Motivation

Anti-entropy (Apache Cassandra repairs) is essential for every Apache Cassandra cluster to fix data at rest inconsistencies. Frequent data deletions and downed nodes are common causes of data inconsistency. A few open-source orchestration solutions that trigger repair externally are available, as many large users have needed to figure out a scalable repair solution. However, multiple custom solutions have led to a lot of confusion in the community. Therefore, the repair activity, like Compaction, should be an integral part of Cassandra to call it a complete solution.

The proposal is to align one solution among the existing solutions and make it part of the core Cassandra. Here is the design for one of the solutions:

Inside Cassandra, there are multiple repairs we would have to schedule:
1) Full Repair
2) Incremental Repair
3) Paxos Repair
4) Preview Repair

The design of the scheduler should be capable of extending multiple repair categories with a minimal code change, and all repair types should progress automatically with minimal manual intervention.
Migrating[1] (and rollback) to/from incremental repair has been extremely challenging, especially in a large fleet. One of the design principles is to make it almost touchless from the operator's point of view.

The Scheduler

Keeping the above motivation in mind, this design embarks on our journey to have the repair orchestration inside Cassandra itself, which will repair the entire ring.

A dedicated thread pool is assigned to the repair scheduler at a higher level. The repair scheduler inside Cassandra maintains a new replicated table under a distributed *system_distributed* keyspace. This table maintains the repair history for all the nodes, such as when it was repaired the last time, etc. The scheduler will pick the node(s) that run the repair first and continue orchestration to ensure every table and all of their token ranges are repaired. The algorithm can also run repairs simultaneously on multiple nodes and splits the token range into subranges with the necessary retry to handle transient failures. This has been running for some time, and over that period has become reliable enough to configure it to just auto-start with a new Cassandra cluster, requiring no operator intervention.

Due to this fully automated repair scheduler inside Cassandra there is no dependency on the control plane, significantly reducing our operational overhead.

Detailed Design

This section deep-dives into the design by categorizing it into three parts:

1. Global view of all nodes about which node is currently running repair, and the last time it ran the repair.
2. Supporting multiple repair types, each working independently of the other without any interference.
3. Flow diagram showing how a given repair works on a Cassandra node

Global View

Cassandra's *system_distributed* keyspace has already been replicated. The idea here is to add two tables with the following schema to have a globally consistent view among the nodes:

Schema

Table 1 auto_repair_history

This table tracks the repair status for each node for a given repair type.

Column Name	Data Type	Description
repair_type	text (PK)	Repair type; possible values: > full > incremental
host_id	uuid (CK)	UUID of a Cassandra node
repair_start_ts	timestamp	Timestamp when a node started repairing
repair_finish_ts	timestamp	Timestamp when a node completes repairing
repair_turn	text	The node's current repair status, possible values: > > MY_TURN: The node is running repair. > MY_TURN_DUE_TO_PRIORITY: The node is running repair because the admin explicitly prioritized it. This is not a common scenario.
witness_deleted_hosts	set<uuid>	It is useful to determine when to delete the metadata entry of **this** table for the node that departed the ring. The scheduler uses Gossip as the source of truth for available nodes in the ring. When a node leaves the ring, then the scheduler would have to delete its metadata entry from this table for that node. In the worst case, if nodes have different views of the topology, then it could falsely delete this metadata entry. Let's understand with an example - if there are five nodes in the ring: N1, N2, N3, N4, N5 and N5 leaves the ring. When N1, N2, N3, and N4 go through a check, they will all append as witnesses if their Gossip view tells N5 is deleted. repair_type: Full Host_id: N5

