsible for a range of data based on the hash value.
Figure 3: Hash values in a four node cluster

Datacenter alpha
DataStax Enterprise places the data on each node according to the value of the partition key and the range that the node is responsible for. For example, in a four node cluster, the data in this example is distributed as follows:

<table>
<thead>
<tr>
<th>Node</th>
<th>Start range</th>
<th>End range</th>
<th>Partition key</th>
<th>Hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-9223372036854775808</td>
<td>-4611686018427387904</td>
<td>johnny</td>
<td>-6723372854036780875</td>
</tr>
<tr>
<td>2</td>
<td>-4611686018427387903</td>
<td>-1</td>
<td>jim</td>
<td>-22456326767232323822</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4611686018427387903</td>
<td>suzy</td>
<td>1168604627387940318</td>
</tr>
<tr>
<td>4</td>
<td>4611686018427387904</td>
<td>9223372036854775807</td>
<td>carol</td>
<td>7723358927203680754</td>
</tr>
</tbody>
</table>

Virtual nodes

Virtual nodes (vnodes) distribute data across nodes at a finer granularity than can be easily achieved using a single-token architecture. Virtual nodes simplify many tasks in DataStax Enterprise:

- Tokens are automatically calculated and assigned to each node.
- A cluster is automatically rebalanced when adding or removing nodes. When a node joins the cluster, it assumes responsibility for an even portion of data from the other nodes in the cluster. If a node fails, the load is spread evenly across other nodes in the cluster.
- Rebuilding a dead node is faster because it involves every other node in the cluster.
- The proportion of vnodes assigned to each machine in a cluster can be assigned, so smaller and larger computers can be used in building a cluster.

To convert an existing single-token architecture cluster to vnodes, see Enabling virtual nodes on an existing production cluster.

Distributing data using vnodes

In single-token architecture clusters, you must calculate and assign a single token to each node in a cluster. Each token determines the node's position in the cluster (or, ring) and its portion of data according to its hash value. Vnodes allow each node to own a large number of small partition ranges distributed throughout the cluster. Although vnodes use consistent hashing to distribute data, using them doesn't require token generation and assignment.
Understanding the database architecture

Figure 4: Virtual vs single-token architecture

Ring without virtual nodes

Node 1
- A
- F

Node 2
- B

Node 3
- C

Node 4
- D
- C

Node 5
- E

Node 6
- F
- D

Ring with virtual nodes

Node 1
- B
- E

Node 2
- A
- P

Node 3
- K
- G
- C
- N

Node 4
- M

Node 5
- E

Node 6
- H
- F
- J
- I
The top portion of the graphic shows a cluster without vnodes. In the single-token architecture paradigm, each node is assigned a single token that represents a location in the ring. Each node stores data determined by mapping the partition key to a token value within a range from the previous node to its assigned value. Each node also contains copies of each row from other nodes in the cluster. For example, if the replication factor is 3, range E replicates to nodes 5, 6, and 1. A node owns exactly one contiguous partition range in the ring space.

The bottom portion of the graphic shows a ring with vnodes. Within a cluster, vnodes are randomly selected and non-contiguous. The placement of a row is determined by the hash of the partition key within many smaller partition ranges belonging to each node.

While vnodes provide considerable operational benefits, it's important to keep in mind that the number of vnodes assigned to a given node can impact cluster-wide operations. For instance, when the number of vnodes are increased, so are the number of repairs that need to be run during a repair cycle, thus increasing full cluster repair times. In addition, as the number of vnode token ranges increase, features such as DSE Search that span token ranges can see their performance suffer.

**Data replication**

DataStax Enterprise (DSE) stores replicas on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines the particular nodes where replicas are placed. The total number of replicas across the cluster is referred to as the replication factor. A replication factor of 1 means that there is only one copy of each row in the cluster. A replication factor of 2 means two copies of each row, where each copy is on a different node.

Never use a replication factor of 1. If the node containing the row goes down, the row cannot be retrieved.

All replicas are equally important; there is no primary or master replica. As a general rule, the replication factor should not exceed the number of nodes in the cluster. However, you can increase the replication factor and then add the desired number of nodes later. The replication factor also depends on the node type. Before using DSE in production, you must set the replication factors for analytics keyspaces and security keyspaces.

Two replication strategies are available:

**SimpleStrategy**

Use only for development and if you only have a single datacenter and one rack. However, if you ever intend to have more than one datacenter, use the NetworkTopologyStrategy instead. SimpleStrategy places the first replica on a node determined by the partitioner. Additional replicas are placed on the next nodes clockwise in the ring without considering topology (rack or datacenter location).

**NetworkTopologyStrategy**

Highly recommended for most deployments because it is much easier to expand your cluster to multiple datacenters. This strategy specifies how many replicas you want in each datacenter. NetworkTopologyStrategy places replicas in the same datacenter by walking the ring clockwise until reaching the first node in another rack. It also attempts to place replicas on distinct racks because nodes in the same rack (or similar physical grouping) can fail at the same time due to power, cooling, or network issues.

When deciding how many replicas to configure in each datacenter, the two primary considerations are satisfying reads locally without incurring cross datacenter latency, and failure scenarios. The two most common ways to configure multiple-datacenter clusters are:

- Two replicas in each datacenter: This configuration tolerates the failure of a single node per replication group and still allows local reads at a consistency level of ONE.
- Three replicas in each datacenter: This configuration tolerates either the failure of one node per replication group at a strong consistency level of LOCAL_QUORUM or multiple node failures per datacenter using consistency level ONE.

Asymmetrical replication groupings are also possible. For example, you can have three replicas in one datacenter to serve real-time application requests and use a single replica elsewhere for running analytics.

Replication strategy is defined per keyspace, and is set during keyspace creation. To set up a keyspace, see Creating a keyspace.
Understanding the database architecture

For more replication strategy options, see Changing keyspace replication strategy.

**Partitioners**

A partitioner determines how data is distributed across the nodes in the cluster (including replicas).

A partitioner is a function for deriving a token representing a partition key, typically by hashing. Each row of data is then distributed across the cluster by the value of the token. Any IPartitioner may be used, including your own, as long as it is in the classpath.

The default Murmur3Partitioner uses tokens to help assign equal portions of data to each node and evenly distribute data from all the tables throughout the ring or other grouping, such as a keyspace. This is true even if the tables use different partition keys, such as user names or timestamps. Because each part of the hash range receives an equal number of partitions on average, the read and write requests to the cluster are evenly distributed and load balancing is simplified. For more information, see Consistent hashing.

Whether or not you need assign tokens to each node depends on the type of architecture:

- Virtual nodes: Use either the allocation algorithm or the random selection algorithm to specify the number of tokens distributed to nodes within the datacenter. All systems in the datacenter must use the same algorithm.
- Single-token architecture: To ensure data is evenly divided across the nodes in the cluster, you must enter values in the initial_token parameter in the cassandra.yaml file for each node. To determine the token value, see Calculating tokens for single-token architecture nodes.

**Types of partitioners**

Be aware that partitioners are not compatible with each other. Data partitioned with one partitioner cannot be easily converted to the other partitioner.

DataStax Enterprise supplies the following partitioners:

**Murmur3Partitioner (default)**

Uniformly distributes data across the cluster based on MurmurHash hash values. This hashing function creates a 64-bit hash value of the partition key with a possible range from \(-2^{63}\) to \(+2^{63}-1\). This partitioner is the right choice for new clusters in almost all cases and is 3 to 5 times more performant than the RandomPartitioner.

When using this partitioner, you can page through all rows using the token function in a CQL query.

**RandomPartitioner**

This partitioner is included for backwards compatibility. It uniformly distributes data evenly across the nodes using an MD5 hash value of the row key. The possible range of hash values is from 0 to \(2^{127}-1\). Because it uses a cryptographic hash, which isn't required by the database, it takes longer to generate the hash value than the Murmur3Partitioner.

When using this partitioner, you can page through all rows using the token function in a CQL query.

**ByteOrderedPartitioner**

This partitioner is included for backwards compatibility. This partitioner orders rows lexically by key bytes. It is not recommended because it requires significant administrative overhead to load balance the cluster, sequential writes can cause hot spots, and balancing for one table can result in uneven distribution for another table in the same cluster.

DataStax recommends using the default partitioner, Murmur3Partitioner. You can change the partitioner in the cassandra.yaml file.

**Snitches**

A snitch determines which datacenters and racks nodes belong to. Snitches inform the database about the network topology so requests are routed efficiently, and enable DataStax Enterprise (DSE) to distribute replicas by grouping machines into datacenters and racks. Specifically, the replication strategy places the replicas based on the information provided by the new snitch. All nodes in a cluster must use the same snitch. DSE strives to not have more than one replica on the same rack (which is not necessarily a physical location).

If you change snitches, you may need to perform additional steps because the snitch affects where replicas are placed. See Switching snitches.
Dynamic snitching

By default, all snitches use a dynamic snitch layer that monitors read latency and, when possible, routes requests away from poorly-performing nodes. The dynamic snitch is enabled by default and is recommended for use in most deployments. For information on how this works, see the Dynamic snitching in Cassandra: past, present, and future blog post. Configure dynamic snitch thresholds for each node in the cassandra.yaml configuration file. Also, see the properties listed under About failure detection and recovery.

Types of snitches

DataStax Enterprise (DSE) provides the following types of snitches:

DseSimpleSnitch (default)
Use this snitch only for development deployments. This snitch does not recognize datacenter or rack information. When using this snitch, define the keyspace to use SimpleStrategy and specify a replication factor.

GossipingPropertyFileSnitch
This snitch is recommended for production. It uses rack and datacenter information for the local node defined in the cassandra-rackdc.properties file and propagates this information to other nodes via gossip. To configure, see cassandra-rackdc.properties file.

Ec2Snitch
Use this snitch for simple cluster deployments on Amazon EC2 where all nodes in the cluster are within a single region.
In EC2 deployments, the region name is treated as the datacenter name, and availability zones are treated as racks within a datacenter. For example, if a node is in the us-east-1 region, us-east is the datacenter name and 1 is the rack location. (Racks are important for distributing replicas, but not for datacenter naming.) Because private IPs are used, this snitch does not work across multiple regions.
To configure, see Ec2Snitch.

Ec2MultiRegionSnitch
Use this snitch for deployments on Amazon EC2 where the cluster spans multiple regions.
You must configure settings in both the cassandra.yaml file and the property file (cassandra-rackdc.properties) used by the Ec2MultiRegionSnitch. To configure, see Ec2MultiRegionSnitch.

GoogleCloudSnitch
Use this snitch for DSE deployments on Google Cloud Platform across one or more regions. The region is treated as a datacenter, and the availability zones are treated as racks within the datacenter.
All communication occurs over private IP addresses within the same logical network. To configure, see Configuring the Google Cloud Platform snitch.

CloudstackSnitch
Use this snitch for Apache Cloudstack environments. Because zone naming is free-form in Apache Cloudstack, this snitch uses the <country> <location> <az> notation. To configure, see CloudstackSnitch.

PropertyFileSnitch
This snitch determines proximity as determined by rack and datacenter. It uses the network details located in the cassandra-topology.properties file. When using this snitch, define datacenter names using a standard convention, and make sure that the datacenter names correlate to the name of your datacenters in the keyspace definition. Every node in the cluster should be described in the cassandra-topology.properties file, which must be exactly the same on every node in the cluster.

RackInferringSnitch
This snitch determines the proximity of nodes by datacenter and rack, which are assumed to correspond to the second and third octet of the node's IP address, respectively. It is best used as an example for writing a custom snitch class (unless this format happens to match your deployment conventions).
Node repair in DataStax Enterprise

About repair

Over time, data in a replica can become inconsistent with other replicas due to the distributed nature of the database. Node repair corrects the inconsistencies so that all nodes have the same and most up-to-date data. Node repair is an important part of regular maintenance for every DataStax Enterprise (DSE) cluster. Use database settings or various DSE tools to configure each type of repair.

DataStax Enterprise provides the following repair processes. The following links detail when to use and how to configure each repair type.

NodeSync: Continuous background repair

The NodeSync service has low overhead and provides consistent performance and virtually eliminates manual efforts to run repair operations in a DataStax cluster. The nodesync table property continuously verifies the data consistency across replicas in the background. NodeSync is enabled on a per table basis.

Hinted Handoff

If a node becomes unable to receive a particular write, the write’s coordinator node preserves the data to be written as a set of hints. When the node comes back online, the coordinator hands off hints so that the node can catch up with the required writes.

Read Repair

During the read path, a query assembles data from several nodes. The coordinator node for the read compares the data from each replica node. If any replica node has outdated data, the coordinator node sends it the most recent version. The scope of this type of repair depends on the keyspace’s replication factor. During a read, the database collects only enough replica data to satisfy the replication factor, and only performs read repair on nodes that participate in that read operation.

Anti-Entropy Repair
DataStax Enterprise provides the `nodetool repair` tool to ensure data consistency across replicas; it compares the data across all replicas and then updates the data to the most recent version. Use `nodetool repair` as part of your regular maintenance routine.

DataStax recommends stopping repair operations during topology changes; the Repair Service does this automatically. Repairs running during a topology change are likely to error when it involves moving ranges.

**NodeSync: Continuous background repair**

NodeSync is an easy to use continuous background repair that has low overhead and provides consistent performance.

- Continuously validates that data is in sync on all replica
- Always running but low impact on cluster performance
- Fully automatic, no manual intervention needed
- Completely replace anti-entropy repairs

**NodeSync service**

By default, each node runs the NodeSync service. The service is idle unless it has something to validate. NodeSync is enabled/disabled on each table individually. The service continuously validates local data ranges for NodeSync-enabled tables and repairs any inconsistency found. The local data ranges are split into small segments, which act as validation save points. Segments are prioritized in order to try to meet the per-table deadline target.

**Segments**

A segment is a small local token range of a table. It is created by NodeSync by recursively splitting local ranges in half a certain number of times (depth). The depth calculated using the total table size, assuming equal distribution of data, create segments that cover no more than 200 MB. Token ranges can be no smaller than a single partition, so very large partitions can result in segments larger than the configured size.

**Validation process and status**

After a segment is selected for validation, NodeSync reads the entirety of the data it covers from all replica (using paging), checks for inconsistencies, and repairs if needed. When a node validates a segment, it “locks” it in a system table to avoid work duplication by other nodes. It is not a race-free lock; there is a possibility of duplicated work which saves the complexity and cost of true distributed locking. Segment validation is saved on completion in the `system_distributed.nodesync_status` table, which is used internally for resuming on failure, prioritization, segment locking, and by tools; not meant to be read directly.

**Hinted handoff: repair during write path**

On occasion, a node becomes unresponsive while data is being written. Reasons for unresponsiveness include hardware problems, network issues, or overloaded nodes that experience long garbage collection (GC) pauses. By design, hinted handoff inherently allows DataStax Enterprise (DSE) to continue performing the same number of writes even when the cluster is operating at reduced capacity.

