Redis Connect

Version 0.11.0

Table of Contents

Introduction

Redis Connect is a distributed platform that enables real-time event streaming, transformation, and propagation of changed-data events from heterogeneous data platforms (e.g., Oracle, MongoDB, Gemfire, etc.) to Redis Enterprise, Redis Cloud, and Azure Cache for Redis Enterprise.

Redis Connect can be easily installed, configured, and deployed to keep Redis continuously synced with committed operations persisted by the source to its transactional log, typically using a proprietary change data capture process.

With Redis Connect, users can continue their reliance on the system of record and, in parallel, offload read-intensive workloads to Redis for real-time performance and reduced costs.

Table of Contents

Key Terms

For efficiency, let's define the following terms to be used throughout the remainder of these docs:

checkpoint

Dictionary of key/value pairs that include a composition of offset-related fields for the purpose of uniquely identifying an exact row in the source's transactional log. See Offset Management for more details.

checkpoint database

By default, checkpoints are stored within the target however it is possible to configure a separate Redis database for the purposes of storing checkpoints. The primary use-case for this is to support jobs that do not use Redis as their target database (e.g. Splunk).

claim

Designation of a Redis Connect instance's sole responsibility for a job partition. See Job Execution for more details.

cluster

Redis Connect has a cloud-native and shared-nothing cluster architecture which allows any node/member/instance, with a connection to the Job Management Database, to operate stand-alone, become the leader of a clustered deployment, perform job execution, and/or provide passive high-availability.

connector

Dedicated thread(s) instantiated by each job partition for the purpose of querying the transactional log for batches of new changed-data events which will be stored within its transient in-memory queue.

data pipeline

Multi-stage event-driven workflow that transforms and propagates changed-data events in the order they were captured at the source. Each partition operates a replica of its own data pipeline in isolation.

instance

A single JVM process (a.k.a. cluster node/member) which participates in a Redis Connect cluster.

job

Unique name to label everything required for end-to-end replication from source to target including each partition's data pipeline, configuration, and metadata.

Job Claim Assignment Stream

Asynchronous event-messaging broker, implemented with Redis Streams, between the Job Reaper and JobClaimer(s) used for the sole purpose of claiming *UNASSIGNED* job partitions by Redis Connect instance(s).

partition

Replica of the data pipeline which enables linear scalability without losing order. Job partitions have no awareness nor direct coordination with each other, therefore can be deployed across one or multiple Redis Connect instance(s).

producer

Dedicated thread instantiated by each job partition for the purpose of polling the connector's in-memory queue so changed-data events can be published to the data pipeline in order.

quiesce

Allowing a data pipeline to process all queued changed-data events before stopping the job partition upon an unrecoverable error or manual administrative intervention (e.g. stop-job, migrate-job). The following exceptions will automatically bypass this process to avoid timeouts and deadlocks - io.lettuce.core.RedisException, InvalidChangeEventException, RedisTransactionFailureException, RedisWaitCommandTimeoutException, UnrecoverableTargetConnectionException, NullPointerException.

stage

Encapsulation of functional logic (i.e. target sink, custom transformation, detokenization, etc.) as part of a data pipeline.

sink

Pre-built stage that uses proprietary client libraries to write data to the target based on the selected data structure/model. They can be configured to also update the checkpoint within a transactional scope for Redis target(s).

source

Data platform, such as PostgreSQL, in which the changed-data events are captured/originate.

target

Data platform to which changed-data events will be replicated. With Redis Connect, the target is usually Redis.

task

represents LOAD job partitions. This labeling exists primarily for internal purposes so can be used synonymously with partitions.

transactional log

(a.k.a. binary log, change-stream, database log, journal, outbox-table, redo log, write-ahead log, etc.) is an ordered record of mutations executed by a DBMS. Each source maintains its own version of the transaction log's schema and offset identifiers. Transaction logs are not always supported in the form of disk-based files. In some cases, the log will be a change-stream or outbox table. Some sources require proprietary transaction log miner libraries to read log entries.

type

The following types are supported: **LOAD**, **STREAM**, and **PARTITIONED_STREAM**.

- **LOAD** (a.k.a. Initial Load) jobs directly query source table(s) at the point-in-time they are started. Initial loads can be horizontally partitioned for linear scalability, independent of the job's PARTITIONED_STREAM strategy (i.e. LOAD might need 10 partitions while the same job's PARTITIONED_STREAM only needs 2). This makes it feasible to complete the entire initial load within minutes (instead of hours) to comply with a bounded enterprise release window or recovery-time objective (RTO). Once each task partition completes its part of the initial load, it will automatically stop and release capacity back to the Redis Connect instance that claimed them. **Disclaimer:** Since each partition will query source table(s) there could be a momentary impact on routine transactions.
- **STREAM** jobs replicate changed-data events from the source's proprietary transactional log. Unlike LOAD jobs, STREAM jobs only stop in the event of an unrecoverable error or manual administrative intervention.
- **PARTITIONED_STREAM** jobs represent STREAM jobs that are partitioned. This type exists primarily for internal purposes so can be used synonymously with STREAM.

Source Distributions

Redis Connect can run within a container or by downloading the code and running it in your environment.

Download

Releases are distributed on GitHub. See the release history for the latest version.

Docker

You can also run and deploy Redis Connect using the following Docker Image.

Once the distribution archive is downloaded, the following directories will be extracted under */redis-connect*:

lib

JARs for Redis Connect and its dependencies.

extlib

JARs for custom stages and database drivers (when necessary such as Oracle and DB2). All JARs must exist within the directory before deployment.

config

Sample job configurations, credentials property files, and Grafana dashboard configurations.

bin

Scripts for running Redis Connect on Linux VMs, containerized environments, and Windows.

Supported Sources

Redis Connect supports seamless integration and continuous data replication with a growing set of heterogeneous data platforms including RDBMS, NoSQL, IMDG, and more. It can also be used as a one-time migration tool or perform periodic ETL / batch loads from sources that include delimited (e.g. CSV, TSV, PSV), JSON, and XML files.

The table below shows the current list of supported sources. Each source's name links to an end-to-end demo including a pre-installed source data platform, sample configuration, and data.

Table 1. Supported source data platforms

Data Platform	Supported Job type(s)	3rd Party Source Configuration Docs
DB2	LOAD	-
Files	LOAD	RIOT docs
Gemfire	LOAD & STREAM	Apache Geode Docs
MongoDB (Atlas*)	LOAD & STREAM	Debezium MongoDB docs
MySQL	LOAD & STREAM	Debezium MySQL docs
Oracle (AWS RDS Oracle*)	LOAD & STREAM	Debezium Oracle docs
PostgreSQL (AWS RDS PostgreSQL*)	LOAD & STREAM	Debezium PostgreSQL docs
Splunk	STREAM	Splunk HEC docs
SQLServer (Azure SQL Database*)	LOAD & STREAM	Debezium SQL Server docs
Vertica	LOAD	-

* Supported Cloud-Managed Services

Redis Connect Architecture

Overview

Redis Connect has a cloud-native and shared-nothing architecture which allows any cluster node/member/instance to perform both Job Management and Job Execution. Each Redis Connect instance is multi-threaded and segregates the control plane (Job Management) from the data plane (Job Execution) so administration does not impact performance.