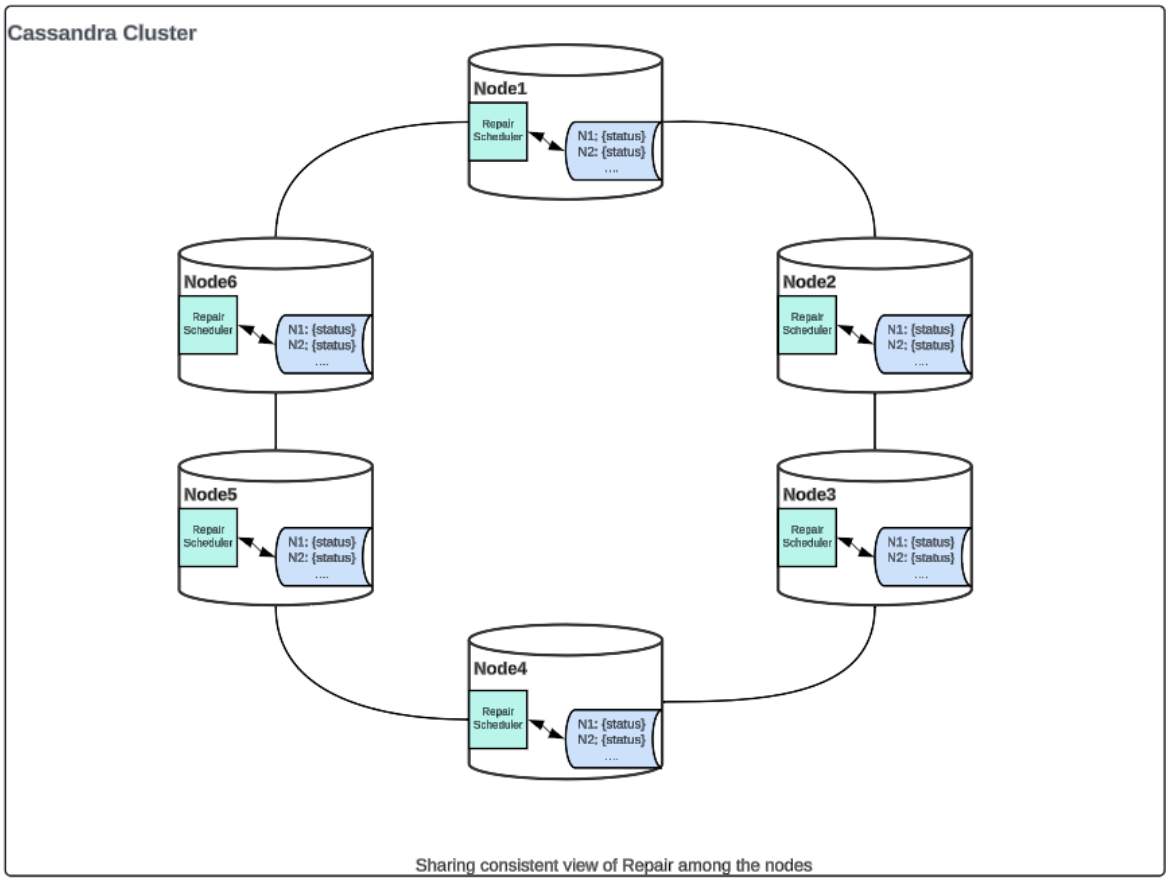
		deleted_host: <N2,N3,N4,N5> Now, anytime the scheduler sees the quorum number of live nodes (3 out of 4) have witnessed N5's deletion, then it can safely remove N5's metadata entry. This won't be needed after CEP-21 (Transactional Metadata).
delete_hosts_update_time	timestamp	The last time the deleted_hosts was updated. In the example above, if N5 were falsely marked as deleted by N1, we would have to clear N5's deleted_hosts column; after 2 hours, deleted_hosts would be cleared. The 2 hours is chosen because a topology mismatch should generally settle down within that duration.
force_repair	boolean	If this flag is set, then the node will go through the repair cycle regardless. The default is "false" This is useful if an admin wants to force one or more nodes to undergo repair, say, when a node restarts, before decommissioning. An admin can set this flag through *nodetool*. When the admin sets this flag, the scheduler's natural list of nodes will still continue the repair, but on top of that, these additional nodes will also undergo repair.

Table 2 auto_repair_priority

This table tracks any explicit priority set by the admin. By default, the scheduler will select the nodes' based on the oldest `repair_finish_ts` among all the nodes in a cluster. If, for whatever reason, an admin wants to prioritize a few nodes, then they can prioritize those nodes through a *nodetool* command. Those priority nodes are stored in this table, and when set, those nodes are prioritized over other nodes.

Column Name	Data Type	Description
repair_type	text (PK)	Repair type; possible values: > full > incremental
repair_priority	set<uuid>	List of nodes the schedulers should prioritize repair.

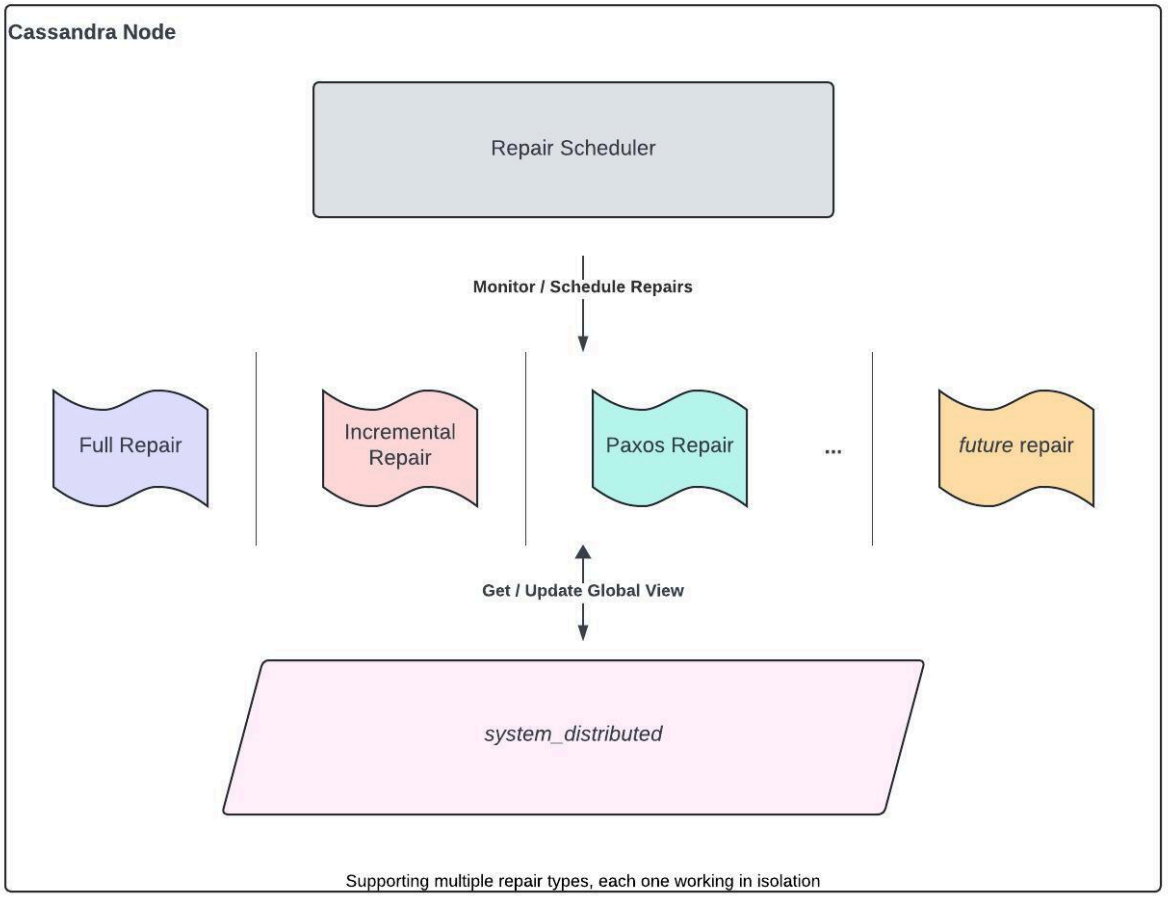
The diagram below explains that each node in a Cassandra cluster has the same global view of the above two tables. As a result, everyone can make a deterministic decision about whether to undergo the repair.