If the failure detector marks a node as down and hinted handoff is enabled in the cassandra.yaml file, missed writes are stored by the coordinator node for a period of time. In DataStax Enterprise 5.0 and later, the hint is stored in a local hints directory on each node for improved replay. The hint consists of the following information:

- Target ID for the downed node
- Hint ID that is a time UUID for the data
- Message ID that identifies the DataStax Enterprise version
- The data itself as a blob

Hints are flushed to disk every ten seconds, reducing the staleness of the hints. When gossip discovers a node is back online, the coordinator replays each remaining hint to write the data to the newly-returned node, then
Understanding the database architecture

deletes the hint file. If a node is down for longer than the value configured in the max_hint_window_in_ms parameter (three hours by default), the coordinator stops writing new hints.

The coordinator also checks every ten minutes for hints corresponding to writes that timed out during an outage too brief for the failure detector to notice through gossip. If a replica node is overloaded or unavailable, and the failure detector has not yet marked the node as down, then expect most or all writes to that node to fail after the timeout triggered by write_request_timeout_in_ms (2 seconds by default). The coordinator returns a TimeOutException error, and the write will fail. However, a hint will be stored. If several nodes experience brief outages simultaneously, substantial memory pressure can build up on the coordinator. The coordinator tracks how many hints it is currently writing. If the number of hints increases too much, the coordinator refuses writes and throws the OverloadedException error.

Consistency level ONE

The consistency level of a write request affects whether hints are written and a write request subsequently fails. If the cluster consists of two nodes, A and B, with a replication factor of one, each row is stored on only one node. Suppose node A is the coordinator, but goes down before a row K is written to it with a consistency level of ONE. In this case, the consistency level specified cannot be met, and because node A is the coordinator, it cannot store a hint. Node B cannot write the data because it has not received the data as the coordinator, and a hint has not been stored. The coordinator checks the number of replicas that are up and will not attempt to write the hint if the consistency level specified by a client cannot be met. In this case, the coordinator will return an UnavailableException error. The write request fails and the hint is not written.

In general, the recommendation is to have enough nodes in the cluster and a replication factor sufficient to avoid write request failures. For example, consider a cluster consisting of three nodes, A, B, and C, with a replication factor of three. When a row K is written to the coordinator (node A in this case), even if node C is down, the consistency level of ONE or QUORUM can be met. Why? Both nodes A and B will receive the data, so the consistency level requirement is met. A hint is stored for node C and written when node C comes up. In the meantime, the coordinator can acknowledge that the write succeeded.
**Figure 6:**

```
Figure 6:
```

**Consistency level ANY**

For applications that want DataStax Enterprise to accept writes when all the normal replicas are down and consistency level ONE cannot be satisfied, the database provides consistency level ANY. ANY guarantees that the write is durable and readable after an appropriate replica target becomes available and receives the hint replay.
Understanding the database architecture

Nodes that die might have stored undelivered hints, because any node can be a coordinator. The data on the dead node will be stale after an extended outage. If a node has been down for an extended period of time, run a manual repair.

At first glance, it seems that hinted handoff eliminates the need for manual repair, but this is not true because hardware failure is inevitable and has the following ramifications:

- Loss of the historical data necessary to tell the rest of the cluster exactly what data is missing.
- Loss of hints-not-yet-replayed from requests that the failed node coordinated.

When removing a node from the cluster by decommissioning the node or by using the `nodetool removenode` command, the database automatically removes hints targeting the node that no longer exists and removes hints for dropped tables.

For more explanation about hint storage, see the What's Coming to Cassandra in 3.0: Improved Hint Storage and Delivery blog post. For basic information, see the Modern hinted handoff blog post.

**Read Repair: repair during read path**

When a read query encounters inconsistent results at a consistency level greater than `ONE` or `LOCAL_ONE`, DSE initiates a read repair. Such read repairs run in the foreground and block application operations until the repair process is complete.

Read queries with a consistency level of `ONE` or `LOCAL_ONE` do not block application operations since a data mismatch must exist to trigger the read repair, and, since only one replica is queried, no comparison occurs, therefore no mismatch.

In more detail:

1. The coordinator node asks one replica for data and the others for a digest of their data.
2. If there is a mismatch in the data returned to the coordinator from the replicas, a read is requested from all replicas involved in the query (dictated by the consistency level) and the results are merged.
3. If a single replica doesn't have all of the latest data for each column, a new record is assembled by mixing and matching columns from different replicas.
4. After determining the latest version, the record is written back to only the replicas involved in the request.

For example, in the case of a `LOCAL_QUORUM` read with a replication factor of three, two replicas are queried, so only those two replicas are repaired.

Read repair does not propagate expired tombstones, nor does it consider expired tombstones when actually repairing data. That means that if there is tombstoned data that has not been propagated to all replica nodes before `gc_grace_seconds` has expired, that data may continue to be returned as live data.

Read repair cannot be performed on tables that use `DateTieredCompactionStrategy` (DTCS) - Deprecated, due to the method of checking timestamps used in DTCS compaction.

**Anti-entropy repair**

Anti-entropy node repairs are important for every DataStax Enterprise (DSE) cluster. Frequent data deletions and downed nodes are common causes of data inconsistency. Use anti-entropy repair for routine maintenance and when a cluster needs fixing by running the `nodetool repair`. For continuous background repair, use NodeSync: Continuous background repair.

**How does anti-entropy repair work?**

DSE accomplishes anti-entropy repair using Merkle trees, which are binary hash trees whose leaves are hashes of the individual key values (similar to Dynamo and Riak). Anti-entropy is a process of comparing the data of all replicas and updating each replica to the newest version. DSE has two phases to the process:

1. Build a Merkle tree for each replica
2. Compare the Merkle trees to discover differences
The leaf of a DSE Merkle tree is the hash of a row value. Each parent node higher in the tree is a hash of its respective children. Because higher nodes in the Merkle tree represent data further down the tree, Cassandra can check each branch independently without requiring the coordinator node to download the entire data set. For anti-entropy repair, DSE uses a compact tree version with a depth of 15 ($2^{15} = 32$K leaf nodes). For example, for a node containing one million partitions with one damaged partition, about 30 partitions are streamed, which is the number that fall into each of the leaves of the tree. DSE works with smaller Merkle trees because they require less storage memory and can be transferred more quickly to other nodes during the comparison process.
After the initiating node receives the Merkle trees from the participating peer nodes, the initiating node compares every tree to every other tree. If a difference is detected, the differing nodes exchange data for the conflicting range(s), and the new data is written to SSTables. The comparison begins with the top node of the Merkle tree. If no difference is detected, then the data requires no repair. If a difference is detected, then the process proceeds to the left child node and compares and then the right child node. When a node is found to differ, inconsistent data exists for the range that pertains to that node. All data that corresponds to the leaves below that Merkle tree node will be replaced with new data. For any given replica set, DSE performs validation compaction on only one replica at a time.

Merkle tree building is resource intensive, stressing disk I/O and using memory. Some of the options discussed here help lessen the impact on the cluster performance.

Run the `nodetool repair` command on either a specified node or on all nodes if a node is not specified. The node that initiates the repair becomes the coordinator node for the operation. To build the Merkle trees, the coordinator node determines peer nodes with matching ranges of data. A major, or validation, compaction is triggered on the peer nodes. The validation compaction reads and generates a hash for every row in the stored column families, adds the result to a Merkle tree, and returns the tree to the initiating node. Merkle trees use hashes of the data, because in general, hashes will be smaller than the data itself. The Repair in Cassandra blog post discusses this process in more detail.

To switch from incremental to full repair, see Changing repair strategies.

**Sequential vs Parallel repair**

Sequential repair takes action on one node after another. Parallel repair repairs all nodes with the same replica data at the same time. Datacenter parallel (-dc-par) combines sequential and parallel by simultaneously running a sequential repair in all datacenters; one node in each datacenter runs repair until the repair is complete.

Sequential repair takes a snapshot of each replica. Snapshots are hardlinks to existing SSTables. They are immutable and require almost no disk space. The snapshots are active while the repair proceeds, then the database deletes them. When the coordinator node finds discrepancies in the Merkle trees, the coordinator node makes required repairs from the snapshots. For example, for a table in a keyspace with a replication factor of three (RF=3) and replicas A, B and C, the `repair` command takes a snapshot of each replica immediately and then repairs each replica from the snapshots sequentially (using snapshot A to repair replica B, then snapshot A to repair replica C, then snapshot B to repair replica C).

Parallel repair constructs the Merkle tables for all nodes in all datacenters at the same time. It works on nodes A, B, and C all at once. During parallel repair, the dynamic snitch processes queries for this table using a replica in the snapshot that is not undergoing repair. Use parallel repair to complete the repair quickly or when you have operational downtime that allows the resources to be completely consumed during the repair.

Datacenter parallel repair, unlike sequential repair, constructs the Merkle tables for all datacenters at the same time. Therefore, no snapshots are required (or generated).

Sequential repair is the default in DSE 4.8 and earlier. Parallel repair is the default for DSE 5.0 and later.
Chapter 4. Component architecture

DSE Analytics

Use DSE Analytics to analyze extremely large data sets. DSE Analytics provides distributed storage, real-time, streaming, and batch analytics with built-in integration with Apache Spark™, a distributed, parallel data processing engine.

DSE Analytics features

No single point of failure

DSE Analytics supports a peer-to-peer, distributed cluster for running Spark jobs. Being peers, any node in the cluster can load data files, and any analytics node can assume the responsibilities of Spark Master.

Spark Master management

DSE Analytics provides automatic Spark Master management.

Analytics without ETL

Using DSE Analytics, you run Spark jobs directly against data in the database. You can perform real-time and analytics workloads at the same time without one workload affecting the performance of the other. Starting some cluster nodes as Analytics nodes and others as pure transactional real-time nodes automatically replicates data between nodes.

DataStax Enterprise file system (DSEFS)

DSEFS (DataStax Enterprise file system) is a fault-tolerant, general-purpose, distributed file system within DataStax Enterprise. It is designed for use cases that need to leverage a distributed file system for data ingestion, data staging, and state management for Spark Streaming applications (such as checkpointing or write-ahead logging). DSEFS is similar to HDFS, but avoids the deployment complexity and single point of failure typical of HDFS. DSEFS is HDFS-compatible and is designed to work in place of HDFS in Spark and other systems.

DSE Analytics Solo

DSE Analytics Solo datacenters are devoted entirely to DSE Analytics processing, for deployments that require separation of analytics jobs from transactional data.

Integrated security

DSE Analytics uses the advanced security features of DSE, simplifying configuration and deployment.

AlwaysOn SQL

AlwaysOn SQL is a highly-available service that provides JDBC and ODBC interfaces to applications accessing DSE Analytics data.

DSE Search architecture

In a distributed environment, the data is spread over multiple nodes. Deploy DSE Search nodes in their own datacenter to run DSE Search on all nodes.

Data is written to the database first, and then indexes are updated next.

When you update a table using CQL, the search index is updated. Indexing occurs automatically after an update. Writes are durable. All writes to a replica node are recorded in memory and in a commit log before they are acknowledged as a success. If a crash or server failure occurs before the memory tables are flushed to disk, the commit log is replayed on restart to recover any lost writes.
DSE Search terms

In DSE Search, there are several names for an index of documents on a single node:

- A search index (formerly referred to as a search core)
Component architecture

- A collection
- One shard of a collection

See the following table for a mapping between database and DSE Search concepts.

Table 1: Relationship between the database and DSE Search concepts

<table>
<thead>
<tr>
<th>Database</th>
<th>Search single node environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Search index (core) or collection</td>
</tr>
<tr>
<td>Row</td>
<td>Document</td>
</tr>
<tr>
<td>Primary key</td>
<td>Unique key</td>
</tr>
<tr>
<td>Column</td>
<td>Field</td>
</tr>
<tr>
<td>Node</td>
<td>n/a</td>
</tr>
<tr>
<td>Partition</td>
<td>n/a</td>
</tr>
<tr>
<td>Keyspace</td>
<td>n/a</td>
</tr>
</tbody>
</table>

How DSE Search works

- Each document in a search index is unique and contains a set of fields that adhere to a user-defined schema.
- The schema lists the field types and defines how they should be indexed.
- DSE Search maps search indexes to tables.
- Each table has a separate search index on a particular node.
- Solr documents are mapped to rows, and document fields to columns.
- A shard is indexed data for a subset of the data on the local node.
- The keyspace is a prefix for the name of the search index and has no counterpart in Solr.
- Search queries are routed to enough nodes to cover all token ranges.
  
  # The query is sent to all token ranges to get all possible results.
  
  # The search engine considers the token ranges that each node is responsible for, taking into account the replication factor (RF), and computes the minimum number of nodes that is required to query all ranges.

- On DSE Search nodes, the shard selection algorithm for distributed queries uses a series of criteria to route sub-queries to the nodes most capable of handling them. The shard routing is token aware, but is not limited unless the search query specifies a specific token range.

- With replication, a node or search index contains more than one partition (shard) of table (collection) data. Unless the replication factor equals the number of cluster nodes, the node or search index contains only a portion of the data of the table or collection.

DSE Search path

This section provides an overview of the DSE Search path, and how Solr integrates with Cassandra:

1. A row mutation is performed in Cassandra.
2. A thread in the Thread Per Core (TPC) architecture processes the mutation.
3. The mutation is forwarded to the secondary index API.
4. A Lucene document is built from the latest full row in the backing table.
5. The document is placed in the Lucene RAM buffer.
6. Control is returned to Cassandra.
7. The Cassandra write operation is completed.

Note:
- The RAM buffer is flushed when a commit is performed.
  - Commits occur when the RAM buffer is full, or a soft commit or hard commit is performed.
  - Soft commits are triggered by the auto soft commit timer.
  - Hard commits are triggered by a memtable flush on the base Cassandra table.
- The Lucene documents are flushed to disk into a Lucene segment.
- Part of the flush process ensures that for a given document identifier, only one live document exists. Therefore, any duplicate older documents are deleted.
- Lucene merges segments periodically in a similar way that Cassandra performs compaction.

**DSE Graph architecture**

DSE Graph is composed of a number of components that provide online transaction processing (OLTP) and online analytical processing (OLAP) capabilities. The web-based DataStax Studio allows users to process queries to the graph database as well as visualize the data. The Gremlin console accesses the database through a command-line interface.

OLTP applications interact with DSE Graph through the DSE Server or the TinkerPop 3 Gremlin Server. OLAP applications interact with DSE Graph through the TinkerPop 3 GraphComputer, Spark, and a DataStax Spark connector to DataStax Enterprise.

The DSE database is an embedded component of DataStax Enterprise (DSE), and comprises the storage backend for DSE Graph. DSE Search provides the index backend for DSE Graph.
When to use DSE Graph

DataStax Enterprise, powered by the best distribution of Apache Cassandra™, seamlessly integrates your code, allowing applications to utilize a breadth of techniques to produce a mobile app or online applications. Each commercial extension, such as DSE Graph, inherits the benefits of Apache Cassandra as part of DataStax Enterprise’s database while adding the ability to adapt to enterprise needs with other models such as graph or JSON data storage.