Redis Connect instances are also multi-tenant and support the collocated execution of job partitions with heterogeneous sources in isolation, so they don't become noisy neighbors. For example, a single instance can simultaneously process a data pipeline with Gemfire as its source and another with Oracle as its source.

Redis Connect is compiled in JAVA and deploys on a platform-independent JVM which allows it to be agnostic of the underlying operating system (Linux, Windows, Containers, etc.) It was designed to minimize the use of infrastructure-resources and avoids complex dependencies on other distributed platforms such as Kafka and ZooKeeper. In fact, the majority of deployments will only require a single instance to perform both Job Management and Job Execution, while additional *m* instance(s) can be optionally deployed for N+*m* redundancy / high-availability.

NOTE	Redis Connect's multi-tenancy allows instances to vertically-scale without the overhead of requiring additional JVMs per each job partition or distinct source. See Capacity Estimation for more information.

Distributed Consensus

Redis Connect was built to execute mission-critical data pipelines on environments (including cloud) with potentially unbounded network delays and random failure scenarios. For this reason, it was implemented with a distributed and shared-nothing architecture which allows any Redis Connect instance to perform the function of Job Management or Job Execution or both.

Redis Connect only allows a single instance to act as the cluster leader at any given time. Similarly, only a single instance can claim a job partition. The consensus algorithms for these single leader elections are based on a distributed locking pattern and heartbeat mechanisms.

By default, at startup each instance will initiate its candidacy to become the cluster leader and begin to periodically check for new job claim requests in the Job Claim Assignment Stream. If

unsuccessful, the instance will continue to periodically attempt to gain distributed consensus based on the cluster.election.attempt.interval and job.claim.attempt.interval, respectively.

Redis Connect allows leadership claims to be renewed **indefinitely** which avoids unnecessary overhead and context-switching. To avoid deadlocks, claims are implemented as a lease with an auto-timeout mechanism via key-expiry (TTL) and a heartbeat notification for periodic renewal based on the cluster.leader.heartbeat.lease.renewal.ttl and job.claim.heartbeat.lease.renewal.ttl, respectively. The lease is used as a barrier to block new elections, and prevent split-brain scenarios, until the leader is no longer available or manually relinquishes its claim.

NOTE	The cluster leader will only relinquish its claim due to a process failure, elongated garbage-collection pause, network delay, etc. However, job ownership claims can be relinquished due to unrecoverable data pipeline errors or manual administrative intervention (e.g. stop-job, migrate-job).

Job Execution

- Offset Management
- Initial Load
- Delivery Guarantees
- Job Capacity
- Custom Stages

Overview

Each job partition is executed, end-to-end, within a single Redis Connect instance to reduce network I/O, context-switching, and garbage-collection pauses. This allows for predictable resource utilization, optimized performance, and clean failover between Redis Connect instances.

Job Execution (data plane) is a multi-threaded orchestration. It begins with a connector thread that periodically queries the source for batches of changed-data events and stores them within the job partition's (transient) in-memory queue. Concurrently, a producer thread periodically polls the in-memory queue, transforms each changed-data event, and publishes them to the job partition's data pipeline (in time-order).

NOTE	If the connector's polling interval is not explicitly set as a custom configuration (e.g. poll.interval.ms), it will use the producer's pollSourceInterval.

The data pipeline is implemented using a specialized engine, LMAX Disruptor, which was built to maximize concurrency/throughput and minimize garbage-collection/overhead. Since the data pipeline is an event-driven workflow, each stage can benefit from its own dedicated thread/vCore. See Production Readiness for more information on capacity estimation.

Back-pressure support and circuit-breakers are implemented at every level to avoid OOM failures and noisy neighbors. For example, if there is a slow consumer or a large unexpected spike of changes, the producer thread will bypass its polling interval until the data pipeline has enough

capacity to enqueue the next batch. The same is true at the connector level.

WARNING	Back pressure support is a last resort mechanism which will log all occurrences of its use. If warnings often show up in your logs, then it is a sign of an imbalance between source and target throughput. If it persists, see Production Readiness about tuning options and job partitioning.

Data pipelines only require a single sink stage, however, additional user-defined stages can be added to handle proprietary business logic, custom transformations, specialized detokenization, etc. See Custom Stages for more information.

NOTE	By default, Job Execution is enabled on every Redis Connect instance however can be disabled via job.claim.service.enabled. Disabling Job Execution does not prevent it from performing Job Management capabilities including acting as a REST/CLI endpoint.

NOTE	Redis Connect instances are stateless. All data streamed through its data pipelines is considered *data in transit* / *data in motion* / *data in-flight* which alleviates typical security concerns.

Offset Management

Most sources persist committed changed-data events into a proprietary transactional log via change data capture (CDC) processes. Each changed-data event within the log is marked with one or more position-markers to uniquely identify its offset. Within Redis Connect, specialized connectors embed libraries to traverse and identify theses position-maker(s).

Offset management is critical to maintain pipeline continuity, order, delivery-guarantees, and resiliency in the event of an error during job execution or startup from a specific row within the transaction log. Central to offset management are checkpoints which are a composition of offset-related fields for the purpose of uniquely identifying an exact row in the transaction log. Upon each iteration of the data pipeline, the sink stage with checkpointStageIndicator enabled, will commit the changed-data event's checkpoint to the target.

NOTE	A checkpoint database can be configured if the target is not chosen as its offset store. Checkpoints are never committed to the Job Management Database.

By enabling checkpointTransactionsEnabled, the checkpoint will be committed before the changed-data event to the target within a transactional scope. This strengthens delivery guarantees and improves performance by avoiding an additional network hop for iteration of the data pipeline.

NOTE	Enabling checkpointTransactionsEnabled will create 16384 checkpoint keys (one for each slot) within the target for **each job partition**. With ~250 bytes per checkpoint, each partition's estimated overhead is ~4MB in addition to a lightweight index to track and query the last committed. When disabled, there is only a single checkpoint key per job partition.

Once a Redis Connect instance has claimed responsibility for job execution, it must determine the offset from which to begin querying changed-data events. If no checkpoint yet exists, most connectors will begin from the beginning of the transaction log. For edge cases, see Initial Load on how to set the latest checkpoint manually.

Initial Load

LOAD job type (a.k.a. Initial Load) directly queries the source's table(s) at the point-in-time it is started via the **POST** /connect/api/v1/job/transition/start/{jobName}/LOAD REST API or CLI command.