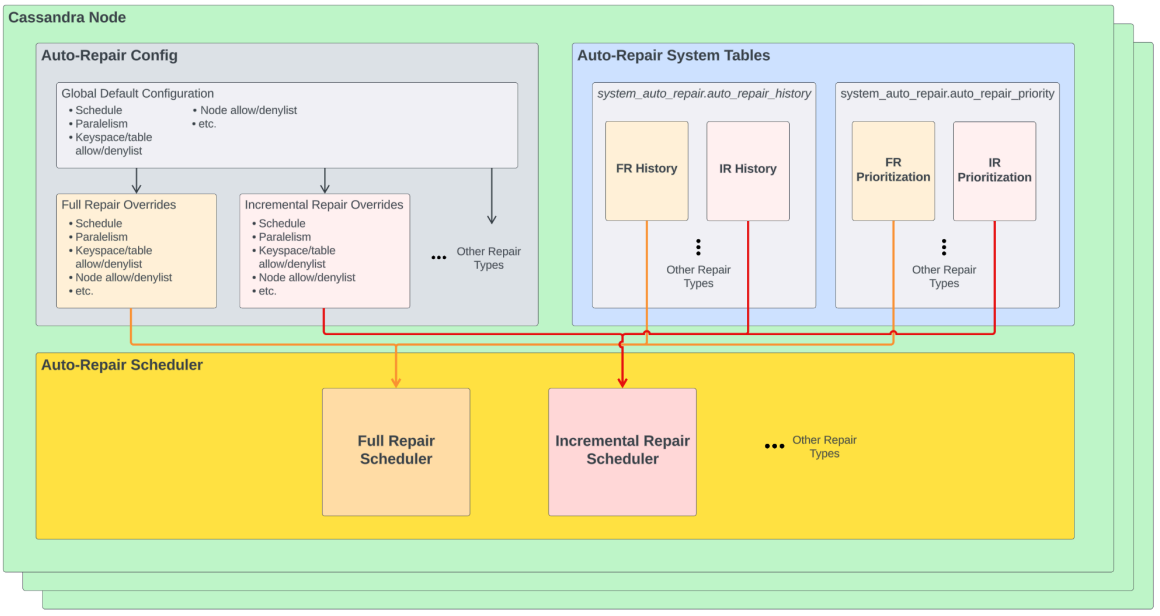


Support For Multiple Repair Types

Apache Cassandra needs to run multiple repair types to guarantee data consistency. The scheduler can run each repair type as a separate lightweight thread inside a Cassandra, and that thread will continue to track the lifecycle of a given repair type. The thread does not consume many resources because it runs every 5 minutes, reads/updates the metadata tables above-mentioned, and then schedules repair sessions serially. The repair session would do the actual anti-entropy, the exact mechanism when the repair is triggered through a traditional nodetool command, a.k.a. `nodetool repair`.

Since each repair type is managed through a separate thread, and the status for each repair type is also tracked separately in the metadata tables, they operate independently without interfering with each other, as shown in the diagram below.





Repair Flow On A Node

This section covers a detailed flow state transition for each repair type. Let's use *Full* repair as an example. The *Incremental* repair would work the same way.

The *Full* repair scheduler thread in Cassandra would work as follows:

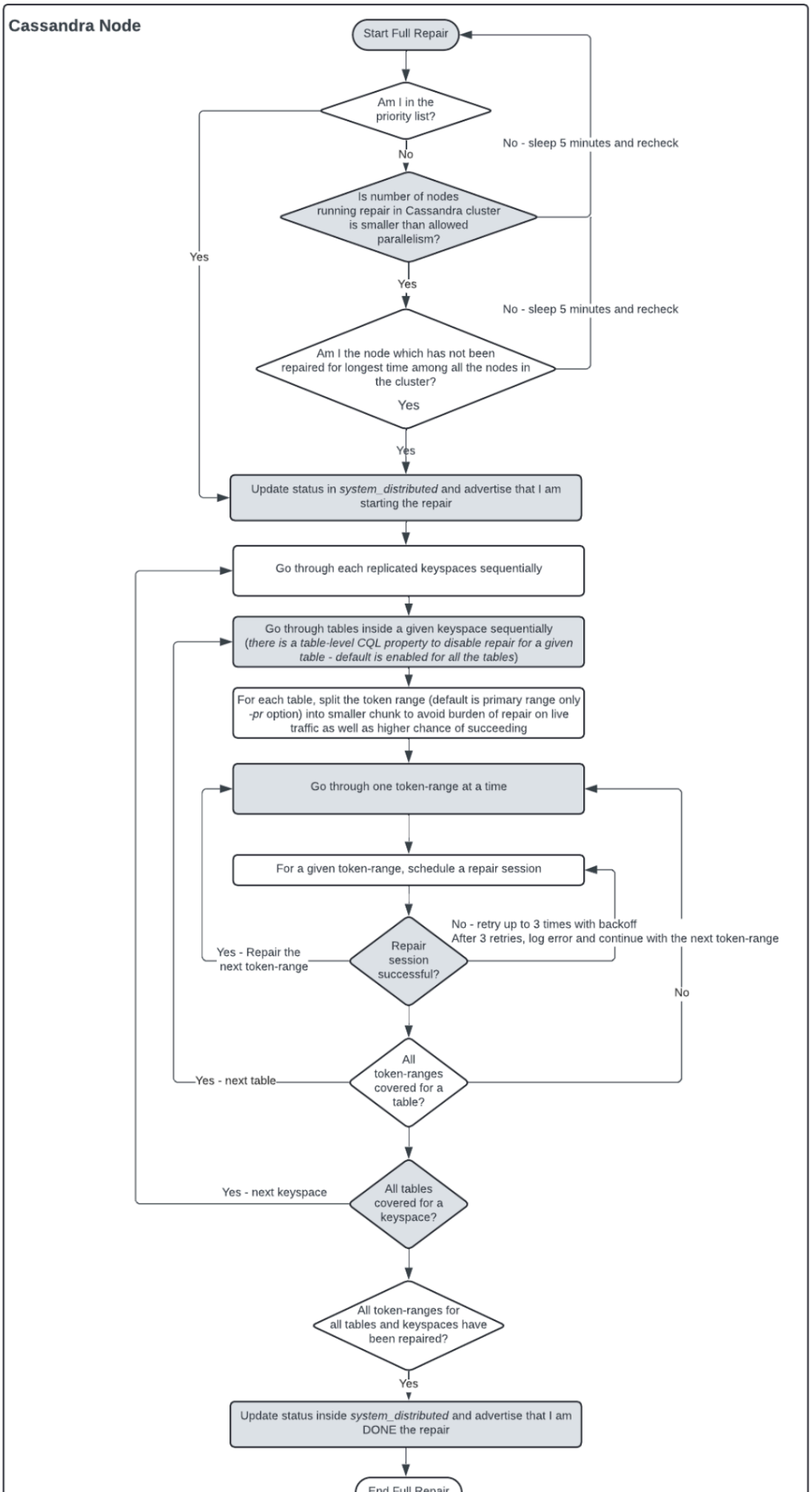
- Periodic check: At every 5-minute interval, it will check the global view of the repair in the cluster and determine whether this node should start a repair.
- Priority check: Pull the priority from auto_repair_priority and see if it is listed in the priority. If so, then start repairing.
- Parallelism: Pull the global view from auto_repair_history on which nodes run repair. Start repairing if the total number of nodes currently running repair in the Cassandra is smaller than the allowed parallelism.
- Announce Starting: Once the node decides it's their turn to repair, update auto_repair_history column `repair_start_ts` with the current timestamp.
- All Keyspaces: The node will consider all the replicated keyspaces (skipping `LocalStrategy`) and process them sequentially.
- All Tables: The node will consider all the tables inside a keyspaces and process them sequentially. There will be new CQL table properties to control enabling/disabling at a table-level granularity. By default, they are enabled, which means all tables inside a keyspace will go through a repair cycle.

```
$> CREATE TABLE t1 (
    a int PRIMARY KEY,
    b int,
    c int
) WITH additional_write_policy = '99p'
….
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p'
    AND repair_full = {'enabled': 'true'}
    AND repair_incremental = {'enabled': 'true'};

$> ALTER TABLE t1 WITH repair_incremental = {'enabled':'false'}
```

- Split Token Range: A repair can succeed if the data it touches is small enough; otherwise, it fails and jeopardizes the live traffic. By default, the token range will be broken down into 16 sub-ranges, and each range will go through the repair sequentially.
- Repair A Range: A selected token range will go through the repair cycle with a timeout set at three hours because, many times, a repair session just hangs forever. If a repair session fails, then it will be retried three times. If it fails, the scheduler will log a failure message, emit a metric, and then move to the next token range. Conversely, if it succeeds, then it emits a *success* metric.
- Announce Ending: Once the node completes all of the keyspaces, tables, and token ranges, it will update the auto_repair_history column `repair_end_ts` with the current timestamp.

Here is the flow diagram that captures the state transition on each node.


Cassandra Node
Start Full Repair
Am I in the priority list?
No - sleep 5 minutes and recheck
No
Is number of nodes running repair in Cassandra cluster is smaller than allowed parallelism?
Yes
Yes
No - sleep 5 minutes and recheck
Am I the node which has not been repaired for longest time among all the nodes in the cluster?
Yes
Yes
Update status in system_distributed and advertise that I am starting the repair
Go through each replicated keyspaces sequentially
Go through tables inside a given keyspace sequentially (there is a table-level CQL property to disable repair for a given table - default is enabled for all the tables)
For each table, split the token range (default is primary range only -pr option) into smaller chunk to avoid burden of repair on live traffic as well as higher chance of succeeding
Go through one token-range at a time
For a given token-range, schedule a repair session
Repair session successful?
No - retry up to 3 times with backoff
After 3 retries, log error and continue with the next token-range
Yes - Repair the next token-range
No
All token-ranges covered for a table?
Yes - next table
All tables covered for a keyspace?
Yes - next keyspace
All token-ranges for all tables and keyspaces have been repaired?
Yes
Update status inside system_distributed and advertise that I am DONE the repair
End Full Repair