DSE Graph is the optimal choice for storing data when the following characteristics are required:

- Comprehensive data model
- Data is database centric with single query
- Entities and relationships are queried
• Application is read heavy

The DSE database is the optimal choice for storing data when the following characteristics are required:

• Heavy denormalization
• Data is application centric with multiple queries
• Individual entities are queried
• Application is write heavy

DSE Graph is an extension of the DSE database that reaps benefits if the data is highly connected. The graph data model is simple to understand. The connectedness of the data reveals both depth and breadth to the relationships between entities. DSE Graph uses query optimization that automatically parallelizes as much of the query as possible to increase performance. Graph index structures are used to create optimal entry points for queries before starting a graph traversal. Graph partitioning handles vertices with extreme connectedness to prevent hotspots during graph traversal. All of these aspects of DSE Graph take advantage of the underlying DSE database that is used to store DSE Graph data.

DSE Graph, OLTP, and OLAP

Online transactional processing (OLTP) is characterized by a large number of short, online transactions for very fast query processing. OLTP is typically used for data entry and retrieval with transaction-oriented applications. Online analytical processing (OLAP) is typically used to perform multidimensional analysis of data, doing complex calculations on aggregated historical data.

OLTP applications require sub-second response times, whereas OLAP applications take much longer to finish queries. Graph databases are a random access data system. In these databases, OLAP traversals do a linear scan of all vertices in the graph. Conversely, OLTP traversals are localized to a particular subgraph of the global graph. OLTP traversals leverage indexes to “jump” in to a particular vertex in the graph before starting a scan on the subgraph.

OLTP queries

OLTP queries are best for questions that require access to a limited subset of the entire graph. OLTP queries use filters to limit the number of vertices that will be walked to find answers. DSE Graph co-locates vertices with their edges and adjacent neighbors. When a subgraph is specified in a traversal using indexes, the number of requests to disk are reduced to locate and write the requested subgraph to memory. Once in memory, the traversal performs a link walk from vertex to vertex along the edges.

OLAP queries

OLAP queries are best for questions that must access a significant portion of the data stored in a graph. Using the previous method to evaluate OLAP queries will not be efficient, so a different process is used. When OLAP queries are processed, the entire graph is interpreted as a sequence of star graphs, each composed of a single vertex, along with its properties, incident edges, and the edges’ properties. The star graphs are linearly processed, jumping from one star graph to the next until all star graphs are processed and an aggregation of the discovered data is completed.

Principles for writing graph traversals

Understanding these underlying principles can lead to writing better graph traversals to query the graph data. A simple example illustrates the differences. Using the food graph, the query is “How many recipes has Julia Child created?”

Consider the following graph traversal:

```java
g.V().in().has('name','Julia Child').count() ===>6
```

This traversal completes the following processing:

1. Looks at all vertices.
2. Walks the incoming edges.
3. Finds the adjacent vertices that have the property key of name and property value of Julia Child.

4. Counts the number of vertices.

This graph traversal is a classic OLAP traversal, which must touch all vertices and does not use indexing. The count returned includes all vertices with edges to Julia Child, and not just the recipes, so as shown later, the count is incorrect and too high.

Consider the number of elements that must be traversed to complete this query. DSE Graph has profiling that aids in analyzing the traversal:

```ruby
gremlin> g.V().in().has('name','Julia Child').count().profile()
```
index-query
 1.344
  \_usesCache=false
  \_statement=SELECT "community_id", "member_id" FROM "DSEQuickStart"."ingredient_p"
  WHERE "\~\~vertex_exists" = ? LIMIT ? ALLOW FILTERING; with params (java.lang.Boolean) true,
  (java.lang.Integer) 5000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}

index-query
 0.938
  \_usesCache=false
  \_statement=SELECT "community_id", "member_id" FROM "DSEQuickStart"."meal_p"
  WHERE "\~\~vertex_exists" = ? LIMIT ? ALLOW FILTERING; with params (java.lang.Boolean) true,
  (java.lang.Integer) 5000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}

index-query
 4.136
  \_usesCache=false
  \_statement=SELECT * FROM "DSEQuickStart"."author_e"
  WHERE "community_id" = ? AND "member_id" = ? LIMIT ?
  ALLOW FILTERING; with params (java.lang.Integer) 588941056,
  (java.lang.Long) 0, (java.lang.Integer) 5000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}

index-query
 1.291
  \_usesCache=false
  \_statement=SELECT "community_id", "member_id" FROM "DSEQuickStart"."reviewer_p"
  WHERE "\~\~vertex_exists" = ? LIMIT ? ALLOW FILTERING; with params (java.lang.Boolean) true,
  (java.lang.Integer) 5000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}

DsegVertexStep(IN, vertex) 78 78
query-optimizer
95.721 61.90
  \_condition=((true) & direction = IN)
vertex-query
0.305
  \_usesCache=false
  \_statement=SELECT * FROM "DSEQuickStart"."author_e"
  WHERE "community_id" = ? AND "member_id" = ?
  ALLOW FILTERING; with params (java.lang.Integer) 588941056,
  (java.lang.Long) 0, (java.lang.Integer) 5000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}
Component architecture

```
c=true}
  \_isPartitioned=false
  \_usesIndex=false
  vertex-query
    0.558
    \_useCache=false
    \_statement=SELECT * FROM "DSEQuickStart"."author_e" WHERE "community_id" = ? AND "member_id" = ? LIMIT ?
      ALLOW FILTERING; with params (java.lang.Integer) 1432048000,
      (java.lang.Long) 1, (java.lang.Integer) 50000
    \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
      fallbackConsistency=Optional.empty,
      waitForSchemaAgreement=true, async=true}
  \_isPartitioned=false
  \_usesIndex=false
  vertex-query
    1.146
    \_useCache=false
    \_statement=SELECT * FROM "DSEQuickStart"."author_e" WHERE "community_id" = ? AND "member_id" = ? LIMIT ?
      ALLOW FILTERING; with params (java.lang.Integer) 153541376,
      (java.lang.Long) 1, (java.lang.Integer) 50000
    \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
      fallbackConsistency=Optional.empty,
      waitForSchemaAgreement=true, async=true}
  \_isPartitioned=false
  \_usesIndex=false
  query-setup
    0.941
    \_isFitted=false
    \_isSorted=true
    \_isScan=false
    query-setup
      0.015
      \_isFitted=false
      \_isSorted=true
      \_isScan=false
  vertex-query
    1.966
    \_useCache=false
    \_statement=SELECT * FROM "DSEQuickStart"."author_e" WHERE "community_id" = ? AND "member_id" = ? LIMIT ?
      ALLOW FILTERING; with params (java.lang.Integer) 138026496,
      (java.lang.Long) 0, (java.lang.Integer) 50000
    \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
      fallbackConsistency=Optional.empty,
      waitForSchemaAgreement=true, async=true}
  \_isPartitioned=false
  \_usesIndex=false
  query-setup
    0.015
    \_isFitted=false
    \_isSorted=true
    \_isScan=false
    query-setup
      0.013
      \_isFitted=false
```
\_isSorted=true  
\_isScan=false

query-setup
0.016
\_isFitted=false  
\_isSorted=true  
\_isScan=false

NoOpBarrierStep(2500)  
2.877  1.86

HasStep({name.=(Julia Child)})  
25.242  16.32

CountGlobalStep  
1.859  1.20

>TOTAL

The time each step takes depends on caching and other factors. For the purposes of this discussion, ignore the times reported. The `profile()` method now includes CQL commands that are executed due to Gremlin commands.

**Figure 10: Studio profile output for Traversal 1**

Looking at the first step, all vertices in the graph are traversed. This graph is very small, so the number of vertices is negligible compared to production graphs. In the next step, the traversal must find all incoming edges to the vertices. Again, for a small graph, the number of edges is negligible, but in production graphs, edges can number in the millions to billions. Now, the adjacent vertices are filtered for the property key information specified, narrowing the number of vertices to 6. The last two steps accomplish the count and profiling metrics.
Specifying an edge label

Now consider a modification to the original traversal that specifies the edge label for the incoming edges:

\[
g.V().in('created').has('name','Julia Child').count()
\]

\[\Rightarrow 3\]

This modified traversal still looks at all vertices, but in walking the incoming edges, it is limited to those that are labeled as `created`. The following profile shows an improved picture:

```sql
query g.V().in('created').has('name','Julia Child').count().profile()
```

<table>
<thead>
<tr>
<th>Step</th>
<th>Count</th>
<th>Traversers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DsegGraphStep (vertex,[])</td>
<td>61</td>
<td>61</td>
</tr>
<tr>
<td>query-optimizer</td>
<td>1.760</td>
<td></td>
</tr>
<tr>
<td>_condition=((label = FridgeSensor</td>
<td>label = author</td>
<td>label = book</td>
</tr>
<tr>
<td>_isFitted=true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_isSorted=false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_isScan=true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index-query</td>
<td>1.139</td>
<td></td>
</tr>
<tr>
<td>_usesCache=false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_statement=SELECT &quot;city_id&quot;, &quot;sensor_id&quot; FROM &quot;DSEQuickStart&quot;.&quot;FridgeSensor_p&quot; WHERE &quot;~~vertex_exists&quot; = ? LIMIT ? ALLOW FILTERING; with params (java.lang.Boolean) true, (java.lang.Integer) 50000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty, fallbackConsistency=Option.al.empty, waitForSchemaAgreement=true, async=true}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index-query</td>
<td>2.012</td>
<td></td>
</tr>
<tr>
<td>_usesCache=false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_statement=SELECT &quot;community_id&quot;, &quot;member_id&quot; FROM &quot;DSEQuickStart&quot;.&quot;author_p&quot; WHERE &quot;~~vertex_exists&quot; = ? LIMIT ? ALLOW FILTERING; with params (java.lang.Boolean) true, (java.lang.Integer) 50000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty, fallbackConsistency=Option.al.empty, waitForSchemaAgreement=true, async=true}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index-query</td>
<td>0.549</td>
<td></td>
</tr>
<tr>
<td>_usesCache=false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty, fallbackConsistency=Option.al.empty, waitForSchemaAgreement=true, async=true}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
index-query
  0.849
  \_usesCache=false
  \_statement=SELECT "community_id", "member_id" FROM "DSEQuickStart\"."ingredient_p" WHERE "~~vertex_exists" = ? LIMIT ? ALLOW FILTERING; with params (java.lang.Boolean) true,
  (java.lang.Integer) 5000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}

index-query
  0.887
  \_usesCache=false
  \_statement=SELECT "community_id", "member_id" FROM "DSEQuickStart\"."meal_p" WHERE "~~vertex_exists" = ?
  LIMIT ? ALLOW FILTERING; with params (java.lang.Boolean) true,
  (java.lang.Integer) 50000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}

index-query
  0.889
  \_usesCache=false
  \_statement=SELECT "community_id", "member_id" FROM "DSEQuickStart\"."recipe_p" WHERE "~~vertex_exists" = ?
  LIMIT ? ALLOW FILTERING; with params (java.lang.Boolean) true,
  (java.lang.Integer) 50000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}

index-query
  0.499
  \_usesCache=false
  \_statement=SELECT "community_id", "member_id" FROM "DSEQuickStart\"."reviewer_p" WHERE "~~vertex_exists" = ?
  LIMIT ? ALLOW FILTERING; with params (java.lang.Boolean) true,
  (java.lang.Integer) 50000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}

DsegVertexStep(IN,[created],vertex)                                    8           8

103.458    78.62
query-optimizer
0.618
\_condition=((label = created) & (true)) & direction = IN

vertex-query
0.261
\_usesCache=false
\_statement=SELECT * FROM "DSEQuickStart\"."author_e" WHERE "community_id" = ? AND "member_id" = ? AND "~~edge_label_id" = ? LIMIT ? ALLOW FILTERING; with params
  (java.lang.Integer) 1432048000, (java.lang.Long) 1, (java.lang.Integer) 65577, (java.lang.Integer) 50000
  \_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
  fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
  waitForSchemaAgreement=true, async=true}
Component architecture

```java
\_isPartitioned=false
\_usesIndex=false
vertex-query 0.200
\_usesCache=false
\_statement=SELECT * FROM "DSEQuickStart"."author_e" WHERE "community_id" = ? AND "member_id" = ? AND "~~ edge_label_id" = ? LIMIT ? ALLOW FILTERING; with params
(java.lang.Integer) 153541376, (java.lang.Long) 65577, (java.lang.Integer) 50000
\_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty, fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty, waitForSchemaAgreement=true, async=true}
\_isPartitioned=false
\_usesIndex=false
query-setup 0.017
\_isFitted=true
\_isSorted=true
\_isScan=false
vertex-query 6.140
\_usesCache=false
\_statement=SELECT * FROM "DSEQuickStart"."author_e" WHERE "community_id" = ? AND "member_id" = ? AND "~~ edge_label_id" = ? LIMIT ? ALLOW FILTERING; with params
(java.lang.Integer) 588941056, (java.lang.Long) 0, (java.lang.Integer) 65577, (java.lang.Integer) 50000
\_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty, fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty, waitForSchemaAgreement=true, async=true}
\_isPartitioned=false
\_usesIndex=false
query-setup 0.017
\_isFitted=true
\_isSorted=true
\_isScan=false
vertex-query 0.201
\_usesCache=false
\_statement=SELECT * FROM "DSEQuickStart"."author_e" WHERE "community_id" = ? AND "member_id" = ? AND "~~ edge_label_id" = ? LIMIT ? ALLOW FILTERING; with params
(java.lang.Integer) 771301632, (java.lang.Long) 0, (java.lang.Integer) 65577, (java.lang.Integer) 50000
\_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty, fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty, waitForSchemaAgreement=true, async=true}
\_isPartitioned=false
\_usesIndex=false
query-setup 0.012
\_isFitted=true
\_isSorted=true
\_isScan=false
vertex-query 0.173
\_usesCache=false
```
As with the original traversal, the first step still finds all the vertices. In the next step, however, the number of edges walked is significantly decreased. However, in a production graph, finding all the vertices in the entire
Component architecture

graph will take a long time. The third step now reflects the true answer for how many recipes Julia Child has created; in the first traversal, other incoming edges for Julia Child's books were included in the count.
This graph traversal is still an OLAP traversal that touch all vertices and does not use indexes.