Initial loads can be horizontally partitioned into tasks for linear scalability simply by configuring LOAD partitions. Tasks are the equivalent to PARTITIONED_STREAM job partitions in most ways including how they transform changed-data events and execution of their data pipeline. However, a key distinction is that after their workload is **COMPLETE**, tasks automatically release capacity back to the Redis Connect instance(s) that claimed them.

Initial load tasks can be configured independent of PARTITIONED_STREAM partitions for the same job. For example, an initial load might require 10 tasks to replicate a table with millions of rows within 10-20 minutes however only require 2 job partitions to handle 15k op/s during event-streaming. Furthermore, since jobs can include many tables and each table will have its own row count, initial load partitioning can be independently configured at the **table** level.

WARNING	Since multiple tasks will query the same source table, there could be a momentary impact on routine transactions.

By partitioning the initial load, administrators have control over its duration (i.e. minutes instead of hours) which allows for compliance with bounded enterprise release windows and tight recovery-time objectives (RTO). It also allows for initial loads to run on different infrastructure from long-running stream jobs for cost optimization.

NOTE	Linear scalability requires adequate resources for parallelization. Since each task requires the equivalent resources to a job partition, the same capacity estimation applies for maximum performance.

There are effectively two scenarios for performing initial load before starting event-streaming:

Job is supported by a dedicated CDC process
In this scenario, the initial load should start in coordination with enabling the CDC process so changed-data events will begin to be captured at the exact same time. Once the initial load is complete, the job can be started which would replicate all changed-data events in the order they were captured from the beginning of the transaction log.

Job is supported by a shared CDC process
In this scenario, the CDC process was already started for other means and therefore the transaction log is full of changed-data events that predate the initial load. To avoid overriding data, and unnecessarily increasing lag, before reaching consistency (potentially by hours), you will need to set the latest checkpoint manually.

Redis Connect supports an administrative capability dedicated for this purpose. First, we recommend that you capture the checkpoint schema specific to your job's source in a development environment by using the **GET** /connect/api/v1/job/checkpoint/{jobId} REST API/CLI. Next, you will need to query the source to identify the offset which occurred at the time you started the initial load. Copy the offset position marker(s) into the checkpoint schema and pass it as a parameter to the **POST** /connect/api/v1/job/checkpoint/{jobId} REST API/CLI before starting the job.

NOTE	The checkpoint schema of each source will be the same across all environments. Querying the source to identify an offset at a specific point-in-time might require the assistance of your DBA.

Delivery Guarantees

No Data Loss

Avoiding data loss is a **shared responsibility**. Redis Connect's offset management and stateless nature, prevents it from losing data as long as the source transactional log is available and includes the offsets stored within the last committed checkpoint.

WARNING	Most 24/7 production environments are at low risk of losing data. However, in non-production environments it is possible that no changed-data events are created over an extended period of time meanwhile a routine process truncates the transaction log. In this scenario, Redis Connect would be prevented from matching the offset during a recovery, or restart, which can lead to data loss if the transaction log cannot be recovered. If this scenario is possible in your production environment, see Production Readiness on best practices and capabilities to avoid data loss.

at-least-once

By default, Redis Connect jobs provide an *at-least-once* end-to-end delivery guarantee. This means that in the event of an unrecoverable error, a **single** changed-data event might be replicated again upon recovery / restart. In non-failure scenarios, events will always be replicated *exactly-once*.

exactly-once

Redis Connect jobs can be configured to make every effort for *exactly-once* delivery even if an unrecoverable error was to occur. By enabling checkpointTransactionsEnabled, the checkpoint will be committed before the changed-data event to the target within a transactional scope. If an unrecoverable error were to occur in the middle of that transaction, Redis Connect would seamlessly rollback the last committed checkpoint to its previous state. This would effectively result in the target having the exact overall same state as it had before the transaction began.

NOTE	When Redis is the target, rollback is also supported to handle replication issues between the primary and backup shards by enabling redis.wait.enabled.

Job Capacity

Redis Connect instances are multi-tenant and multi-threaded which comes with the many advantages described in Architecture Overview, however, it is not without risks. One of the major concerns is around resource utilization.

Resource utilization is important to maintain performance SLAs, avoidance of noisy neighbors, and unexpected crashes. For example, if a Redis Connect instance was deployed on server with only 4 vCores and claimed 4 job partitions, there would be a risk of timeouts due to context switching and resource starvation. See Capacity Estimation for more details.

For this reason, Redis Connect supports a maximum limit to the amount of job partitions (inclusive of initial load tasks) that can be concurrently claimed by a single instance via the job.claim.max.capacity configuration. This limit acts as a control which prevents the Job Claimer service from over-provisioning and is used as part of capacity validation during transitions (e.g. start-job, migrate-job). For example, if all nodes are at full capacity both start-job and migrate-job transitions could not be scheduled until capacity was either added to the cluster or released by one of the nodes.

Example Logging

```
17:10:45.059 [JOB_MANAGER_THREADPOOL-1] INFO redis-connect-manager - Instance:
2468@Allen has 9 remaining capacity to claim job(s) from its 10 max allowable capacity
```

NOTE Each instance can configure its own limit. The limit cannot be changed without restarting the JVM process.

Additionally, there is a control at the job-level which limits the maximum partitions associated to the same job that can be claimed by a single instance. For example, due to a limitation of the Gemfire client library, a single instance can only claim a single job partition with Gemfire as its source even if it has excess capacity at the member-level. To achieve this, maxPartitionsPerClusterMember must be set to 1.

NOTE Using maxPartitionsPerClusterMember has implications on high availability and transitions (e.g. start-job, migrate-job).

Custom Stages

Redis Connect supports user-defined stages which can be implemented in JAVA simply by extending BaseCustomStageHandler and ChangeEventHandlerFactory. Typical use-cases for stages include advanced/proprietary transformations, de-tokenization, callback method invocation, and other forms of enrichment.