Configuration

Table Level

A given repair type can be enabled or disabled per table level through table-level parameters

```
$> CREATE TABLE t1 (
    a int PRIMARY KEY,
    b int,
    c int
) WITH additional_write_policy = '99p'
….
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p'
    AND repair_full = {'enabled': 'true'}
    AND repair_incremental = {'enabled': 'true'};

$> ALTER TABLE t1 WITH repair_incremental = {'enabled':'false'}
```

YAML Configuration

Cassandra.yaml Snippet

```
auto_repair:
  enabled: true
  repair_type_overrides:
    full:
      enabled: true
      number_of_repair_threads: 2
      repair_max_retries: 2
      repair_primary_token_range_only: true
    incremental:
      enabled: true
      number_of_repair_threads: 1
      repair_max_retries: 3
      repair_primary_token_range_only: false
```

The following Yaml configuration options are supported, with a nodetool capability to get/set them.

Repair Scheduler Global Settings

Param	Default	Description
enabled	false	enable/disable auto repair globally, overrides all other settings. It can be modified dynamically. If it is set to false, then no repair will be scheduled, including full and incremental repairs by this framework. If it is set to true, then this repair scheduler will consult another config available for each RepairType, and based on that config, it will schedule repairs. This flag controls the creation of a new lightweight monitoring thread. Hence, it is not supported dynamically. Once a thread is spun, another flag (in the below) will determine whether a given repair type is allowed. In short, we have two flags controlling the behavior:

		1. This flag Controls whether to create a thread or not 2. Another flag below controls whether we want to allow a given repair or not. That flag can be controlled dynamically so that admin can stop repair anytime dynamically.
repair_check_interval	5m	The interval between successive checks for the repair scheduler to check if either the ongoing repair is completed or if none is going, then check if it's time to schedule or wait. The default is 5m. Users should not have to configure it at all. It just tells the interval between two subsequent checks. Theoretically, in a rare case, an admin might want to. If they have an extremely tiny table, data consistency is very important, and they want to repair to be scheduled every 2m interval, the interval could be reduced from 5 m to, say, 10s.
history_clear_delete_hosts_buffer_interval	2h	When any nodes leave the ring, then the repair schedule needs to adjust the order, etc. The repair scheduler keeps the deleted hosts information in its persisted metadata for the defined interval in this config. This information is useful so the scheduler is absolutely sure that the node is indeed removed from the ring, and then it can adjust the repair schedule accordingly. So, the duration in this config determines for how long deleted host's information is kept in the scheduler's metadata.
repair_max_retries	3	Number of times to retry a failed repair session before moving it to the next repair session.
repair_retry_backoff	1m	The back-off time for retrying a repair session
incremental_repair_disk_headroom_reject_ratio	0.2	This setting is only applicable if you have incremental repair enabled. At least 20% of disk must be unused to run incremental repair. Incremental repair could make the disk 100% full in the worst-case scenario and it can jeopardize the live-traffic. This setting will automatically stop the ongoing and future incremental repairs if the current disk usage is >80%. More details here.

Repair Type Settings

Parameter	**Default**	**Description**
repair_type_overrides	N/A	Determines which repair to run. Possible values: *> full* *> incremental* *This is how it would look in the yaml* *auto_repair:* *enabled: true* *repair_type_overrides:* ***full:*** *enabled: true* ***incremental:***

		enabled: false
enabled	false	enable/disable auto repair for the given repair type
repair_by_keyspace	false	auto repair is default repair table by table, if this is enabled, the framework will repair all the tables in a keyspace in one go.
number_of_subranges	16	The number of subranges to split each to-be-repaired token range into: the higher this number, the smaller the repair sessions will be. How many subranges to divide one range into? The default is 1. If you are using v-node, say 256, then the repair will always go one v-node range at a time; this parameter, additionally, will let us further subdivide a given v-node range into subranges. With the value “1” and v-nodes of 256, a given table on a node will undergo the repair 256 times. But with a value “2,” the same table on a node will undergo a repair 512 times because every v-node range will be further divided by two. If you do not use v-nodes or the number of v-nodes is pretty small, say 8, setting this value to a higher number, say 16, will be useful to repair on a smaller range, and the chance of succeeding is higher. But this behaviour can be completely overridden and one can specify their own algorithm for the split. This was discussed with Chris and Andy extensively and then this new option "token_range_splitter" (below property) got added.
token_range_splitter	DefaultAutoRepairTokenSplitter	* DefaultAutoRepairTokenSplitter.class: This is the default option. It splits the tokens based on the token ranges owned by this node divided by the number of 'number_of_subranges' * UnrepairedBytesBasedTokenRangeSplitter: Splits token ranges based on the data size in SSTable. It can be configured as follows, basically it has an option to add a cap on one repair session as well max cap at a table level token_range_splitter: class_name:UnrepairedBytesBasedTokenRangeSplitter parameters: max_bytes_per_schedule: 200MiB subrange_size: 100MiB
number_of_repair_threads	1	The number of repair threads to run for a given invoked Repair Job. Once the scheduler schedules one repair session, then how many threads to use inside that job will be controlled through this parameter.