Specifying the vertex label

What effect does specifying the vertex label have on improving the traversal?

```java
g.V().hasLabel('recipe').in().has('name','Julia Child').count()
```

This modified traversal now is limited to the `recipe` vertices, but walks all incoming edges. The profile shows a somewhat better picture:

```sql
❯ gremlin> g.V().hasLabel('recipe').in().has('name','Julia Child').count().profile()
===>Traversal Metrics
<table>
<thead>
<tr>
<th>Step</th>
<th>Count</th>
<th>Traversers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td>% Dur</td>
<td></td>
</tr>
<tr>
<td>DsegGraphStep([-label.=('recipe'))]</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>query-optimizer</td>
<td>0.241</td>
<td></td>
</tr>
<tr>
<td>query-setup</td>
<td>0.187</td>
<td></td>
</tr>
<tr>
<td>index-query</td>
<td>1.225</td>
<td></td>
</tr>
<tr>
<td>vertex-query</td>
<td>0.896</td>
<td></td>
</tr>
<tr>
<td>DsegVertexStep(IN,vertex)</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>query-optimizer</td>
<td>0.150</td>
<td></td>
</tr>
<tr>
<td>query-setup</td>
<td>0.047</td>
<td></td>
</tr>
<tr>
<td>vertex-query</td>
<td>0.896</td>
<td></td>
</tr>
</tbody>
</table>
```

DSE 6.0 Architecture Guide Previous DSE version Latest 6.0 patch: 6.0.11
```
_isPartitioned=false
__usesIndex=false

vertex-query
1.415
__usesCache=false
__statement=SELECT * FROM "DSEQuickStart"."recipe_e" WHERE "community_id" = ? AND
"member_id" = ? LIMIT ?
ALLOW FILTERING; with params (java.lang.Integer) 96517120,
(java.lang.Long) 1, (java.lang.Integer) 50000
__options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
waitForSchemaAgreement=true, async=true}

vertex-query
2.846
__usesCache=false
__statement=SELECT * FROM "DSEQuickStart"."recipe_e" WHERE "community_id" = ? AND
"member_id" = ? LIMIT ?
ALLOW FILTERING; with params (java.lang.Integer) 1598713728,
(java.lang.Long) 1, (java.lang.Integer) 50000
__options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
waitForSchemaAgreement=true, async=true}

query-setup
0.038
__isFitted=false
__isSorted=true
__isScan=false

vertex-query
0.364
__usesCache=false
__statement=SELECT * FROM "DSEQuickStart"."recipe_e" WHERE "community_id" = ? AND
"member_id" = ? LIMIT ?
ALLOW FILTERING; with params (java.lang.Integer) 1146421632,
(java.lang.Long) 1, (java.lang.Integer) 50000
__options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty,
fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty,
waitForSchemaAgreement=true, async=true}

query-setup
0.014
__isFitted=false
__isSorted=true
__isScan=false

vertex-query
0.431
__usesCache=false
__statement=SELECT * FROM "DSEQuickStart"."recipe_e" WHERE "community_id" = ? AND
"member_id" = ? LIMIT ?
ALLOW FILTERING; with params (java.lang.Integer) 384373760,
(java.lang.Long) 2, (java.lang.Integer) 50000
```
Figure 12: Studio profile output for Traversal 3

A limited number of vertices are found in the first step. A number of edges are walked. However, in a production graph, finding even a limited number of vertices will take some time without indexing, and the number of edges walked could be quite large.

This graph traversal is still an OLAP traversal that does not use indexes. Although this traversal narrows the query by limiting the vertex label initially, an index is not used to find the starting point for the traversal.

**Using an edge label plus a vertex label**

Indexes are identified by vertex label and property key. The following graph traversal twists the direction of the query:

```java
g.V().has('author', 'name', 'Julia Child').outE('created').count()
```
This traversal starts at a single vertex by specifying both vertex label `author` and a specific property key and value `Julia Child`, and walks only the outgoing edges that have an edge label `created`.

```sql
gremlin> g.V().has('author','name','Julia Child').outE('created').count().profile()
```

---

**Traversal Metrics**

<table>
<thead>
<tr>
<th>Step</th>
<th>Count</th>
<th>Traversers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DsegGraphStep (label=(author), name=(Julia Child))</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>29.049</td>
<td>84.45</td>
</tr>
<tr>
<td>query-optimizer</td>
<td>7.673</td>
<td></td>
</tr>
<tr>
<td>_condition=(((label = author) &amp; (true)) &amp; name = Julia Child)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>query-setup</td>
<td>0.033</td>
<td></td>
</tr>
<tr>
<td>_isFitted=true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_isSorted=false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index-query</td>
<td>17.694</td>
<td></td>
</tr>
<tr>
<td>_indexType=Secondary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_usesCache=false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_statement=SELECT &quot;community_id&quot;, &quot;member_id&quot; FROM &quot;DSEQuickStart&quot;.&quot;author_p&quot; WHERE &quot;name&quot; = ? LIMIT ?; with params (java.lang.String) Julia Child, (java.lang.Integer) 50000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty, fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty, waitForSchemaAgreement=true, async=true}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DsegVertexStep (OUT,[created],edge)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>5.265</td>
<td>15.31</td>
</tr>
<tr>
<td>query-optimizer</td>
<td>0.200</td>
<td></td>
</tr>
<tr>
<td>_condition=(((label = created) &amp; (true)) &amp; direction = OUT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vertex-query</td>
<td>0.586</td>
<td></td>
</tr>
<tr>
<td>_usesCache=false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_options=Options{consistency=Optional[ONE], serialConsistency=Optional.empty, fallbackConsistency=Optional.empty, pagingState=null, pageSize=-1, user=Optional.empty, waitForSchemaAgreement=true, async=true}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CountGlobalStep</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>0.081</td>
<td>0.24</td>
</tr>
</tbody>
</table>
A single vertex starts the traversal. An edge label filters the edges.

This graph traversal is an OLTP traversal. An index on the vertex label `author` and property key `name` can be used to start the traversal directly at an indexed vertex. This example results in a single vertex, but queries that use indexing to limit the starting point to even several vertices will be more efficient than a linear scan that must check all vertices in the graph. Thus, a subgraph, or portion of the graph is traversed.

The key to creating OLTP graph traversals is considering how the graph will be traversed. Use of indexing is critical to the success of fast transactional processing. The profiling tool included with DSE Graph is valuable to analyzing how the traversal performs.

For information on running OLAP queries using Spark, see DSE Graph and Graph Analytics.

**Comparing DSE Graph and relational databases**

DSE Graph is a good choice for data and queries that are highly complex. Graph databases and relational databases can both store data. Both database types can be queried to retrieve filtered results. However, the storage methods, scalability, and indexing techniques that each database type uses is very different.

DSE Graph is recommended for the following use cases:

- Fraud detection
- Recommendation engines
- IT network and device management
- Inventory management
- Master data management

**Storage**

Both relational databases and DSE Graph store data in tables. In a relational database, a key is defined to retrieve data, and a foreign key is required to link tables that share a relationship. Data is normalized, which means that the data is stored so that no duplication occurs. To write complex queries, several tables must be accessed to join and retrieve data.

DSE Graph stores data in the DSE database tables, and the relationship between data is embedded in the model. Data storage in a graph database can be compared to a pre-joined relational database, with built-in data relationships, so foreign keys are unnecessary. Data is retrieved by traversing the graph, so time-consuming and error-prone JOIN operations are not required.
Scaling
An added bonus for graph databases is the ability to scale database distribution. A graph database query starts with a particular vertex and filters the graph based on the query requirements. Because the retrieval generally involves filtering subset(s) of the entire graph, the graph can be partitioned. The query can run in parallel for some steps of the traversal, operating on either a single or small number of nodes in the DataStax Enterprise (DSE) cluster. Because of interconnected tables, successful queries require accessing many tables in a relational databases. Joining data is costly, which makes relational databases hard to scale out and distribute.

Indexing
Indexing techniques between graph databases and relational databases are distinct and can affect performance markedly. Consider the query for looking up all buyers that purchased an Xbox One. In a relational database, three tables would be accessed to answer the query:

- Each buyer in the buyers table
- The item being queried in the items table
- The buyer and item in the buyers-items table, created through a JOIN operation

Figure 14:

In contrast, a graph database requires only one index lookup for the item, and then traverses the graph to each buyer vertex to complete the query.
The graph database query will run in near-constant time. The relational database query, if a balanced tree (B-tree) index is assumed, runs in \(O(\log n)\) time, where \(n\) is the number of records in the table for each foreign key that must be referenced. Even a one-deep query will greatly impact performance; if a query goes to a more deeply nested depth, the performance gap will further increase.
Writing complex queries in DSE Graph is much simpler than in relational databases using SQL. Consider the query required for a recommendation engine about products: the SQL query on the left is vastly more complex than the Gremlin query on the right.

Although many developers are familiar with SQL, this example illustrates the difficulty of writing a complex query, even for experts. By contrast, Gremlin query can intuitively be read from left to right, and each step follows from the previous step.
SELECT TOP 950 [t14].[ProductName] 
FROM (SELECT COUNT(*) AS [value] 
    FROM [customers] AS [t0] 
    CROSS APPLY (SELECT [t9].[ProductName] 
        FROM [orders] AS [t1] 
        CROSS JOIN [order details] AS [t2] 
        INNER JOIN [products] AS [t3] 
        ON [t3].[ProductID] = [t2].[ProductID] 
        CROSS JOIN [order details] AS [t4] 
        INNER JOIN [orders] AS [t5] 
        ON [t5].[OrderID] = [t4].[OrderID] 
        LEFT JOIN [customers] AS [t6] 
        ON [t6].[CustomerID] = [t5].[CustomerID] 
        CROSS JOIN [orders] AS [t7] 
        CROSS JOIN [order details] AS [t8] 
        INNER JOIN [products] AS [t9] 
        ON [t9].[ProductID] = [t8].[ProductID]) 
WHERE NOT EXISTS(SELECT NULL AS [EMPTY] 
    FROM [order] AS [t10] 
    CROSS JOIN [order details] AS [t11] 
    INNER JOIN [products] AS [t12] 
    ON [t12].[ProductID] = [t11].[ProductID] 
    WHERE [19].[ProductID] = [t12].[ProductID] 
    AND [10].[CustomerID] = [0].[CustomerID] 
    AND [11].[OrderID] = [10].[OrderID] 
    AND [16].[CustomerID] <> [0].[CustomerID] 
    AND [11].[CustomerID] <> [0].[CustomerID] 
    AND [12].[OrderID] <> [11].[OrderID] 
    AND [14].[ProductID] <> [3].[ProductID] 
    AND [17].[CustomerID] <> [17].[CustomerID] 
    AND [18].[OrderID] <> [17].[OrderID]) AS [t13] 

VS.

g.V('customerId','ALFKI').as('customer') 
  .out('ordered').out('contains').out('is').as('products') 
  .in('is').in('contains').in('ordered').except('customer') 
  .out('ordered').out('contains').out('is').except('products') 
  .groupCount().cap().orderMap(T.decr[0..<5].productName
Migrating to DSE Graph from a relational database

Migrating a relational database to DSE Graph requires analyzing how the data is stored in the relational tables. Consider the following factors:

- A relational table can represent either a vertex or an edge. If the relational table has an independent primary key, then the data is stored as a vertex in DSE Graph. If the relational table is a join table, then the data is stored as an edge in DSE Graph.

- Relational table columns that are not foreign keys are stored as property keys in DSE Graph.

- Foreign keys in relational tables indicate relationship. Determine the use of referencing foreign keys to discover relationships between vertices and edges in DSE Graph.

A good starting point to build the graph data model is an entity-relationship (ER) diagram for the relational database. The ER diagram will show entities that will be vertex labels, entity attributes that represent property keys, and the relationships that will be edges in DSE Graph.

Migrating to DSE Graph from Apache Cassandra

Migrating an Apache Cassandra™ database to DSE Graph requires analyzing how the data is stored in the Cassandra tables. The following factors should be considered:

- A Cassandra table can represent a vertex and edges. Generally, the relationships will be duplicated amongst Cassandra tables due to denormalization. Identifying the entities and their relationships will require scrutinizing multiple tables.

- Cassandra table columns, especially clustering columns, are good candidates for property keys in DSE Graph.

A good starting point to build the graph data model is an entity-relationship (ER) diagram for the Apache Cassandra database. The ER diagram will show entities that will be vertex labels, entity attributes that represent property keys, and the relationships that will be edges in DSE Graph.
Storage engine

Unlike a typical relational database that uses a balanced tree (B-tree), the DataStax Enterprise (DSE) database uses a storage structure similar to a log-structured merge tree. Essentially, the database avoids reading before writing. Read-before-write, especially in a large distributed system, can result in large latencies in read performance and other problems. For example, two clients read at the same time; one overwrites the row to make update A, and the other overwrites the row to make update B, removing update A. This race condition results in ambiguous query results, making it difficult to determine which update is correct.

To avoid using read-before-write for most writes, the storage engine groups inserts and updates in memory and, at intervals, sequentially writes the data to disk in append mode. Once written to disk, the data is immutable and is never overwritten. Reading data involves combining this immutable, sequentially-written data to discover the correct query results. You can use lightweight transactions (LWT) to check the state of the data before writing. However, this feature is recommended only for limited use.

A log-structured engine that avoids overwrites and uses sequential I/O to update data is essential for writing to hard disks (HDD) and solid-state disks (SSD). On HDD, writing randomly involves a higher number of seek operations, which carries substantial penalties.

For many databases, write amplification is a problem on SSDs. On these drives, memory must be erased before it can be written. Rewriting data requires a portion of the drive to be read, updated, and written to a new location, while also erasing the new location if was used previously before the write occurs. Therefore, much larger portions of the drive must be erased and rewritten than actually required by the new data. This phenomenon of write amplification impacts the life and speed of SSDs.

Because the DSE database sequentially writes immutable files, thereby avoiding write amplification and disk failure, the database accommodates inexpensive, consumer SSDs extremely well.

How the database reads and writes data

To manage and access data, it is important to understand how the DataStax Enterprise (DSE) database stores data. The hinted handoff feature plus conformance and non-conformance to the ACID (atomicity, consistency, isolation, durability) database properties are key concepts to understand reads and writes. In DataStax Enterprise, consistency refers to how up-to-date and synchronized a row of data is on all of its replicas.

Client utilities and application programming interfaces (APIs) for developing applications for data storage and retrieval are available through the DataStax drivers.

How is data written?

The DataStax Enterprise database processes data at several stages on the write path, starting with the immediate logging of a write and ending in with a write of data to disk:

1. Logging data in the commit log

2. Writing data to the memtable

3. Flushing data from the memtable

4. Storing data on disk in SSTables
The time stamp for all writes is UTC (Universal Time Coordinated).

**Logging writes and memtable storage**

When a write occurs, the database stores the data in a memory structure called memtable. To provide configurable durability, the database also appends writes to the commit log on disk. When `commitlog_sync` is set to `sync`, the commit log receives every write made to a node. These durable writes survive permanently even if
power fails on a node. The memtable is a write-back cache of data partitions that the database looks up by key. The memtable stores write operations in sorted order until reaching a configurable limit, and then is flushed.

**Flushing data from the memtable**

To flush data from the memtable, the database writes data to disk in the memtable-sorted order. A partition index is also created on the disk that maps the tokens to a location on disk.

When the memtable content exceeds the configurable threshold, or the commitlog space exceeds the commitlog_total_space_in_mb, the memtable is put in a queue that is flushed to disk. If the data to be flushed exceeds the memtable_cleanup_threshold (which is automatically calculated), the database blocks writes until the next flush succeeds.