Detailed instructions and examples on how to implement custom stages can be found here - https://github.com/redis-field-engineering/redis-connect-custom-stage-sample

Example Custom Stage

```
public class TransformJsonClobStage extends BaseCustomStageHandler {
```

```
    private static final ObjectMapper mapper = new ObjectMapper();
    private String jsonClobColumnName;

    public TransformJsonClobStage(String jobId, String jobType, JobPipelineStageDTO
stage) {
        super(jobId, jobType, jobPipelineStage);
        this.jsonClobColumnName = "JSON_DATA"; // Can be configured through an ENV
variable
    }

    @Override
    public void onEvent(ChangeEventDTO<Map<String, Object>> changeEvent) throws
Exception {

        Map<String, Object> values = changeEvent.getValues();
        if (values != null && values.containsKey(jsonClobColumnName)) {

            JsonNode jsonNode = mapper.readTree((String)
values.get(jsonClobColumnName));

            jsonNode.fieldNames().forEachRemaining(key -> {
                values.put(key, jsonNode.get(key));
            });

            values.remove(jsonClobColumnName);
        }
    }

    @Override
    public void init() {
        setSequenceCallback(new Sequence());
    }

    @Override
    public void shutdown() {}

}
```

Example Custom Stage Factory

```
public class CustomChangeEventHandlerFactory implements ChangeEventHandlerFactory {

    private final String instanceId = ManagementFactory.getRuntimeMXBean().getName();

    private static final String TYPE_TRANSFORM_JSON_CLOB_CUSTOM_STAGE =
"TRANSFORM_JSON_CLOB";

    private static Set<String> supportedChangeEventHandlers = new HashSet<>();
    static {
```

```
        supportedChangeEventHandlers.add(TYPE_TRANSFORM_JSON_CLOB_CUSTOM_STAGE);
    }

    @Override
    public ChangeEventHandler getInstance(String jobId, String jobType,
                                          JobPipelineStageDTO stage) throws Exception
{

        ChangeEventHandler changeEventHandler;

        switch (jobPipelineStage.getStageName()) {
            case TYPE_TRANSFORM_JSON_CLOB_CUSTOM_STAGE: changeEventHandler =
                        new TransformJsonClobStage(jobId, jobType, jobPipelineStage);
                break;
            default: {
                throw new ValidationException("Instance: " + instanceId +  " failed to
load
                            change event handler for JobId: " + jobId + " due to an
invalid
                            Stage: " + jobPipelineStage.getStageName());
            }
        }

        return changeEventHandler;
    }

    @Override
    public String getType() {
        return "CUSTOM";
    }

    @Override
    public boolean contains(String stageName) {
        return supportedChangeEventHandlers.contains(stageName);
    }
}
```

Example Custom Stage Configuration

```
{
  "partitions" : 1,
  "pipeline": {
    "stages" : [
      {
        "index": 1,
        "stageName": "TRANSFORM_JSON_CLOB",
        "userDefinedType": "CUSTOM"
      },
      {
        "index" : 2,
```

```
        "stageName": "REDIS_JSON_SINK",
        "checkpointStageIndicator": true,
        "database": {
          "credentialsDirectoryPath" : "/usr/creds",
          "credentialsRotationEventListenerEnabled" : true,
          "databaseURL": "redis://12.345.67.89:10000",
          "databaseType": "REDIS"
        }
      }
    ]
  },
```

Job Management

Overview

- Job Claimer
- Job Reaper
- Job Manager
- Job Orchestrator
- Metrics Reporter
- REST API & SWAGGER UI

Job Management (control plane) is an orchestration between multiple services, each handling a distinct operational concern, that manage the Redis Connect cluster and provision jobs for execution. Job Management services include the Job Claimer, Job Reaper, Job Manager, Metrics Reporter, Job Orchestrator, and hosting the REST API, SWAGGER UI, & CLI.

While Job Management is entirely data-driven, Redis Connect instances are stateless. This is due to Redis Connect's required dependency on the Job Management Database for its metadata including configurations, claims, heartbeat leases, metrics, transition logs, etc.

Since Redis Connect instances are stateless, they can fail without risking data loss and time-order of events. As long as another cluster member/node has capacity, or the original instance can be recovered, each lost job partition will quickly pick up from its last recorded checkpoint.

NOTE	By default, Job Management is enabled on every Redis Connect instance however can be disabled via job.manager.services.enabled. Disabling Job Execution does not prevent an instance from performing Job Management capabilities including acting as a REST API, SWAGGER UI, & CLI endpoint.

Job Claimer

Each Redis Connect instance, with Job Management enabled, schedules a thread to run the Job Claimer service. Job Claimers are responsible for claiming *UNASSIGNED* job partitions which were published to the Job Claim Assignment Stream. At startup, each Job Claimer subscribes as a

consumer to the Job Assignment Claim Stream's Consumer Group.

Redis Connect instances are in a constant race-condition (see Distributed Consensus for more details) as each Job Claimer periodically attempts to be the first to consume from the Job Claim Assignment Stream in order to claim a job partition. For efficient resource utilization, the job.claim.max.capacity and job.claim.batch.size.per.attempt configurations can be used to prevent over-provisioning until existing claim capacity is relinquished.

Upon successful claim, the job partition's ownership designation is updated within the Job Management Database and a heartbeat notification is initiated. The claim's lease is periodically refreshed to avoid having the Job Reaper service change the ownership designation back to *UNASSIGNED* and subsequently add the job partition back onto the Job Claim Assignment Stream.

By using Redis Streams Consumer Groups, Redis Connect assurances that only a single consumer can consume a message from the stream. Once the message is consumed, it is added to a Pending Entries List (PEL) within the Redis Streams Consumer Group and only removed once ACKnowledged by its consumer. If a failure was to occur before the XACK, there is recovery logic built into the Job Claimer to avoid a deadlock.

NOTE	By default, Job Execution is enabled on every Redis Connect instance however can be disabled via job.claim.service.enabled. Disabling Job Execution does not prevent an instance from performing Job Management capabilities including acting as a REST API, SWAGGER UI, & CLI endpoint.

Job Reaper

Each Redis Connect instance, with Job Management enabled, schedules a thread to run the Job Reaper service based on job.reap.attempt.interval, however, only the cluster leader is allowed to perform its core function. In other words, there will only ever be one Job Reaper actively performing its service across the entire cluster.

The Job Reaper is a watchdog service which identifies job partitions with expired heartbeat leases for the purpose of changing their ownership designation to *UNASSIGNED* and subsequently publishing new claim requests to the Job Claim Assignment Stream. It also takes part in an orchestrated resource cleanup to handle network partition failure scenarios.

Job Manager

Each Redis Connect instance, with Job Management enabled, schedules a thread to run its Job Manager service which facilitates its candidacy for cluster leadership. See Distributed Consensus for more details.

NOTE	To avoid a deadlock scenario with single member clusters, if the cluster leader's heartbeat expires however the Redis Connect instance remains alive, this now former cluster leader will not be blocked from attempting to become reelected.

Job Orchestrator

Each Redis Connect instance, with Job Management enabled, can support all administrative operations via the REST API, SWAGGER UI, or CLI. These operations all flow through the Job Orchestrator which has its own connection to the Job Management Database and equally performs validations regardless of the channel used to initiate the command.

Since Redis Connect has a shared-nothing architecture, administrative operations can be issued from any Redis Connect instance. However, the one caveat is for Log Level operations which only impact the Redis Connect instance that initiated the REST API, SWAGGER UI, or CLI endpoint command.

NOTE	It's typical that Redis Connect instances which are configured to operate the CLI are used as administration-only cluster members with job.claim.service.enabled disabled. This is so they can be spun up and removed without unnecessarily impacting Job Execution.