		This is similar to -j for repair options for the nodetool repair command.
parallel_repair_count	0	The number of repair sessions that can run in parallel in a single group The number of nodes running repair parallelly. If parallel repaircount is set, it will choose the larger value of the two. The default is 3. This configuration controls how many nodes would run repair in parallel. The value “3” means, at any given point in time, at most 3 nodes would be running repair in parallel. These selected nodes can be from any datacenters. If one or more node(s) finish repair, then the framework automatically picks up the next candidate and ensures the maximum number of nodes running repair do not exceed “3”.
parallel_repair_percentage	3	the number of repair sessions that can run in parallel in a single group as a percentage of the total number of nodes in the group [0,100]. The percentage of nodes in the cluster that repair parallelly. If a parallel_repair_count is set, it will choose the larger value of the two. The problem with a fixed number of nodes (the above property) is that in a large-scale environment, the nodes keep getting added/removed due to elasticity, so if we have a fixed number, then manual interventions would increase because, on a continuous basis, operators would have to adjust to meet the SLA. The default is 3%, which means that 3% of the nodes in the Cassandra cluster would be repaired in parallel. So now, if a fleet, an operator won't have to worry about changing the repair frequency, etc., as overall repair time will continue to remain the same even if nodes are added or removed due to elasticity. Extremely fewer manual interventions as it will rarely violate the repair SLA for customers
sstable_upper_threshold	10000	The upper threshold of SSTables allowed to participate in a single repair session Threshold to skip a table if it has too many sstables. The default is 10000. This means, if a table on a node has 10000 or more SSTables, then that table will be skipped. This is to avoid penalizing good tables (neighbors) with an outlier. A metric "SkippedTokenRangesCount" will be

		incremented if token ranges are skipped. So, the operator needs to set up an alarm on this metric.
min_repair_interval	24h	The minimum time is in hours between repairing the same node again. This is useful for extremely tiny clusters, say five nodes, which finish repair quickly. The default is 24 hours. This means that if the scheduler finishes one round on all the nodes in < 24 hours. On a given node, it won't start a new repair round until the last repair conducted on a given node is < 24 hours.
ignore_dcs	<Empty>	Specifies a denylist of data centers to not repair. This is useful if you want to completely avoid running repairs in one or more data centers. By default, it is empty, i.e., the framework will repair nodes in all the data centers.
repair_primary_token_range_only	true	Set this 'true' if AutoRepair should repair only the primary ranges owned by this node; else, 'false' is the same as -pr in nodetool repair options.
force_repair_new_node	false	Configures whether to force immediate repair on the newly joined node - default it is set to 'false'; this is useful if you want to repair new nodes immediately after they join the ring.
table_max_repair_time	6h	The maximum time that a repair session can run for a single table. Max time for repairing one table on a given node, if exceeded, skip the table. The default is 6 hours. Let's say there is a Cassandra cluster in that there are 10 tables belonging to 10 different customers. Out of these 10 tables, 1 table is humongous. Repairing this 1 table, say, takes five days in the worst case, but others could finish in just 1 hour. Then we would penalize nine customers just because of one bad actor, and those nine customers would ping an operator and would require a lot of back-and-forth manual interventions, etc. So, the idea here is to penalize the outliers instead of good candidates. This can easily be configured with a higher value if we want to disable the functionality.
repair_session_timeout	3h	Repair session timeout - this is applicable for each repair session. The major issue with Repair is a session sometimes hangs, so this timeout is useful to unblock such problems
mv_repair_enabled	false	The default is 'true'. This flag determines whether the auto-repair framework needs to run anti-entropy, a.k.a, repair on the MV

		table.
initial_scheduler_delay	15m	The minimum delay after a node starts before the scheduler starts running repair

Nodetool

Here is what an example nodetool output would look like.

nodetool getautorepairconfig

All the YAML configuration mentioned above can be read through.

```
NAME
        nodetool getautorepairconfig - Print autorepair configurations
SYNOPSIS
        nodetool [(-h <host> | --host <host>)] [(-p <port> | --port <port>)]
                [(-pp | --print-port)] [(-pw <password> | --password <password>)]
                [(-pwf <passwordFilePath> | --password-file <passwordFilePath>)]
                [(-u <username> | --username <username>)] getautorepairconfig
OPTIONS
        -h <host>, --host <host>
            Node hostname or ip address
        -p <port>, --port <port>
            Remote jmx agent port number
        -pp, --print-port
            Operate in 4.0 mode with hosts disambiguated by port number
        -pw <password>, --password <password>
            Remote jmx agent password
        -pwf <passwordFilePath>, --password-file <passwordFilePath>
            Path to the JMX password file
        -u <username>, --username <username>
            Remote jmx agent username
```

Example

```
$> nodetool getautorepairconfig
repair scheduler configuration:
       repair eligibility check interval: 5m
       TTL for repair history for dead nodes: 2h
       max retries for repair: 3
       retry backoff: 60s
configuration for repair type: full
       enabled: true
       minimum repair interval: 1h
       repair threads: 1
       number of repair subranges: 16
       priority hosts:
       sstable count higher threshold: 10000
       table max repair time in sec: 6h
       ignore datacenters:
       repair primary token-range: true
       number of parallel repairs within group: 3
       percentage of parallel repairs within group: 3
       mv repair enabled: false
       initial scheduler delay: 1m
       repair setssion timeout: 3h
configuration for repair type: incremental
       enabled: true
       minimum repair interval: 15m
       repair threads: 1
       number of repair subranges: 16
```

```
       priority hosts:
       sstable count higher threshold: 10000
       table max repair time in sec: 6h
       ignore datacenters:
       repair primary token-range: true
       number of parallel repairs within group: 3
       percentage of parallel repairs within group: 3
       mv repair enabled: false
       initial scheduler delay: 1m
       repair setssion timeout: 3h
```