You can manually flush a table using nodetool flush or nodetool drain (flushes memtables without listening for connections to other nodes). To reduce the commit log replay time, DataStax recommends flushing the memtable before you restart the nodes. If a node stops working, replaying the commit log restores the writes to the memtable that were there before the node stopped.

**Purging commit log segments**

The database uses the commit log to rebuild memtables during startup to recover after a crash. During normal operation the commit log is divided into segments. Writes are recorded in order and new segments are created when the current segment reaches the commitlog_segment_size_in_mb. The database purges commit log segments only after all the data in a segment has been flushed to disk. If the commit log directory reaches the maximum size (commitlog_total_space_in_mb), the oldest segments are purged and the corresponding tables are flushed to disk.

For example, consider the following two tables:

- Table A has extremely high throughput
- Table B has very low throughput

All the commit log segments contain writes for both table A and table B, as well as system tables. Table A’s memtable fills up rapidly and gets flushed frequently; while table B’s memtable fills up slowly and is rarely flushed. When the commit log reaches the maximum size, it forces Table B’s memtable to flush and then purges the segments.

Table B is flushed into large chunks instead of hundreds of tiny SSTables. If the commit log space and memtable space are equal, Table B’s memtable would flush every time Table A is flushed, despite being much smaller. To summarize, if there is more than one table, it makes sense to have a larger space for commit log segments.

**Storing data on disk in SSTables**

Memtables and SSTables are maintained per table. The commit log is shared among tables. SSTables are immutable, not written to again after the memtable is flushed. Consequently, a partition is typically stored across multiple SSTable files. A number of other SSTable structures exist to assist read operations.

**SSTable names and versions**

SSTables are files stored on disk. The data files are stored in a data directory that varies with installation. For each keyspace, a directory within the data directory stores each table.

For example, the following SSTable version is `aa` and the format is `bti`:

```
data/cycling/cyclist_expenses-e4f31e122bc511e8891b23da85222d3d/aa-1-bti-Data.db```

where

- `cycling` is the keyspace name which distinguishes the keyspace for streaming or bulk loading data.
- `cyclist_expenses` is the table name which is followed by a dash and a hexadecimal string (`-e4f31e122bc511e8891b23da85222d3d`) the unique identifier of the table.
- `aa` is the SSTable version and `bti` is the format. Use this information to determine the DSE version the SSTable is compatible with, see DataStax Enterprise, Apache Cassandra, CQL, and SSTable compatibility.
The database creates a subdirectory for each table, which can be referenced to a chosen physical drive or data volume through a symbolic link (symlink). To improve performance, this capability allows you to move very active tables to faster media, such as SSDs, and also divides tables across all attached storage devices for better I/O balance at the storage layer.

For each SSTable, the database creates the following structures:

**Compression Information (CompressionInfo.db)**
A file holding information about uncompressed data length, chunk offsets, and other compression information

**Data (Data.db)**
The SSTable data

**Digest (Digest.crc32, Digest.adler32, or Digest.sha1)**
A file holding adler32 checksum of the data file

**Bloom filter (Filter.db)**
A structure stored in memory that checks if row data exists in the memtable before accessing SSTables on disk

**Partition index (Partitions.db)**
An index of the partitions keys with pointer to the partition-level in the row index (Row.db) or data file.

**Row index (Row.db)**
Contains the row indices for all partitions in the SSTable.

**Statistics (Statistics.db)**
Statistical metadata about the content of the SSTable

**CRC (CRC.db)**
A file holding the CRC32 for chunks in an uncompressed file.

**SSTable Table of Contents (TOC.txt)**
A file that stores the list of all components for the SSTable TOC

**Secondary Index (SI_*.db)**
Built-in secondary index. Multiple SIs may exist per SSTable

### How is data maintained?

The DataStax Enterprise (DSE) database write process stores data in files called SSTables. SSTables are immutable. Instead of overwriting existing rows with inserts or updates, the database writes new timestamped versions of the inserted or updated data in new SSTables. The database does not perform deletes by removing the deleted data. Instead, the database marks deleted data with tombstones.

Over time, the database may write many versions of a row in different SSTables. Each version may have a unique set of columns stored with different timestamps. As SSTables accumulate, the distribution of data can require accessing more and more SSTables to retrieve a complete row.

To keep the database healthy, the database periodically merges SSTables and discards old data. This process is called compaction.

### Compaction

Compaction works on a collection of SSTables. From these SSTables, compaction collects all versions of each unique row and assembles one complete row, using the most up-to-date version (by timestamp) of each of the row's columns. The merge process is performant, because rows are sorted by partition key within each SSTable, and the merge process does not use random I/O. The new versions of each row is written to a new SSTable. The old versions, along with any rows that are ready for deletion, are left in the old SSTables, and are deleted when any pending reads are completed.
Compaction causes a temporary spike in disk space usage and disk I/O while old and new SSTables co-exist. As it completes, compaction frees up disk space occupied by old SSTables. It improves read performance by incrementally replacing old SSTables with compacted SSTables. The database can read data directly from the new SSTable even before it finishes writing, instead of waiting for the entire compaction process to finish.

DSE provides predictable high performance even under heavy load by replacing old SSTables with new ones in memory as it processes writes and reads. Caching the new SSTable, while directing reads away from the old one, is incremental and does not cause a the dramatic cache miss storm.

### Compaction strategies

The DSE database supports different compaction strategies. These strategies control which SSTables are chosen for compaction and how the compacted rows are sorted into new SSTables. Each strategy has its own strengths. The sections that follow explain each compaction strategy.

Although each of the following sections starts with a generalized recommendation, many factors complicate the choice of a compaction strategy. See Choosing a compaction strategy.

**SizeTieredCompactionStrategy (STCS)**

Recommended for write-intensive workloads.

- **Pros:** Compacts write-intensive workload very well.
- **Cons:** Can hold on to stale data too long. Required memory increases over time.

The SizeTieredCompactionStrategy (STCS) initiates compaction when the database has accumulated a set number (default: 4) of similar-sized SSTables. STCS merges these SSTables into one larger SSTable. As the larger SSTables accumulate, STCS merges these into even larger SSTables. At any given time, several SSTables of varying sizes are present.
While STCS works well to compact a write-intensive workload, it makes reads slower because the merge-by-size process does not group data by rows. This makes it more likely that versions of a particular row may be spread over many SSTables. Also, STCS does not evict deleted data predictably because its trigger for compaction is SSTable size, and SSTables might not grow quickly enough to merge and evict old data. As the largest SSTables grow in size, the amount of disk space needed for both the new and old SSTables simultaneously during STCS compaction can outstrip a typical amount of disk space on a node.

**LeveledCompactionStrategy (LCS)**
Recommended for read-intensive workloads.

- **Pros:** Disk requirements are easier to predict. Read operation latency is more predictable. Stale data is evicted more frequently.
- **Cons:** Much higher I/O utilization impacting operation latency

The LeveledCompactionStrategy (LCS) alleviates some of the read operation issues with STCS. This strategy works with a series of levels. First, data in memtables is flushed to SSTables in the first level (L0). LCS compaction merges these first SSTables with larger SSTables in level L1.

The SSTables in levels greater than L1 are merged into SSTables with a size greater than or equal to \( \text{sstable.size.in.mb} \) (default: 160 MB). If a L1 SSTable stores data of a partition that is larger than L2, LCS moves the SSTable past L2 to the next level up.
In each of the levels above L0, LCS creates SSTables that are about the same size. Each level is 10 times the size of the last level, so level L1 has 10 times as many SSTables as L0, and level L2 has 100 times as many as L0. If the result of the compaction is more than 10 SSTables in level L1, the excess SSTables are moved to level L2.

Keep in mind that the maximum overhead when using LCS is the sum of N-1 levels. For example, given a maximum table size of 160 megabytes, once past level 3, overhead requirements expand drastically from 1.7 terabytes at level 4 to 17 terabytes at level 5:

<table>
<thead>
<tr>
<th>Level</th>
<th>SSTables * Size</th>
<th>Total Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>1 * 160 MB</td>
<td>160 MB</td>
</tr>
<tr>
<td>L1</td>
<td>10 * 1600 MB</td>
<td>16000 MB</td>
</tr>
<tr>
<td>L2</td>
<td>100 * 16000 MB</td>
<td>1600000 MB</td>
</tr>
<tr>
<td>L3</td>
<td>1000 * 1600000 MB</td>
<td>16000000 MB</td>
</tr>
<tr>
<td>L4</td>
<td>10000 * 16000000 MB</td>
<td>1600000000 MB</td>
</tr>
<tr>
<td>L5</td>
<td>100000 * 1600000000 MB</td>
<td>16000000000 MB</td>
</tr>
</tbody>
</table>

To mitigate that situation, switch to STCS and add additional nodes or reduce the sstable size using `sstablesplit`.

The LCS compaction process guarantees that the SSTables within each level starting with L1 have non-overlapping data. For many reads, this process enables the database to retrieve all the required data from only one or two SSTables. In fact, 90% of all reads can be satisfied from one SSTable. Since LCS does not compact L0 tables, however, resource-intensive reads involving many L0 SSTables may still occur.

At levels beyond L0, LCS requires less disk space for compacting: generally, 10 times the fixed size of the SSTable. Obsolete data is evicted more often, so deleted data uses smaller portions of the SSTables on disk. However, LCS compaction operations take place more often and place more I/O burden on the node. For write-intensive workloads, the payoff of using this strategy is generally not worth the performance loss to I/O operations. In many cases, tests of LCS-configured tables reveal I/O saturation on writes and compactions.

The database bypasses compaction operations when bootstrapping a new node using LCS into a cluster. The original data is moved directly to the correct level because there is no existing data, so no partition overlap per level is present. For more information, see the Apache Cassandra 2.2 - Bootstrapping Performance Improvements for Leveled Compaction blog.

**TimeWindowCompactionStrategy (TWCS)**

Recommended for time series and expiring time-to-live (TTL) workloads.
• **Pros**: Well-suited for time series data, stored in tables that use the default TTL for all data. Simpler configuration than that of DateTieredCompactionStrategy (DTCS), which is deprecated in favor of TWCS.

• **Cons**: Not appropriate if out-of-sequence time data is required, because SSTables will not compact well. Also, not appropriate for data without a TTL workload, as storage will grow without bound. Less fine-tuned configuration is possible than with DTCS.

The TimeWindowCompactionStrategy (TWCS) is similar to DTCS with simpler settings. TWCS groups SSTables using a series of time windows. During compaction, TWCS applies STCS to uncompacted SSTables in the most recent time window. At the end of a time window, TWCS compacts all SSTables that fall into that time window into a single SSTable based on the SSTable maximum timestamp. After the major compaction for a time window is completed, no further compaction of the data occurs, although tombstone compaction can still be run on sstables after the `compaction_window` threshold has passed. The process starts over with the SSTables written in the next time window.

For tables where all cells have a **time-to-live (TTL)** applied, or tables using the default TTL, once all TTLs are passed, the `gc_grace_seconds` period has expired, and the droppable tombstone ratio is 100%, the sstable can be dropped without a compaction.
Figure 22: How TimeWindowCompactionStrategy works

10 AM

memtables are all flushed at 100MB in size.

Within the time window, SSTables are STCS compacted.

11 AM

10 AM

When the time window passes, all SSTables are compacted to one table.

11 AM

12 PM

Once the “active” window passes, SSTables are never compacted.
As the figure shows, from 10 AM to 11 AM, the memtables are flushed from memory into 100MB SSTables. These SSTables are compacted into larger SSTables using STCS. At 11 AM, all these SSTables are compacted into a single SSTable, and never compacted again by TWCS.

At 12 PM, the new SSTables created between 11 AM and 12 PM are compacted using STCS, and at the end of the time window, the TWCS compaction repeats. Notice that each TWCS time window contains varying amounts of data.

For an animated explanation, see the DataStax Academy Time Window Compaction Strategy video. A valid DataStax Academy account is required to view the video.

The TWCS configuration has two main property settings:

- `compaction_window_unit`: time unit used to define the window size (milliseconds, seconds, hours, and so on)
- `compaction_window_size`: how many units per window (1, 2, 3, and so on)

The configuration for the above example: `compaction_window_unit = 'minutes', compaction_window_size = 60`

**DateTieredCompactionStrategy (DTCS) - Deprecated**

Use TimeWindowCompactionStrategy (TWCS) instead.

The DateTieredCompactionStrategy (DTCS) is similar to STCS. But instead of compacting based on SSTable size, DTCS compacts based on SSTable age. Each column in an SSTable is marked with the timestamp at write time. As the age of an SSTable, DTCS uses the oldest (minimum) timestamp of any column the SSTable contains.

**More information about compaction**

The following blog posts and videos provide more information from developers that have tested compaction strategies:

- Choosing a compaction strategy
- When to Use Leveled Compaction
- Leveled compaction in Apache Cassandra
- Using TimeWindowCompactionStrategy for Time Series Workloads
- What delays a tombstone purge when using LCS in Cassandra

**How is data updated?**

The DataStax Enterprise (DSE) database treats each new row as an upsert: if the new row has the same primary key as that of an existing row, the database processes it as an update to the existing row.

During a write, DataStax Enterprise adds each new row to the database without checking on whether a duplicate record exists. This policy makes it possible that many versions of the same row may exist in the database.

Periodically, the rows stored in memory are streamed to disk into structures called SSTables. At certain intervals, the database compacts smaller SSTables into larger SSTables. If the database encounters two or more versions of the same row during this process, it only writes the most recent version to the new SSTable. After compaction, the database drops the original SSTables, deleting the outdated rows.

Most DSE installations store replicas of each row on two or more nodes. Each node performs compaction independently. This means that out-of-date versions of a row have been dropped from one node, they may still exist on another node.

This is why the database performs another round of comparisons during a read process. When a client requests data with a particular primary key, DataStax Enterprise retrieves many versions of the row from one or more replicas. The version with the most recent timestamp is the only one returned to the client ("last-write-wins").
Some database operations may only write partial updates of a row, so some versions of a row may include some columns, but not all. During a compaction or write, the database assembles a complete version of each row from the partial updates, using the most recent version of each column.

**How is data deleted?**

The processes for deleting data are designed to improve performance and work with the DataStax Enterprise (DSE) database built-in properties for data distribution and fault-tolerance.

The database treats a delete as an insert or *upsert*. The data being added to the partition in the `DELETE` command is a deletion marker called a *tombstone*. The tombstones go through the write path, and are written to SSTables on one or more nodes. A key differentiator of a tombstone is a built-in expiration known as the grace period, set by `gc_grace_seconds`. At the end of its expiration period, the tombstone will be deleted as part of the normal compaction process, if it no longer marks any deleted data in another SSTable.

For more information about tombstones, see [What are tombstones?](#)

**Deletion in a distributed system**

In a multi-node cluster, the DSE database can store replicas of the same data on two or more nodes. This helps prevent data loss, but complicates the delete process. If a node receives a delete for data it stores locally, the
node marks the specified record for deletion and tries to pass the tombstone to other nodes containing replicas of that record. If one replica node is unresponsive at that time, it does not receive the tombstone immediately, so it still contains the pre-delete version of the record.

If the tombstone has already been deleted from the rest of the cluster before that node recovers, the database treats the record on the recovered node as new data, and propagates it to the rest of the cluster. This kind of deleted but persistent record is called a zombie.

Figure 24:

To prevent the reappearance of zombies, the database gives each tombstone a grace period. The purpose of the grace period is to give unresponsive nodes time to recover and process tombstones normally. When multiple replica answers are part of a read request, and those responses differ, then whichever values are most recent take precedence. For example, if a node has a tombstone but another node has a more recent change, then the final result includes the more recent change.

If a node has a tombstone and another node has only an older value for the record, then the final record will have the tombstone. If a client writes a new update to the tombstone during the grace period, the database overwrites the tombstone.

When an unresponsive node recovers, DataStax Enterprise uses hinted handoffs to replay the database mutations that the node missed while it was down. DSE does not replay a mutation for a tombstone during its grace period. If the node does not recover until after the grace period ends, the deletion might be missed.

After the tombstone's grace period ends, DSE deletes the tombstone during compaction.
Expiring data

The grace period for a tombstone is set by the `gc_grace_seconds` property. The default value is 864,000 seconds (ten days), and each table can have its own value for this property. On a single-node cluster, this property can safely be set to zero.

- The expiration date/time for a tombstone is the date/time of its creation plus the value of the `gc_grace_seconds` property.
- To completely prevent the reappearance of zombie records, run `nodetool repair` on a node after it recovers, and on each table every interval set by `gc_grace_seconds`.

If all records in a table are given a TTL at creation, are allowed to expire, and are not deleted manually, it is not necessary to run `nodetool repair` for that table on a regular basis. For more information expiring data with TTL, see Expiring data with TTL.

If using `SizeTieredCompactionStrategy` (STCS), delete any expired tombstones immediately by manually starting the compaction process.

If forcing compaction, the database might create one very large SSTable from all the data and will not trigger another compaction for a long time. The data in the SSTable created during the forced compaction can grow very stale during this long period of non-compaction.

DSE also supports batch data insertion and updates. This procedure introduces the danger of replaying a record insertion after that record has been removed from the rest of the cluster. DSE does not replay a batched mutation for a tombstone that is still within its grace period.

DSE supports immediate deletion through the `DROP KEYSPACE` and `DROP TABLE` statements.

Diagram legend

The following table explains the icons in the delete diagrams:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="data_icon.png" alt="Data icon" /></td>
<td>Data on a node.</td>
</tr>
<tr>
<td><img src="data_unavailable_icon.png" alt="Data icon" /></td>
<td>Data on an unavailable replica node.</td>
</tr>
<tr>
<td><img src="data_removed_icon.png" alt="Data icon" /></td>
<td>Data removed from node.</td>
</tr>
<tr>
<td><img src="tombstone_icon.png" alt="Tombstone icon" /></td>
<td>Tombstone indicating that data has been deleted.</td>
</tr>
<tr>
<td><img src="tombstone_removed_icon.png" alt="Tombstone icon" /></td>
<td>Tombstone removed from node.</td>
</tr>
</tbody>
</table>

What are tombstones?

In DataStax Enterprise (DSE), a tombstone is created when data is deleted from a keyspace that has a replication factor (RF) greater than 1. The following list of examples is not exhaustive, but illustrates some operations that generate tombstones:

- Using a CQL DELETE statement.
- Expiring data with time-to-live (TTL).
- Using internal operations, such as Using materialized views.
- INSERT or UPDATE operations with a null value.
• **UPDATE** operations on a collection column.

When a tombstone is created, it can be marked on different parts of a partition. Based on the location of the marker, tombstones can be categorized into one of the following groups. Each category typically corresponds to one unique type of data deletion operation.

- Partition tombstones
- Row tombstones
- Range tombstones
- ComplexColumn tombstones
- Cell tombstones
- TTL tombstones

The tombstones go through the write path, and are written to SSTables on one or more nodes. A key differentiator of a tombstone is a built-in expiration known as the grace period, set by `gc_grace_seconds`. At the end of its expiration period, the tombstone will be deleted as part of the normal compactation process, if it no longer marks any deleted data in another SSTable.

Having an excessive number of tombstones in a table can negatively impact application performance. Many tombstones often indicate potential issues with either the data model or in the application.

**Create the keyspace and tables**

In the following examples, the `cycling` keyspace is used to illustrate different tombstone categories. Two tables are used: `rank_by_year_and_cycling_name` and `cyclist_career_teams`.

Because the following examples use both `cqlsh` and CQL commands, using two different terminals is recommended.

Alternatively, use one terminal for `cqlsh` and issue CQL commands using DataStax Studio.

Before getting started, copy the following commands into a `cqlsh` prompt to create the `cycling` keyspace, create both tables, and insert data into the `rank_by_year_and_cycling_name` table.

You insert data later into the `cyclist_career_teams` table in Cell tombstones and TTL tombstones.