Metrics Reporter

Each Redis Connect instance, with Job Execution and cluster-metrics enabled, schedules a thread to run the Metrics Reporter service. Its function is to periodically record the collected metrics from across all claimed job partitions with metrics enabled.

There are currently two versions of metrics reporting supported : RedisTimeSeries and OpenTelemetry. {Add more detail once JB is done

REST API & SWAGGER UI

Each Redis Connect instance, with rest.api.enabled enabled, exposes a REST API and a SWAGGER UI for ease of use. By default, the REST API uses port 8282, however, it's configurable via rest.api.port and must be unique if multiple Redis Connect instances are deployed on the same host.

WARNING	When the SWAGGER UI is exposed on an open port in the firewall, it's a best practice not to use the default port.

Example REST API CURL command

```
curl -X GET
"http://localhost:8282/connect/api/v1/job/checkpoint/%7Bconnect%7D%3Ajob%3Acdc-job" -H
"accept: */*"
```

Example REST API Request URL

```
http://localhost:8282/connect/api/v1/job/checkpoint/%7Bconnect%7D%3Ajob%3Acdc-job
```

Example REST API Response Body

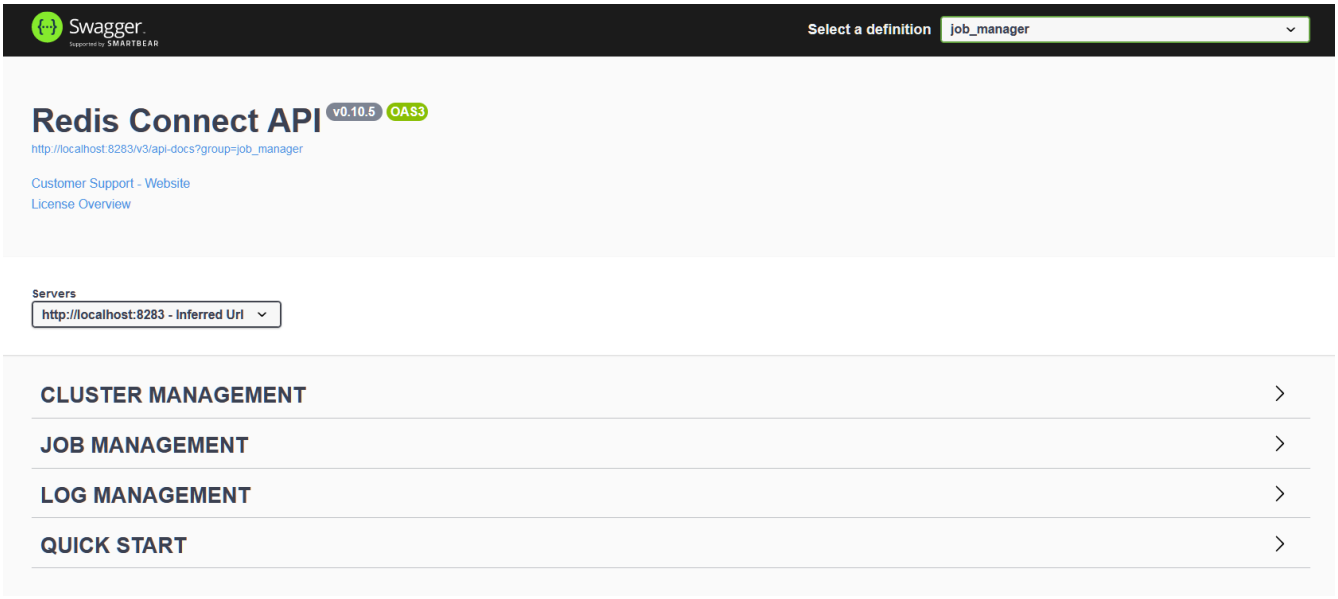
```
{
```

```
  "connect:timestamp" : "1698095512324",
  "src:tx:seq" : "999",
  "connect:name" : "connect:cp:cdc-job:{14698}",
  "src:tx:time" : "1698095450560",
  "lsn" : "23672320",
  "txId" : "500",
  "ts_usec" : "1698095450560554"
}
```

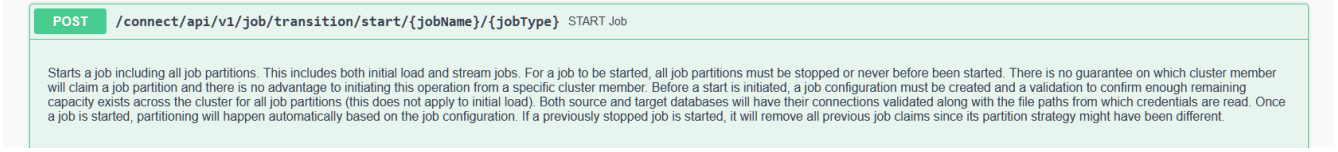
REST API commands are categorized into 3 domains - Cluster Management, Job Management, and Log Management. All Job Management administration can be controlled via REST API.

WARNING	Administrators should **never** directly manipulate metadata within the Job Management Database without using either the CLI or REST API & SWAGGER UI.

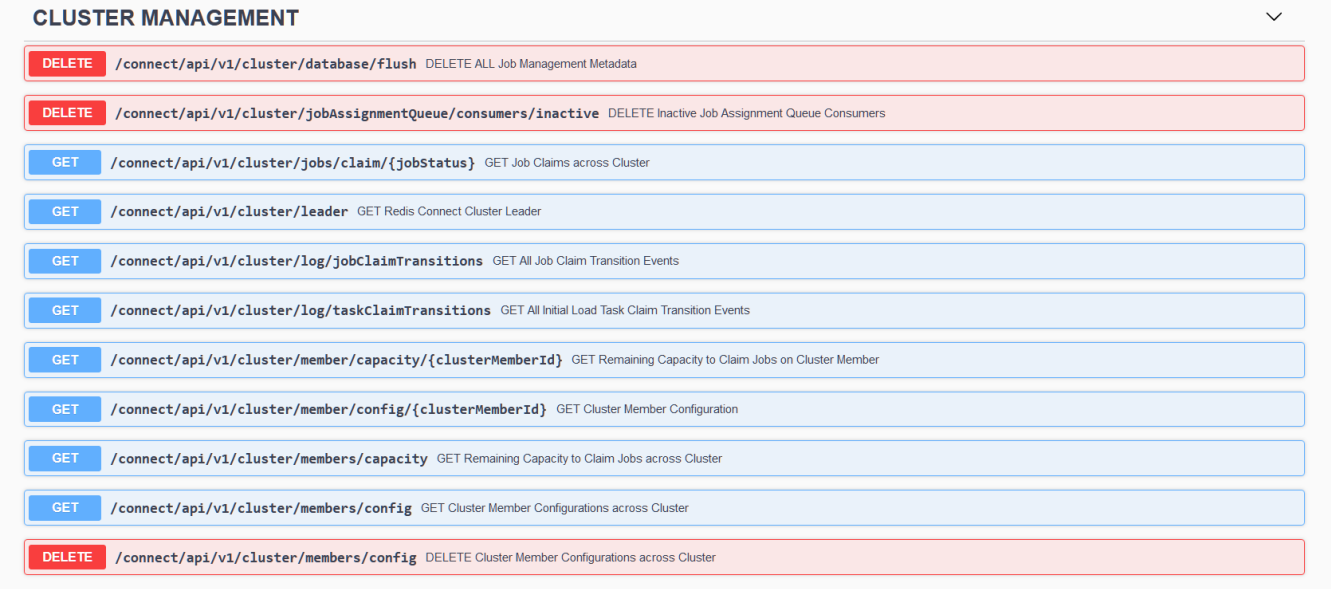
For ease of use, and as a form of API catalogue, Redis Connect supports a SWAGGER UI that can be accessed on the localhost and port of each REST API enabled instance via http://localhost:8283/swagger-ui/index.html#/.