nodetool setautorepairconfig

All the YAML configuration mentioned above can be updated dynamically through nodetool without any Cassandra restarts, and the scheduler will adjust! If an admin decides to tune or stop the ongoing repair, it can be done quickly through the *nodetool*.

```
NAME
        nodetool setautorepairconfig - sets the autorepair configuration
SYNOPSIS
        nodetool [(-h <host> | --host <host>)] [(-p <port> | --port <port>)]
                [(-pp | --print-port)] [(-pw <password> | --password <password>)]
                [(-pwf <passwordFilePath> | --password-file <passwordFilePath>)]
                [(-u <username> | --username <username>)] setautorepairconfig
                [(-t <repair type> | --repair-type <repair type>)] [--]
                <autorepairparam> <value>
OPTIONS
        -h <host>, --host <host>
            Node hostname or ip address
        -p <port>, --port <port>
            Remote jmx agent port number
        -pp, --print-port
            Operate in 4.0 mode with hosts disambiguated by port number
        -pw <password>, --password <password>
            Remote jmx agent password
        -pwf <passwordFilePath>, --password-file <passwordFilePath>
            Path to the JMX password file
        -t <repair type>, --repair-type <repair type>
            Repair type
        -u <username>, --username <username>
            Remote jmx agent username
        --
            This option can be used to separate command-line options from the
            list of argument, (useful when arguments might be mistaken for
            command-line options
        <autorepairparam> <value>
            autorepair param and value. Possible autorepair parameters are as
            following: [start_scheduler
|number_of_repair_threads|number_of_subranges|min_repair_interval|sstable_upper_threshold|enabled|
table_max_repair_time|priority_hosts|forcerepair_hosts|ignore_dcs|history_clear_delete_hosts_buffe
r_interval|repair_primary_token_range_only|parallel_repair_count|parallel_repair_percentage|mv_rep
air_enabled|repair_max_retries|repair_retry_backoff|repair_session_timeout]
```

Example

```
$> nodetool setautorepairconfig -t incremental number_of_repair_threads 2
$> nodetool setautorepairconfig -t full mv_repair_enabled false
```

nodetool autorepairstatus

Inspired by the famous `nodetool status`, implemented a similar command to know the host ids going through the repair cycle.

```
NAME
        nodetool autorepairstatus - Print autorepair status
SYNOPSIS
        nodetool [(-h <host> | --host <host>)] [(-p <port> | --port <port>)]
                [(-pp | --print-port)] [(-pw <password> | --password <password>)]
                [(-pwf <passwordFilePath> | --password-file <passwordFilePath>)]
                [(-u <username> | --username <username>)] autorepairstatus
                [(-t <repair type> | --repair-type <repair type>)]
OPTIONS
        -h <host>, --host <host>
            Node hostname or ip address
        -p <port>, --port <port>
            Remote jmx agent port number
        -pp, --print-port
            Operate in 4.0 mode with hosts disambiguated by port number
        -pw <password>, --password <password>
            Remote jmx agent password
        -pwf <passwordFilePath>, --password-file <passwordFilePath>
            Path to the JMX password file
        -t <repair type>, --repair-type <repair type>
            Repair type
        -u <username>, --username <username>
            Remote jmx agent username
```

Example

```
$> nodetool autorepairstatus -t incremental
Active Repairs
425cea55-09aa-46e0-8911-9f37a4424574


$> nodetool autorepairstatus -t full
Active Repairs
NONE
```

Observability

The idea here is to have all important stuff captured in metrics, such as time to run repair, skipped/failed/successful ranges, repair is progressing, entire cluster's repair time, etc., so an operator can set up dashboards and alarms accordingly and be rest assured that the automated repair scheduler is working as expected. The following metrics will be provided to show the repair progress in the cluster.

Metric Name

`org.apache.cassandra.metrics.AutoRepair.<MetricName>`

JMX MBean

`org.apache.cassandra.metrics:type=AutoRepair name=<MetricName> repairType=<RepairType>`

Name	Type	Description

RepairsInProgress	Gauge<Integer>	Repair is in progress on the node
NodeRepairTimeInSec	Gauge<Integer>	Time taken to repair the node in seconds
ClusterRepairTimeInSec	Gauge<Integer>	Time taken to repair the entire Cassandra cluster in seconds
LongestUnrepairedSec	Gauge<Integer>	Time since the last repair ran on the node in seconds
SucceededTokenRangesCount	Gauge<Integer>	Number of token ranges successfully repaired on the node
FailedTokenRangesCount	Gauge<Integer>	Number of token ranges failed to repair on the node
SkippedTokenRangesCount	Gauge<Integer>	Number of token ranges skipped on the node
SkippedTablesCount	Gauge<Integer>	Number of tables skipped on the node
TotalMVTablesConsideredForRepair	Gauge<Integer>	Number of materialized views considered on the node
TotalDisabledRepairTables	Gauge<Integer>	Number of tables on which the automated repair has been disabled on the node
RepairTurnMyTurn	Counter	Represents the node’s turn to repair
RepairTurnMyTurnDueToPriority	Counter	Represents the node’s turn to repair due to priority set in the automated repair
RepairTurnMyTurnForceRepair	Counter	Represents the node’s turn to repair due to force repair set in the automated repair

Reliable Incremental Repair Onboarding/Offboarding

Incremental repair does not work reliably for all workload types. Therefore, it is essential to have a smooth onboarding/offboarding process. The current Onboarding/offboarding[1] incremental repair is quite painful. The challenge amplifies exponentially if we want to do this on a large Cassandra cluster and an extensive fleet of Cassandra.
Following changes are made to make the incremental repair more reliable and scalable to manage in a large-fleet.