```sql
CREATE KEYSPACE cycling WITH replication = 
{'class': 'SimpleStrategy', 'replication_factor': '1'} AND durable_writes = true;

CREATE TABLE cycling.rank_by_year_and_name (  
  race_year int,
  race_name text,
  rank int,
  cyclist_name text,
) PRIMARY KEY ((race_year, race_name), rank)
) WITH CLUSTERING ORDER BY (rank ASC);

INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank) VALUES (2015, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Benjamin PRADES', 1);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank) VALUES (2015, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Adam PHELAN', 2);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank) VALUES (2015, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Thomas LEBAS', 3);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank) VALUES (2014, '4th Tour of Beijing', 'Phillippe GILBERT', 1);
```

DSE 6.0 Architecture Guide Previous DSE version Latest 6.0 patch: 6.0.11
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank) VALUES (2014, '4th Tour of Beijing', 'Johan Esteban CHAVES', 3);

CREATE TABLE cycling.cyclist_career_teams (
    id UUID PRIMARY KEY,
    lastname text,
    teams set<text>
);

Flushing to SSTables

After each modification to a table, run the `nodetool flush` command on the cycling keyspace to flush data from the memtable to the SSTables on disk. This step is necessary before running `sstabledump` to view the output.

```bash
$ nodetool flush cycling;
```

After flushing the cycling keyspace, run the `sstabledump` command on the SSTable, as shown in the following example.

```bash
$ cd /var/lib/cassandra/data/cycling/rank_by_year_and_name-bc05fba12ba81e8b4a8ad2b042f3e18 sstabledump mc-2-big-Data.db
```

The `sstabledump` utility is available in Apache Cassandra™ 3.0, DSE 5.0, and later. For versions before these, use the `table2json` utility instead.

Partition tombstones

Partition tombstones are generated when an entire partition is deleted explicitly. In the CQL DELETE statement, the WHERE clause is an equality condition against the partition key.

```cql
DELETE from cycling.rank_by_year_and_name WHERE race_year = 2014 AND race_name = '4th Tour of Beijing';
```

Looking at the sstabledump output for this partition, the `deletion_info` tombstone marker is at the partition level, and is not associated with any rows or cells within the partition.

```json
{
    "partition" : {
        "key" : [ "2014", "4th Tour of Beijing" ],
        "position" : 0,
    },
    "rows" : [ ]
}
```

Row tombstones

Row tombstones are generated when a particular row within a partition is deleted explicitly. The schema has a composite primary key that includes both the partition key and the clustering key. In the CQL DELETE statement, the WHERE clause is an equality condition against both the partition key and the clustering key columns.

```cql
DELETE from cycling.rank_by_year_and_name WHERE
```
Looking at the sstabledump output for this partition, the `deletion_info` tombstone marker is at the row level, and is identified by a clustering key under the partition. Neither the partition nor the row cells contain the tombstone marker.

```
{
  "partition": {
    "key": [ "2015", "Giro d'Italia - Stage 11 - Forli > Imola" ],
    "position": 0
  },
  "rows": [
    {
      "type": "row",
      "position": 74,
      "clustering": [ 2 ],
      "deletion_info": {
        "marked_deleted": "2018-05-18T15:29:06.227148Z",
        "local_delete_time": "2018-05-18T15:29:06Z"
      },
      "cells": [ ]
    }
  ]
}
```

### Range tombstones

Range tombstones occur when several rows within a partition that can be expressed through a range search are deleted explicitly. The schema has a composite primary key that includes both a partition key and a clustering key. In the CQL DELETE statement, the WHERE clause is an equality condition against the partition key, plus an inequality condition against the clustering key.

If following this tutorial from the beginning, drop the `rank_by_year_and_name` table and then recreate it to populate the table with the necessary data.

```
DELETE from cycling.rank_by_year_and_name WHERE race_year = 2015 AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu' AND rank > 1;
```

Looking at the sstabledump output for this partition, the `deletion_info` tombstone marker is at the row level. A special boundary marker, `range_tombstone_bound`, marks the range scope (identified by the clustering key values) of the deleted rows.

```
{
  "partition": {
    "key": [ "2015", "Tour of Japan - Stage 4 - Minami > Shinshu" ],
    "position": 252
  },
  "rows": [
    {
      "type": "range_tombstone_bound",
      "start": {
        "type": "inclusive",
        "deletion_info": {
          "marked_deleted": "2018-05-18T16:09:21.474713Z",
          "local_delete_time": "2018-05-18T16:09:21Z"
        }
      }
    },
    {
      "type": "range_tombstone_bound",
      "end": {
        "type": "exclusive",
        "clustering": [ 1 ]
      }
    }
  ]
}
```
ComplexColumn tombstones

ComplexColumn tombstones are generated when inserting or updating a collection type column, such as set, list, and map.

Previously we created the `cyclist_career_teams` table. Run the following `cqlsh` command to insert data into that table.

```cql
INSERT INTO cycling.cyclist_career_teams (
    id,
    lastname,
    teams)
VALUES (cb07baad-eac8-4f65-b28a-bddc06a0de23, 'ARMITSTEAD', {
    'Boels-Dolmans Cycling Team', 'AA Drink - Leontien.nl', 'Team Garmin - Cervelo' });
```

Looking at the `sstabledump` output for this partition, no explicit manual deletion occurs on the partition, but a `deletion_info` marker is listed at the cell level for the collection type column `teams`.

```json
{
    "partition": {
        "key": [ "cb07baad-eac8-4f65-b28a-bddc06a0de23" ],
        "position": 0,
    },
    "rows": [
        {
            "type": "row",
            "position": 130,
            "liveness_info": { "tstamp": "2018-05-18T16:26:23.779724Z" },
            "cells": [
                {
                    "name": "lastname",
                    "value": "ARMITSTEAD"
                },
                {
                    "name": "teams",
                    "deletion_info": {
                        "marked_deleted": "2018-05-18T16:26:23.779723Z",
                        "local_delete_time": "2018-05-18T16:26:23Z"
                    }
                },
                {
                    "name": "teams",
                    "path": [ "AA Drink - Leontien.nl" ],
                    "value": ""
                },
                {
                    "name": "teams",
                    "path": [ "Boels-Dolmans Cycling Team" ],
                    "value": ""
                },
                {
                    "name": "teams",
                    "path": [ "Team Garmin - Cervelo" ],
                    "value": ""
                }
            ]
        }
    ]
}
```

Cell tombstones

Cell tombstones are generated when explicitly deleting a value from a cell, such as a column for a specific row of a partition, or when inserting or updating a cell with null values, as shown in the following example.

```cql
INSERT INTO cycling.rank_by_year_and_name (
    race_year,
    race_name,
    cyclist_name,
    rank)
```
VALUES (2018, 'Giro d''Italia - Stage 11 - Osimo > Imola', null, 1);

Looking at the "sstabledump" output for this partition, deletion_info tombstone marker is associated with a particular cell.

```
{
  "partition": {
    "key": [ "2018", "Giro d'Italia - Stage 11 - Osimo > Imola" ],
    "position": 0
  },
  "rows": [
    {
      "type": "row",
      "position": 80,
      "clustering": [ 1 ],
      "liveness_info": { "tstamp": "2018-05-18T17:13:42.602827Z" },
      "cells": [
        { "name": "cyclist_name", "deletion_info": { "local_delete_time": "2018-05-18T17:13:42Z" } }
      ]
    }
  ]
}
```

**TTL tombstones**

TTL tombstones are generated when the TTL (time-to-live) period expires. The TTL expiration marker can occur at either the row or cell level. However, DSE marks TTL data differently from tombstone data that was explicitly deleted. Even if a partition has only a single row (with no clustering key), the TTL mark is still made at the row level.

The following statement sets TTL for an entire row.

```
INSERT INTO cycling.cyclist_career_teams (id, lastname, teams)
VALUES (e7cd5752-bc0d-4157-a80f-7523add8dbcd, 'VAN DER BREGGEN', {
  'Rabobank-Liv Woman Cycling Team','Sengers Ladies Cycling Team','Team Flexpoint' })
USING TTL 1;
```

The following statement sets TTL for a single cell.

```
UPDATE cycling.rank_by_year_and_name USING TTL 1
SET cyclist_name = 'Cloudy Archipelago' WHERE race_year = 2018 AND
race_name = 'Giro d''Italia - Stage 11 - Osimo > Imola' AND rank = 1;
```

Looking at the sstabledump output for these partitions, the first CQL statement marks the row (partition key: e7cd5752-bc0d-4157-a80f-7523add8dbcd) with an "expired": true TTL expiration marker in the liveness_info section.

```
{
  "partition": {
    "key": [ "e7cd5752-bc0d-4157-a80f-7523add8dbcd" ],
    "position": 0
  },
  "rows": [
    {
      "type": "row",
      "position": 134,
```
The second CQL statement marks the cell (partition key: 2018, clustering key: 1, column name: cyclist_name) with an "expired": true TTL expiration marker for the specific cell.