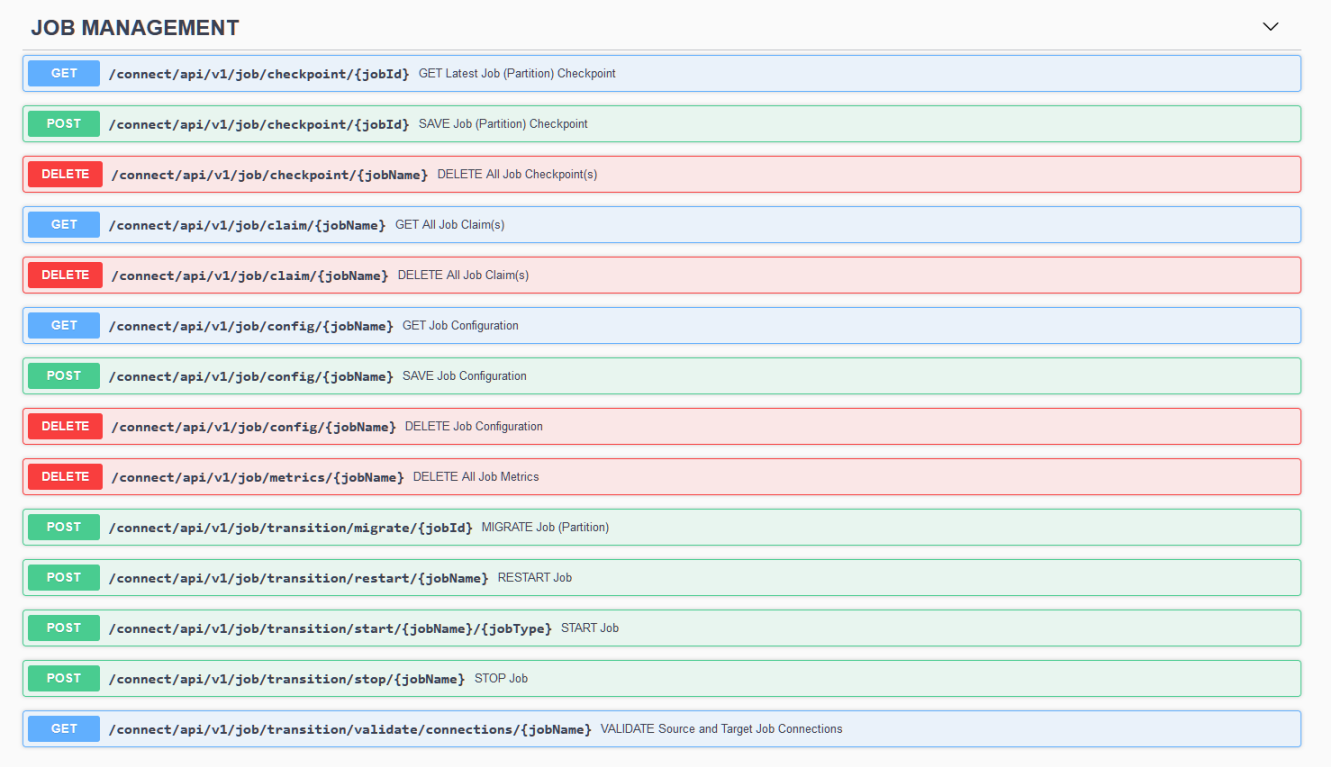


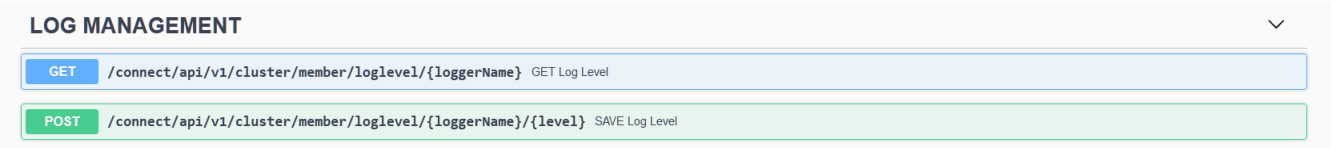
Each API is self-documented and versioned with Redis Connect releases therefore can be considered a single source of truth.



As reference-only, here are the following supported APIs as of Redis Connect v0.10.5:







To access the SWAGGER UI when a Redis Connect cluster is not active OR exposing an open port in the firewall is blocked by security protocols, a reference-only version can be accessed remotely via https://redis-field-engineering.github.io/redis-connect-api-docs/.

NOTE	Even when CURL is used to initiate administrative commands, it is often useful to leverage the SWAGGER UI to construct the Request URL.

CLI

Redis Connect has a shared-nothing architecture therefore any cluster instance/member/node can be configured to expose a command line interface (CLI).

Similar to the REST API & SWAGGER UI, the CLI exposes all administrative commands for the purpose of Job Management. These commands are categorized into 3 domains - Cluster Management, Job Management, and Log Management.

WARNING	Administrators should **never** directly manipulate metadata within the Job Management Database without using either the CLI or REST API & SWAGGER UI.

To begin a CLI prompt, you can simply start a Redis Connect instance with the following parameter:

Example Start CLI

```
$ ./bin/redisconnect.sh cli

redis-connect-cli> |
```

Example List Command Domains

```
redis-connect-cli> help
 -  PicocliCommands registry
Summary: clear   Clears the screen
         cls     Clears the screen
         cluster
         exit
         job
         log
```

Example List Cluster Commands

```
redis-connect-cli> cluster -help
Usage:  cluster [-hV] [COMMAND]
  -h, --help      Show this help message and exit.
  -V, --version   Print version information and exit.
Commands:
  delete_cluster_member_config
  delete_inactive_job_assignment_queue_consumers
  delete_job_management_db_metadata
  get_cluster_job_claims
  get_cluster_leader
  get_cluster_member_config
  get_all_cluster_member_configs
  get_cluster_member_remaining_capacity
  get_cluster_remaining_capacity
  get_job_claim_transitions_log
```

```
get_task_claim_transitions_log
```

For a visual guide on using the CLI, check out this quick start video.