Features

No Restart

By default when we want to onboard/offboard to incremental repair, then it requires setting/resetting each and every SSTables followed by a rolling restart. Restarts delays the issue of mitigation, especially for large Cassandra clusters. Hence a following nodetool command is included that will not require restart, and it will not even ask for any keyspace/tables.

```
NAME
        nodetool sstablerepairedset - Set the repaired state of SSTables for
        given keyspace/tables
SYNOPSIS
        nodetool [(-h <host> | --host <host>)] [(-p <port> | --port <port>)]
                [(-pp | --print-port)] [(-pw <password> | --password <password>)]
                [(-pwf <passwordFilePath> | --password-file <passwordFilePath>)]
                [(-u <username> | --username <username>)] sstablerepairedset
                [--is-repaired] [--is-unrepaired] [--really-set] [--] <keyspace>
                [<table...>]
OPTIONS
        -h <host>, --host <host>
            Node hostname or ip address
        --is-repaired
            Set SSTables to repaired state.
        --is-unrepaired
            Set SSTables to unrepaired state.
        -p <port>, --port <port>
            Remote jmx agent port number
        -pp, --print-port
            Operate in 4.0 mode with hosts disambiguated by port number
        -pw <password>, --password <password>
            Remote jmx agent password
        -pwf <passwordFilePath>, --password-file <passwordFilePath>
            Path to the JMX password file
        --really-set
            Really set the repaired state of SSTables. If not set, only print
            SSTables that would be affected.
        -u <username>, --username <username>
            Remote jmx agent username
        --
            This option can be used to separate command-line options from the
            list of argument (useful when arguments might be mistaken for
            command-line options
        [<keyspace> [<table...>]]
            The keyspace optionally followed by one or more tables. If no keyspace is
given, then it will pick all non-LOCAL keyspaces.
```

Safety Against 100% Disk Full

Incremental repair can lead to 100% disk full in corner-case scenarios. To alleviate the disk usage, a new mechanism has been added in that if the disks are 80% full, then the current and futuristic incremental repair will automatically stop!

A following new YAML configuration has been added, which controls the allowed headroom for incremental repair to proceed. More details about this setting has been added under YAML configuration section.

```
incremental_repair_disk_headroom_reject_ratio: 0.2
```

Protection Against Materialized View and CDC

It is not recommended (due to existing bugs) to use the incremental repair on a table if a Materialized View and/or CDC is enabled. The scheduler additionally confirms whether the "write path" is disabled (CASSANDRA-17666) during the streaming. If the "write path" is not disabled, then it will skip incremental repair, else allow.

Next

We can see the above-mentioned features simplify the incremental repair, but it still requires a limited operator involvement. The plan is even to automate that part, so an operator just needs to configure a few settings, and that's it! The design to make the incremental repair onboarding/offboarding a fully automated experience has been discussed in this document.
Incremental Repair Migration

Scale

This design has been thoroughly tried and tested on an immense scale (hundreds of unique Cassandra clusters, tens of thousands of Cassandra nodes, with tens of millions of QPS) on top of 4.1 open-source. Please see more details here.

Source Code

The PR is currently on top of Apache Cassandra 4.1.6, and it includes everything described above.
Pull request: https://github.com/apache/cassandra/pull/3367/

Appendix

Feedback: Automate Repair in Cassandra

This section covers the feedback received as part of this design discussions from various folks, corresponding action items, and follow ups.

Mar 28, 2024

- Feedback from Andy Tolbert: Auto Repair Notes/Feedback Mar 28 - Google Docs

Apr 25, 2024

Jaydeep:

- Added an interface for token-range calculation so one can override with their own implementation. An example PR

Kristijonas:

- Added Option 4 for IR migration - nodetool command, which mutates SSTable repaired state at runtime.
- Implemented repair type agnostic auto-repair scheduler and tested in staging (at small scale), Option 1 for IR migration is now ready.
- Working on automating migration from repaired to unrepaired state:
    - Performed a survey of all use cases of the repairedAt field and identified where repairedAt should be ignored during repaired -> unrepaired migration.
    - Still need to address some concerns about the impact on transient replication

May 9, 2024

Kristijonas:

- IR is still running at a small scale within our fleet. I'm hoping to start the large-scale migration next week.
- Documented multiple options for the repaired -> unrepaired migration. Would like advice on which option we should go with.

Jaydeep:

- Prepare a new PR with the v2 framework and share it before May'23

May 15, 2024

Here is the PR with the v2 framework on top of 4.1.3, which covers the following functionality:

```
- AutoRepir v2 framework that covers both FR & IR schedulers
- An interface with a default implementation for TokenRange calculation
- Enable/Disable the repair flag as a new CQL Table property
```

With that, the only deliverable item pending is the FR <-> IR roll forward/rollback automation, which we will follow up on shortly. cc:

June 05, 2024

Generate Pull Requests against the 4.1 branch (AI: Jaydeep)

- Here are the two PRs against the 4.1 branch
- 4.1.6 - https://github.com/apache/cassandra/pull/3367
- 4.1.3 - https://github.com/apache/cassandra/pull/3368

Aug 29, 2024

- 4.1.6 PR has been updated with all the bug fixes and minor improvements
- The CEP-37 has also been updated with the latest design details and a link to this doc