```
{
  "partition": {
    "key": [ "2018", "Giro d'Italia - Stage 11 - Osimo > Imola" ],
    "position": 0
  },
  "rows": [
    {
      "type": "row",
      "position": 95,
      "clustering": [ 1 ],
      "cells": [
      ]
    }
  ]
}
```

**How are indexes stored and updated?**

Secondary indexes filter tables for data stored in non-primary key columns. For example, a table storing cyclist names and ages using the last name of the cyclist as the primary key might have a secondary index on the age to allow queries by age. Because querying should always result in a continuous slice of data being retrieved from the table, querying to match a non-primary key column is an anti-pattern.
If the table rows are stored based on last names, the table might be spread across several partitions stored on different nodes. Queries based on a particular range of last names, such as all cyclists with the last name Matthews, retrieve sequential rows from the table. However, a query based on age, such as all cyclists who are 28, requires all nodes to be queried for a value. An index on age could be used, but a better solution is to create a materialized view or additional table that is ordered by age.

Non-primary keys play no role in ordering the data in storage, so querying for a particular value of a non-primary key column results in scanning all partitions. Scanning all partitions generally results in a prohibitive read latency, and is not allowed.

Secondary indexes can be built for a column in a table. These indexes are stored locally on each node in a hidden table and built in a background process. If a query includes both a partition key condition and a secondary index column condition, the query will be successful because the query can be directed to a single node partition.

If a secondary index is used in a query that is not restricted to a particular partition key, the query will have prohibitive read latency because all nodes will be queried. A query with these parameters is allowed only if the query option allow filtering is used. This option is not appropriate for production environments, and does not guarantee trouble-free indexing. Knowing when to use an index is imperative.

As with relational databases, keeping indexes current uses processing time and resources, so unnecessary indexes should be avoided. When a column is updated, the index is updated as well. If the old column value still exists in the memtable, which typically occurs when updating a small set of rows repeatedly, DataStax Enterprise (DSE) removes the corresponding obsolete index entry; otherwise, the old entry remains to be purged by compaction. If a read sees a stale index entry before compaction purges it, the reader thread invalidates it.
How is data read?

To satisfy a read, the DataStax Enterprise (DSE) database must combine results from the active memtable and potentially multiple SSTables. If the memtable has the desired partition data, then the data is read and merged with the data from the SSTables.

The database processes data at several stages on the read path to discover where the data is stored, starting with the data in the memtable and finishing with SSTables:

1. Check the memtable
2. Check row cache, if enabled
3. Check Bloom filter
4. Find the partition offset in the partition index in the chunk cache/memory
5. If any required index data is not present in the cache, pull it from disk
6. Read the data from the uncompressed chunk cache
7. If any required data chunk is not present in the cache:
   a. Locate the data on disk using the compression offset map
   b. Fetch the data from the SSTable on disk into the chunk cache
Figure 26: Read request flow
Row cache

Typical of any database, reads are fastest when the most in-demand data fits into memory. The operating system page cache is best at improving performance, although the row cache can provide some improvement for very read-intensive operations, where read operations are 95% of the load. Rowcache is not recommended for write-intensive operations. If the row cache is enabled, it stores a subset of the partition data stored on disk in the SSTables in memory. In DataStax Enterprise 5.0 and later, the row cache is stored in fully off-heap memory using an implementation that relieves garbage collection pressure in the Java Virtual Machine (JVM). The subset stored in the row cache uses a configurable amount of memory for a specified period of time. When the cache is full, the row cache uses LRU (least-recently-used) eviction to reclaim memory.

The row cache size is configurable, as is the number of rows to store. Configuring the number of rows to be stored is a useful feature, making a “Last 10 Items” query very fast to read. If row cache is enabled, desired partition data is read from the row cache, potentially saving two seeks to disk for the data. The rows stored in row cache are frequently accessed rows that are merged and saved to the rowcache from the SSTables as they are accessed. After storage, the data is available to later queries. The row cache is not write-through. If a write comes in for the row, the cache for that row is invalidated and is not cached again until the row is read. Similarly, if a partition is updated, the entire partition is removed from the cache. When the desired partition data is not found in the row cache, the bloom filter is checked.

Bloom filter

Each SSTable has an associated Bloom filter, which can establish that an SSTable does not contain certain partition data. A Bloom filter can also determine the likelihood that partition data is stored in an SSTable by narrowing the pool of keys, which increases partition key lookup.

The DSE database checks the Bloom filter to discover which SSTables are likely to have the requested partition data. If the Bloom filter does not rule out an SSTable, the DSE database checks the partition index. Not all SSTables identified by the Bloom filter will have data. Because the Bloom filter is a probabilistic function, it can sometimes return false positives.

The Bloom filter is stored in off-heap memory, and grows to approximately 1-2 gigabytes (GB) per billion partitions. In an extreme case, each row can have a partition, so a single machine can easily have billions of these entries. To trade memory for performance, tune the Bloom filter.

Partition index

The partition index maps partition keys to a row index and supports iteration from a partially specified partition position if required. The size of database partitions often varies. The partition index trie data structure uses a unique byte-ordered partition key prefixes to point to:

• A row index for tables that have wide partitions
• Directly to the data position in a file for tables that have partitions that only include a few rows or a single row

If a leaf of the trie is reached, then the prefix of the partition key matches some content in the file, but we are not yet sure if it is a full match for the partition key. The leaf node points to a place in the row index or data file. In either case, the first bytes at the specified position contain a serialization of the partition key, which is compared to the mapped key. If it matches, we have found the partition. If not, since the stored prefixes are unique, no data for this partition exists in the SSTable.

Uncompressed data in chunk cache (memory)

The chunk cache buffers data before it is compressed and written to an SSTable. When reading data, uncompressed data is checked first. If the data is compressed and on disk, the process uses the compression offset map to locate the data on disk and uncompress the SSTable into memory. See Compaction.

Compression offset map

The compression offset map stores pointers to the exact location on disk where the desired partition data will be found. This location is stored in off-heap memory and is accessed by either the partition key cache or the partition index. After the compression offset map identifies the disk location, the desired compressed partition data is fetched from the correct SSTable(s). The query then receives the result set.
Within a partition, all rows are not equally expensive to query. The very beginning of the partition (the first rows, clustered by your key definition) is slightly less expensive to query because there is no need to consult the partition-level index.

The compression offset map grows to 1-3 gigabytes (GB) per terabyte (TB) compressed. The more data is compressed, the greater number of compressed blocks required, and the larger the compression offset table. Compression is enabled by default even though going through the compression offset map consumes CPU resources. Having compression enabled makes the page cache more effective, and typically results in a timely search.

How do write patterns affect reads?

It is important to consider how write operations affect read operations in the cluster. The compaction process is configurable and can significantly affect read performance. Using the SizeTieredCompactionStrategy (STCS) tends to cause data fragmentation when rows are frequently updated. The LeveledCompactionStrategy (LCS) was designed to prevent fragmentation under this condition, and is recommended for read-intensive workloads.

Data consistency

How are consistent read and write operations handled?

Consistency refers to how up-to-date and synchronized all replicas of a row of data are at any given moment. Ongoing repair operations in DataStax Enterprise (DSE) ensure that all replicas of a row will eventually be consistent. Repairs work to decrease the variability in replica data, but constant data traffic through a widely distributed system can lead to inconsistency (stale data) at any time. DSE is an AP (highly available and partition tolerant) system according to the CAP theorem. DSE has flexibility in its configuration, and can perform more like a CP (consistent and partition tolerant) system according to the CAP theorem, depending on the application requirements. Two important consistency features to understand are tunable consistency and linearizable consistency.

Tunable consistency

To ensure the database can provide the proper levels of consistency for its reads and writes, DSE extends the concept of eventual consistency by offering tunable consistency. The consistency level can be tuned for each operation, or set globally for a cluster or datacenter. You can vary the consistency for individual read or write operations so that the data returned is more or less consistent, as required by the client application. This allows DSE to act more like a CP or AP system, depending on the application requirements.

There is a tradeoff between operation latency and consistency: higher consistency incurs higher latency, and lower consistency permits lower latency. You can control latency by tuning consistency.

It is not possible to tune a distributed database into a completely CA system. See You Can't Sacrifice Partition Tolerance for a more detailed discussion.

The consistency level determines the number of replicas that must acknowledge the read or write operation success to the client application. For read operations, the read consistency level specifies how many replicas must respond to a read request before returning data to the client application. If a read operation reveals inconsistency among replicas, the database initiates a read repair to update the inconsistent data.

For write operations, the write consistency level specifies how many replicas must respond to a write request before the write is considered successful. Even at low consistency levels, the database writes to all replicas of the partition key, including replicas in other datacenters. The write consistency level only specifies when the coordinator node can report to the client application that the write operation is considered complete. Write operations use hinted handoffs to ensure the writes are completed when replicas are down or otherwise not responsive to the write request.

Typically, a client specifies a consistency level that is less than the replication factor specified by the keyspace. Another common practice is to write at a consistency level of QUORUM and read at a consistency level of QUORUM. The choices made depend on the client application's needs. DSE provides maximum flexibility for application design.
Linearizable consistency

In ACID terms, linearizable consistency (or serial consistency) is a serial (immediate) isolation level for lightweight transactions. DSE does not use employ traditional mechanisms like locking or transactional dependencies when concurrently updating multiple rows or tables.

However, some operations must be performed in sequence and not interrupted by other operations. For example, duplications or overwrites in user-account creation can have serious consequences. Situations like race conditions (two clients updating the same record) can introduce inconsistency across replicas. Writing with high consistency does nothing to reduce this. You can apply linearizable consistency to a unique identifier, like a user ID or email address, although it is not required for all aspects of the user’s account. Serial operations for these elements can be implemented in the database with the Paxos consensus protocol, which uses a quorum-based algorithm to ensure that at least some surviving processor retains knowledge of search results in the event of failure.

Lightweight transactions can be implemented without the need for a master database or two-phase commit process. Lightweight transaction write operations use the serial consistency level for Paxos consensus and the regular consistency level for writing to the table. For more information, see Lightweight Transactions.

Calculating consistency

Reliability of read and write operations depends on the consistency used to verify the operation. Strong consistency can be guaranteed when the following condition is true:

\[ R + W > N \]

where

- \( R \) is the consistency level of read operations
- \( W \) is the consistency level of write operations
- \( N \) is the number of replicas

For example, if the replication factor is 3, then the consistency level of the reads and writes combined must be at least 4. Read operations using 2 out of 3 replicas to verify the value, and write operations using 2 out of 3 replicas to verify the write value will result in strong consistency. If fast write operations are required, but strong consistency is still desired, the write consistency level is lowered to 1, but now read operations have to verify a matched value on all 3 replicas. Writes will be fast, but reads will be slower.

Eventual consistency occurs if the following condition is true:

\[ R + W \leq N \]

If the replication factor is 3, then the consistency level of the reads and writes combined are 3 or less. For example, read operations using QUORUM (2 out of 3 replicas) to verify the value, and write operations using ONE (1 out of 3 replicas) to do fast writes will result in eventual consistency. All replicas will receive the data, but read operations are more vulnerable to selecting data before all replicas write the data.

Additional consistency examples:

- Write at ONE and the replica crashes one second later. The other messages are not delivered. The data is lost.
- Write at ONE and the operation times out. Future reads can return the old or the new value. You will not know the data is incorrect.
- Write at ONE and one of the other replicas is down. The node comes back online. The application will get old data from that node until the node gets the correct data or a read repair occurs.
- Write at QUORUM and then a read at QUORUM. One of the replicas dies. You will always get the correct data.
How do DataStax Enterprise transactions differ from RDBMS transactions?

DSE does not adhere to ACID (Atomicity, Consistency, Isolation, Durability) transactions with rollback or locking mechanisms, but offers atomic, isolated, and durable transactions with eventual and tunable consistency that allows the user decide how strong or eventual they want each transaction’s consistency to be.

As a non-relational database, DSE does not support joins or foreign keys, and consequently does not offer consistency in the ACID sense. For example, when moving money from account A to B, the total in the accounts does not change. DSE supports atomicity and isolation at the row-level, but trades transactional isolation and atomicity for high availability and fast write performance.

Atomicity

In the DSE database, a write operation is atomic at the partition level, meaning the insertions or updates of two or more rows in the same partition are treated as one write operation. A delete operation is also atomic at the partition level.

For example, if using a write consistency level of QUORUM with a replication factor of three, the database replicates the write to all nodes in the cluster and waits for acknowledgement from two nodes. If the write fails on one node but succeeds on another node, DSE reports a failure to replicate the write on that node, but the replicated write that succeeds on the other node is not automatically rolled back.

DSE uses client-side timestamps to determine the most recent update to a column. The latest timestamp always wins when requesting data, so if multiple client sessions update the same columns in a row concurrently, the most recent update is the one seen by readers.

   The timestamp for all writes is UTC (Universal Time Coordinated).

Isolation

DSE write and delete operations are performed with full row-level isolation. This means that a write to a row within a single partition on a single node is only visible to the client performing the operation. The operation is restricted to this scope until it is complete. All updates in a batch operation belonging to a given partition key have the same restriction. However, a batch operation is not isolated if it includes changes to more than one partition.

Durability

Writes in the DSE database are durable. All writes to a replica node are recorded both in memory and in a commit log on disk before they are acknowledged as a success. If a crash or server failure occurs before the memtables are flushed to disk, the commit log is replayed on restart to recover any lost writes. In addition to the local durability (data immediately written to disk), the replication of data on other nodes strengthens durability.

You can manage the local durability to suit your needs for consistency using commitlog_sync in the cassandra.yaml file. Set the option to either periodic or batch.

How do I accomplish lightweight transactions with linearizable consistency?

Distributed databases present a unique challenge when data must be strictly read and written sequentially. In transactions for creating user accounts or transferring money, race conditions between two potential writes must be regulated to ensure that one write precedes the other. The DataStax Enterprise (DSE) database uses the Paxos consensus protocol to implement lightweight transactions that can handle concurrent operations.

The Paxos protocol is implemented in the database with linearizable consistency, which ensures transaction isolation at a level similar to the serializable level offered by relational database management systems (RDBMSs). This type of transaction is known as compare and set (CAS). Replica data is compared and any data out of date is set to the most consistent value. In DSE, the process combines the Paxos protocol with normal read and write operations to accomplish the CAS operation.

The Paxos protocol is implemented as a series of phases:

1. Prepare/Promise
2. Read/Results
3. Propose/Accept
4. Commit/Acknowledge
These phases are actions that take place between a proposer and acceptors. Any node can be a proposer, and multiple proposers can be operating at the same time. For simplicity, this description will use only one proposer. A proposer prepares by sending a message to a quorum of acceptors that includes a proposal number. Each acceptor promises to accept the proposal if the proposal number is the highest they have received. After the proposer receives a promise from a quorum of acceptors, the value for the proposal is read from each acceptor and sent back to the proposer. The proposer determines which value to use and proposes the value to a quorum of the acceptors along with the proposal number. Each acceptor accepts the proposal with a certain number if the acceptor is not already promised to a proposal with a high number. The value is committed and acknowledged as a write operation if all conditions are met.

These four phases require four round trips between a node proposing a lightweight transaction and any cluster replicas involved in the transaction. Therefore, performance will be affected. Reserve lightweight transactions for situations where concurrency must be considered.

Lightweight transactions will block other lightweight transactions from occurring, but will not stop normal read and write operations from occurring. Lightweight transactions use a timestamping mechanism different from normal operations, so mixing lightweight transactions and normal operations can result in errors. If lightweight transactions are used to write to a row within a partition, only lightweight transactions for both read and write operations should be used. This caution applies to all operations, whether individual or batched.

For example, the following series of operations can fail:

```plaintext
DELETE ...
INSERT .... IF NOT EXISTS
SELECT ....
```

The following series of operations will work:

```plaintext
DELETE ... IF EXISTS
INSERT .... IF NOT EXISTS
SELECT ..... 
```

### Reads with linearizable consistency

A **SERIAL** consistency level allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, the database performs a **read repair** as part of the commit.

#### How do I discover consistency level performance?

Before changing the consistency level on read and write operations, discover how your CQL commands are performing using the **TRACING** command in CQL. Using **cqlsh**, you can vary the consistency level and trace read and write operations. The tracing output includes latency times for the operations.

The CQL documentation includes an **example** comparing consistency levels.

For more information on tracing data, see [this post](https://blog.datastax.com) on the DataStax Support Blog, which explains in detail how to locate data on disk.

#### How is the consistency level configured?

Consistency levels in DataStax Enterprise (DSE) can be configured to manage availability versus data accuracy. Configure consistency for a session or per individual read or write operation.

Within **cqlsh**, use **CONSISTENCY** to set the consistency level for all queries in the current **cqlsh** session. For programming client applications, set the consistency level using an appropriate driver. For example, using the Java driver, call `QueryBuilder.insertInto` with `setConsistencyLevel` to set a per-insert consistency level.

The consistency level defaults to **ONE** for all write and read operations.

### Write consistency levels

This table describes the write consistency levels.
### Database internals

#### Table 3: Write Consistency Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition.</td>
<td>Provides the highest consistency and the lowest availability of any other level.</td>
</tr>
<tr>
<td>EACH_QUORUM</td>
<td>A write must be written to the commit log and memtable on a quorum of replica nodes in each datacenter.</td>
<td>Use in multiple datacenter clusters to strictly maintain consistency at the same level in each datacenter. For example, choose this level if you want a write to fail when a datacenter is down and the QUORUM cannot be reached on that datacenter. Strong consistency.</td>
</tr>
<tr>
<td>QUORUM</td>
<td>A write must be written to the commit log and memtable on a quorum of replica nodes across all datacenters.</td>
<td>Use in single or multiple datacenter clusters to maintain strong consistency across the cluster. Use if you can tolerate some level of failure.</td>
</tr>
<tr>
<td>LOCAL_QUORUM</td>
<td>A write must be written to the commit log and memtable on a quorum of replica nodes in the same datacenter as the coordinator node. Avoids latency of inter-datacenter communication.</td>
<td>Use to maintain consistency within the single datacenter in multiple-datacenter clusters with a rack-aware replica placement strategy, such as NetworkTopologyStrategy, and a properly configured snitch. Strong consistency.</td>
</tr>
<tr>
<td>ONE</td>
<td>A write must be written to the commit log and memtable of at least one replica node.</td>
<td>Satisfies the needs of most users because consistency requirements are not stringent.</td>
</tr>
<tr>
<td>TWO</td>
<td>A write must be written to the commit log and memtable of at least two replica nodes.</td>
<td>Similar to ONE.</td>
</tr>
<tr>
<td>THREE</td>
<td>A write must be written to the commit log and memtable of at least three replica nodes.</td>
<td>Similar to TWO.</td>
</tr>
<tr>
<td>LOCAL_ONE</td>
<td>A write must be sent to and successfully acknowledged by at least one replica node in the local datacenter.</td>
<td>Achieves a consistency level of ONE without cross-datacenter traffic, which is desirable for multiple datacenter clusters. For security and quality reasons, use this consistency level in an offline datacenter. If an offline node goes down, LOCAL_ONE prevent automatic connection to online nodes in other datacenters.</td>
</tr>
<tr>
<td>ANY</td>
<td>A write must be written to at least one node. If all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, an ANY write is not readable until the replica nodes for that partition have recovered.</td>
<td>Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability.</td>
</tr>
</tbody>
</table>

#### Read consistency levels

This table describes read consistency levels.

| Table 4: Read Consistency Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>Returns the record after all replicas have responded.</td>
<td>Provides the highest consistency of all levels and the lowest availability of all levels.</td>
</tr>
<tr>
<td>EACH_QUORUM</td>
<td>Returns the record after a quorum of replica nodes in each datacenter has responded.</td>
<td>Used in multiple datacenters to provide strict read consistency for data returned from each datacenter.</td>
</tr>
<tr>
<td>QUORUM</td>
<td>Returns the record after a quorum of replicas from all datacenters has responded.</td>
<td>Used in single or multiple datacenter clusters to maintain strong consistency across the cluster. Ensures strong consistency if you can tolerate some level of failure.</td>
</tr>
<tr>
<td>LOCAL_QUORUM</td>
<td>Returns the record after a quorum of replicas in the current datacenter as the coordinator node has reported. Avoids latency of inter-datacenter communication.</td>
<td>Use to maintain consistency within the single datacenter in multiple-datacenter clusters with a rack-aware replica placement strategy, such as NetworkTopologyStrategy, and a properly configured snitch.</td>
</tr>
</tbody>
</table>
## Database internals

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL_ONE</td>
<td>Returns a response from the closest replica in the local datacenter.</td>
<td>Achieves a consistency level of ONE without cross-datacenter traffic, which is desirable for multiple datacenter clusters. For security and quality reasons, use this consistency level in an offline datacenter. If an offline node goes down, LOCAL_ONE prevent automatic connection to online nodes in other datacenters.</td>
</tr>
<tr>
<td>SERIAL</td>
<td>Allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit the transaction as part of the read. Similar to QUORUM.</td>
<td>Use to read the latest value of a column after a user has invoked a lightweight transaction to write to the column. The database then checks the inflight lightweight transaction for updates and, if found, returns the latest data.</td>
</tr>
<tr>
<td>LOCAL_SERIAL</td>
<td>Same as SERIAL, but confined to the local datacenter. Similar to LOCAL_QUORUM.</td>
<td>Use to achieve linearizable consistency for lightweight transactions.</td>
</tr>
<tr>
<td>ONE</td>
<td>Returns a response from the closest replica, as determined by the snitch. By default, a read repair runs in the background to make the other replicas consistent.</td>
<td>Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write.</td>
</tr>
<tr>
<td>TWO</td>
<td>Returns the most recent data from two of the closest replicas.</td>
<td>Similar to ONE.</td>
</tr>
<tr>
<td>THREE</td>
<td>Returns the most recent data from three of the closest replicas.</td>
<td>Similar to TWO.</td>
</tr>
</tbody>
</table>

### How QUORUM is calculated

The QUORUM level writes to the number of nodes that make up a quorum. A quorum is calculated, and then rounded down to a whole number, as follows:

\[
\text{quorum} = \left( \frac{\text{sum of replication factors}}{2} \right) + 1
\]

The sum of all the replication_factor settings for each datacenter is the sum_of_replication_factors.

\[
\text{sum of replication factors} = \text{datacenter1\_RF} + \text{datacenter2\_RF} + \ldots + \text{datacenter\_n\_RF}
\]

For example, using a replication factor of 3, a quorum is 2 nodes \((\frac{3}{2}) + 1 - 2\). The cluster can tolerate one replica down.

**Examples:**

- Using a replication factor of 6, a quorum is 4 \((\frac{6}{2}) + 1 - 4\). The cluster can tolerate 2 replicas down.
- In a two-datacenter cluster where each datacenter has a replication factor of 3, a quorum is 4 nodes \((\frac{6}{2}) + 1 - 4\). The cluster can tolerate 2 replica nodes down.
- In a five-datacenter cluster where two datacenters have a replication factor of 3 and three datacenters have a replication factor of 2, a quorum is 7 nodes \((\frac{12}{2}) + 1 - 7\).

The more datacenters, the higher number of replica nodes need to respond for a successful operation.

Similar to QUORUM, the LOCAL_QUORUM level is calculated based on the replication factor of the same datacenter as the coordinator node. Even if the cluster has more than one datacenter, the quorum is calculated with only local replica nodes.

In EACH_QUORUM, every datacenter in the cluster must reach a quorum based on that datacenter’s replication factor for the write request to succeed. For every datacenter in the cluster, a quorum of replica nodes must respond to the coordinator node for the write request to succeed.

### How is the serial consistency level configured?

Serial consistency levels in DataStax Enterprise control lightweight transaction isolation. The consistency levels for lightweight transactions are shown in the following table.
### Database internals

#### Serial consistency levels

**Table 5: Serial Consistency Levels**

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERIAL</td>
<td>Achieves linearizable consistency for lightweight transactions by preventing unconditional updates.</td>
<td>This consistency level is for use only with lightweight transactions. Equivalent to QUORUM.</td>
</tr>
<tr>
<td>LOCAL_SERIAL</td>
<td>Same as SERIAL but confined to a single data center. A conditional write must be written to the commit log and memtable on a quorum of replica nodes in the same datacenter.</td>
<td>Same as SERIAL but used to maintain consistency locally (within the single local datacenter). Equivalent to LOCAL_QUORUM.</td>
</tr>
</tbody>
</table>

For local consistency, which enforces serialization within the local datacenter, use:

- LOCAL_SERIAL for LWT reads.
- LOCAL_QUORUM and LOCAL_SERIAL for LWT writes.

For global consistency, which enforces serialization across multiple datacenters, use:

- SERIAL for LWT reads.
- QUORUM and SERIAL for LWT writes.

Background information: The SERIAL consistency level defines the serial consistency level for the Paxos consensus of lightweight transactions. The learn phase uses a normal consistency level to define which read operations will be guaranteed to complete immediately if lightweight writes are occurring. The SERIAL consistency level is ignored for any query that is not a conditional update.

**How are read requests accomplished?**

There are three types of read requests that a coordinator node can send to a replica:

- A direct read request
- A digest request
- A background read repair request

In a direct read request, the coordinator node contacts one replica node. In a digest request, the coordinator node first contacts the replicas specified by the consistency level. The coordinator node sends requests to replicas that respond the fastest. The contacted nodes respond with a digest of the requested data. If multiple nodes are contacted, the rows from each replica are compared in memory for consistency.

If any replica nodes have out-of-date data, the coordinator node sends a background read repair, which forwards the result from the replica with the most recent data (based on the timestamp) back to the client. Read repair requests ensure that the requested row is made consistent on all replicas involved in a read query.

For illustrated examples of read requests, see Examples of read consistency levels.

**Rapid read protection using speculative_retry**

When the originally selected replica nodes are down or taking too long to respond, rapid read protection allows DataStax Enterprise (DSE) to still deliver read requests. If the table has been configured with the speculative_retry property, the coordinator node for the read request will retry the request with another replica node if the original replica node exceeds a configurable timeout value to complete the read request.
Figure 27: Recovering from replica node failure with rapid read protection
Examples of read consistency levels
The following diagrams show examples of read requests using different consistency levels:

- QUORUM in a single datacenter
- ONE in a single datacenter
- QUORUM in two datacenters
- LOCAL_QUORUM in two datacenters
- ONE in two datacenters
- LOCAL_ONE in two datacenters

Rapid read protection diagram shows how the speculative retry table property affects consistency.

Example: A single-datacenter cluster with a consistency level of QUORUM

In a single-datacenter cluster with a replication factor of 3 and a read consistency level of QUORUM, 2 of the 3 replicas \((\lfloor 3/2 \rfloor + 1 - 2)\) for the given row must respond to fulfill the read request. If the contacted replicas have different versions of the row, the replica with the most recent version will return the requested data. In the background, the third replica is checked for consistency with the first two, and if needed, a read repair is initiated for the out-of-date replicas.
Figure 28: Single datacenter cluster with 3 replica nodes and consistency set to QUORUM
Example: A single-datacenter cluster with a consistency level of ONE

In a single-datacenter cluster with a replication factor of 3 and a read consistency level of ONE, the closest replica for the given row is contacted to fulfill the read request.
Figure 29: Single datacenter cluster with 3 replica nodes and consistency set to ONE
Example: A two-datacenter cluster with a consistency level of QUORUM

In a two-datacenter cluster with a replication factor of 3 and a read consistency of QUORUM, 4 replicas for the given row must respond to fulfill the read request. The 4 replicas can be from any datacenter. In the background, the remaining replicas are checked for consistency with the first four. If needed, a read repair is initiated for the out-of-date replicas.
Figure 30: Multiple datacenter cluster with 3 replica nodes and consistency level set to QUORUM
Example: A two-datacenter cluster with a consistency level of LOCAL_QUORUM

In a two-datacenter cluster with a replication factor of 3 and a read consistency of **LOCAL_QUORUM**, 2 replicas in the same datacenter as the coordinator node for the given row must respond to fulfill the read request. In the background, the remaining replicas are checked for consistency with the first 2. If needed, a read repair is initiated for the out-of-date replicas.
Figure 31: Two-datacenter cluster with 3 replica nodes and consistency set to LOCAL_QUORUM
Example: A two-datacenter cluster with a consistency level of ONE

In a two-datacenter cluster with a replication factor of 3, and a read consistency of ONE, the closest replica for the given row, regardless of datacenter, is contacted to fulfill the read request.
Figure 32: Two-datacenter cluster with 3 replica nodes and consistency set to ONE

Diagram showing a two-datacenter cluster with 3 replica nodes and consistency set to ONE.
Example: A two-datacenter cluster with a consistency level of LOCAL_ONE

In a two-datacenter cluster with a replication factor of 3, and a read consistency of LOCAL_ONE, the closest replica for the given row in the same datacenter as the coordinator node is contacted to fulfill the read request.
Figure 33: Two-datacenter cluster with 3 replica nodes and consistency set to LOCAL_ONE
How are write requests accomplished?

The coordinator node sends a write request to all replicas that own the row being written. As long as all replica nodes are available, they will get the write regardless of the consistency level specified by the client. The write consistency level determines how many replica nodes must respond with a success acknowledgment for the write to be considered successful. Success means data was written to the commit log and the memtable.

The coordinator node forwards the write to replicas of that row. After the coordinator node receives write acknowledgements from the number of nodes specified by the consistency level, the coordinator responds to the client.

- If the coordinator cannot write to enough replicas to meet the requested consistency level, it throws an Unavailable exception and does not perform any writes.
- If there are enough replicas available but the required writes do not finish within the timeout window, the coordinator throws a Timeout exception.

For example, in a single-datacenter, 10-node cluster with a replication factor of 3, an incoming write will go to all three nodes that own the requested row. If the write consistency level specified by the client is ONE, the first node to complete the write responds back to the coordinator, which then proxies the success message back to the client. A consistency level of ONE means it is possible that two of the three replicas can miss the write if they are down when the request is made. If a replica misses a write, the row is made consistent later using one of the built-in repair mechanisms: hinted handoff, read repair, or anti-entropy node repair.
Figure 34: Single datacenter cluster with 3 replica nodes and consistency set to ONE
Multiple datacenter write requests

In multiple datacenter deployments, DataStax Enterprise (DSE) optimizes write performance by choosing one coordinator node. The coordinator node contacted by the client application forwards the write request to one replica in each of the other datacenters, with a special tag to forward the write to the other local replicas.

If the write consistency levels is LOCAL_ONE or LOCAL_QUORUM, only the nodes in the same datacenter as the coordinator node must respond to the client request for the request to succeed. Use either LOCAL_ONE or LOCAL_QUORUM to reduce geographical latency and lessen the impact on response times of client write requests.
Figure 35: Multiple datacenter cluster with 3 replica nodes and consistency set to LOCAL_QUORUM
CQL (Cassandra Query Language) is a query language for the DataStax Enterprise database.
See CQL for DataStax Enterprise 6.0.