NOTE	Since CLIs are short-running processes, it's a best practice to disable both Job Execution and Job Management on the Redis Connect instance used to deploy it.

Job Management Database

Redis Connect's Job Management is entirely data-driven however Redis Connect instances are stateless. For this reason, Redis Connect has a required dependency on Redis Enterprise as the primary database for its metadata including job configurations, claims, heartbeat leases, metrics, job lifecycle logs, etc.

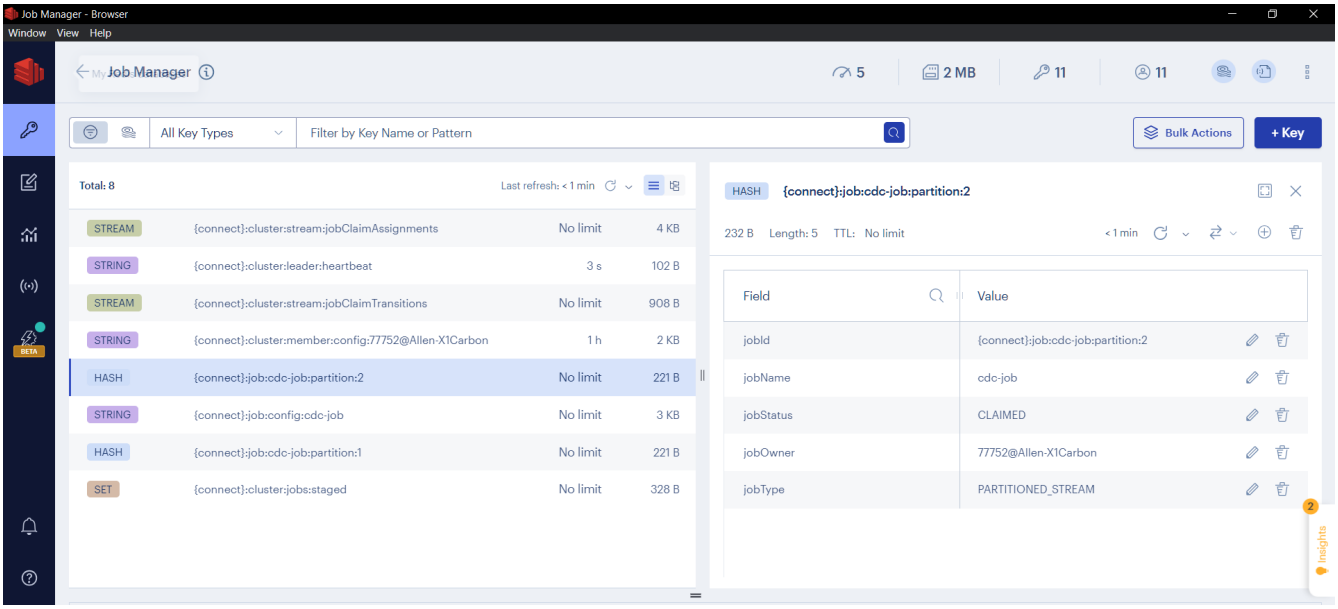
Since metadata availability is a single-point of failure to Redis Connect's operation, it's critical to properly configure Redis Enterprise's advanced operational capabilities including high-availability, persistence, and security. See Production Readiness for more details.

Furthermore, Redis' inherent single-threaded properties and unique commands are foundational to the distributed consensus algorithms which dictate cluster leader election and job claims. Redis Enterprise's data structures/models are even used for the Job Claim Assignment Stream, powered by Redis Streams, and the default Metrics Reporter powered by Redis TimeSeries.

ALL Redis Connect metadata shares a {connect} namespace to ensure that keys are collocated on the same Redis hashslot. This ensures that administrative processes update metadata within Redis transactions to avoid consistency issues. It also provides a simple way to identify Redis Connect metadata when collocated with target Redis database keys in development environments.

WARNING	Production environments (or any with >100k keys) should use a separate database for Redis Connect metadata to avoid painfully long high availability recoveries due the use of the Redis SCAN command. See Production Readiness for more details.

While ALL Redis Connect metadata is available via both CLI and REST API & SWAGGER UI, the easiest way to visualize it is with Redis Insight. Here is an example:



WARNING Administrators should **never** directly manipulate metadata within the Job Management Database without using either the Redis Connect CLI or REST API & SWAGGER UI. See Job Management Database ACLs for more information.

Linear Scalability

Redis Connect jobs can be partitioned for linear scalability. Partitioning allows for increased throughput without impact on end-to-end latency and has minimal overhead on resource utilization. Linear scalability through partitioning is applicable to both LOAD and PARTITIONED_STREAM job types.

NOTE Initial Load can and should be scaled independently of streaming jobs for cost optimization. In fact, it might prove cost-efficient to perform initial load on temporary separate infrastructure to avoid over-provisioing for a one-time task.

Redis Connect supports both vertical and horizontal scalability through its support of multi-tenancy. In the same way, job partitions can be claimed and executed by many Redis Connect instances so too can a single instance claim them all on its own. As long as the cluster adheres to proper capacity estimation, there will be no difference in performance.

NOTE Vertical scalability is recommended to reduce the resource overhead for cluster management and administration. For example, assuming infrastructure allows for it, it would be less overhead to allow a single Redis Connect instance to claim 5 partitions then it would be to maintain a cluster of 5 Redis Connect instances with each claiming a single partition.

Redis Connect routes changed-data events based on a hashing policy applied on the source primary key. This guarantees only a single partition will ever publish a changed-data event to its data pipeline which eliminates duplication and maintains strict order at the key-level.

NOTE While scalability should not be confused with separating tables into different jobs, decisions about which tables to configure within a single job should be considered

with respects to their individual scale requirements.

High Availability

- Redis Connect Instance Failure
- Virtual Machine / Cloud Instance Failure
- Network Partition
- Source Connection Failure
- Target Redis Connection Failure
- Target Redis Slow Consumer
- Recovery Time Considerations
- Graceful Failure

Redis Connect Instance Failure

For high-availability, its best practice to deploy a Redis Connect cluster with *n+1* redundancy. *n* is defined as the minimum number of Redis Connect instances with enough available capacity to execute all staged job partitions. *+1* represent an additional cluster member with enough available capacity to claim the maximum allowable job partitions of a single node.

NOTE There is no limit on how many Redis Connect instances can be deployed for redundancy. However, redundancy comes at a cost, therefore *n*+1 is simply the minimum recommendation with respect to minimizing costs.

In the event of a Redis Connect instance failure, each of its claims will inevitably have an expired heartbeat lease. Once expired, the Job Reaper will identify these job partitions and immediately publish a new request to the Job Claim Assignment Stream. Once published, the first Job Claimer with available capacity to consume the request will mark its claim(s) and automatically restart the pipelines(s) from their last committed checkpoints(s).

NOTE Even if a new Redis Connect instance was quickly restarted on the same virtual machine, for example with supervisord, it would still compete with other cluster members to claim *UNASSIGNED* job partitions.

Virtual Machine / Cloud Instance Failure

For high-availability, it is recommended to deploy each Redis Connect instance on a separate virtual machine or cloud instance. Each virtual machine / cloud instance should have access and privileges to every job's source, target, and the Redis Connect Job Management Database.

For deployment on Kubernetes, see the Redis Connect Kubernetes Docs.

NOTE For on-premises deployments, each virtual machine is deployed on its own server rack to avoid a single point of failure. Similarly, for cloud deployments, cloud

> instances should be chosen across multiple availability zones.

Network Partition

In the event of a network partition, it's theoretically possible that a Redis Connect instance can lose access to the Job Management Database yet still have access to the source and target. This creates the risk of a split-brain scenario, in which case two instances might conclude that they both have a claim to the same job partition. If this happened, delivery guarantees would be nullified.

To avoid a split brain scenario, each job partition queries the Job Management Database, upon **each** iteration of its producer's polling interval to validate that their Redis Connect instance's claim has not expired. Therefore, even if the network-partitioned instance still had access to the source and target, its producer would not be able to poll for additional batches once it loses connection to the Job Management Database nor would it be able to automatically reclaim the job partition when the network partition is remedied.

To avoid false-positives, due to momentary network delays, the Job Claimer cannot claim a job partition unless the Job Reaper identifies it no longer has a heartbeat lease, therefore even if the network-partitioned instance is cutoff from the Job Management Database its lease will remain valid for the duration of its job.claim.heartbeat.lease.renewal.ttl. Only after lease expiration, would the Job Reaper *RELINQUISH* the existing claim from the network-partitioned instance and publish a new request to the Job Claim Assignment Stream.

NOTE	For other permutations of a network partition, such as losing access to either the source or target, there would be no risk of split-brain since the job partition would attempt its normal reconnection logic and eventually stop streaming.

Once events are published to a data pipeline only the Redis Connect instance with a claim to its job partition can stop enqueued events from being committed to the target. For Redis Connect Instance Failure and Virtual Machine / Cloud Instance Failure scenarios, this solves itself since transient queues are also lost, however during a network partition its possible that the Redis Connect instance is still alive. Therefore, to ensure exactly-once delivery, it's important to keep the pipelineBufferSize correlated to the job.claim.heartbeat.lease.renewal.ttl so ample time is afforded for it to quiesce before the claim's heartbeat lease expires.

NOTE	Redis Enterprise has its own split-brain protection so Redis Connect's metadata is protected in the event of a network partition between the primary and backup shards.

Source Connection Failure

Source connections fail for many reasons which are unbeknownst to Redis Connect. Therefore, whether they are intermittent drop-outs or prolonged outages, the producer will always assume that source connection failure is intermittent and attempt to re-establish the connection.

For this purpose, the following configurations exist to control source connection retry logic:

- sourceConnectionMaxRetryAttempts

- sourceConnectionMaxRetryDuration
- sourceConnectionRetryDelayInterval
- sourceConnectionRetryMaxDelayInterval
- sourceConnectionRetryDelayFactor

Example Source Connection Retry Configurations

```
{
  "sourceConnectionMaxRetryAttempts":4,
  "sourceConnectionMaxRetryDuration":10,
  "sourceConnectionRetryDelayInterval":60,
  "sourceConnectionRetryMaxDelayInterval":600,
  "sourceConnectionRetryDelayFactor":2
}
```

In the example above, if the first attempt to reconnect fails, the producer will wait 60 seconds before attempting its first retry. If that retry fails, then the producer will wait 60 * 2 = 120 seconds for its second retry, and so on, until either the sourceConnectionMaxRetryAttempts or sourceConnectionMaxRetryDuration is reached.

Target Redis Connection Failure

Similar to source connection failures, target Redis databases also have a retry logic to handle connection failures. These retries are built into the underlying Redis client (Lettuce).

NOTE	Default Lettuce configurations are typically acceptable with exception to redis.connection.timeout.duration which might need to be adjusted for production environments.

Target Redis Slow Consumer

Sudden spikes in changed-data events on the source can quickly outpace capacity estimations and throughput limits of existing job partitions. To avoid overwhelming their pipelines, Redis Connect has built-in back-pressure support which ensures the instance will survive and protect against noisy neighbors.

NOTE	An indicator of this scenario would be an increase in lag between the source and target which can be tracked via metrics monitoring.

Once the pipelineBufferSize is reached, a circuit breaker will be triggered to disable the producer's polling interval until the requisite buffer capacity becomes available. While the pipeline is iterating through its queued events, the job partition's producer will periodically check if the circuit is closed again, so it can publish the next batch of changed-data events.

In between each attempt, the producer will sleep for intermittentEventSleepDuration, which allows

the pipeline time to quiesce. The producer will retry until either the requisite capacity becomes available or the slowConsumerMaxRetryAttempts limit is reached. Once reached, the job will stop.

Example Back Pressure Logging

```
18:22:45.359 [JOB_PRODUCER_THREADPOOL-1] INFO redis-connect - Instance: 2468@Allen
JobId: {connect}:job:cdc-job:partition:1 skipped polling the source -
SlowConsumerRetryAttempt: 1 out of MaxRetryAttempts: 50 - since its replication
pipeline's RequiredCapacity: 500 only has AvailableCapacity: 125. Therefore the
producer will sleep for IntermittentEventSleepDuration: 3000 before the next attempt.
If MaxRetryAttempts == -1, it could iterate forever.
```

NOTE	In production environments, it's best practice to manually configure the noted thresholds instead of defaulting to unlimited retires. Though back-pressure support will protect an imbalance between source and target throughput, endlessly relying on it will cause an increasing lag which will impact consistency. See Production Readiness for more information on capacity estimation and partitioning.

Recovery Time Considerations

Assuming there is available capacity somewhere across the cluster, recovery time is based on multiple factors including job.reap.attempt.interval, job.claim.attempt.interval, job.claim.heartbeat.lease.renewal.ttl, checkpoint retrieval time from the target, and the duration of starting the connector.

WARNING	It is critical to avoid race-conditions that can cause unexpected behavior between the Job Reaper and Job Claimer services by allowing some buffer for them to do their work. For example, job.reap.attempt.interval must be at least 2 seconds more than the job.claim.attempt.interval.

Graceful Failure

Redis Connect instances are multi-tenant and multi-threaded which comes with the many advantages described in Architecture Overview, however, it is not without risks. One of the major concerns is around resource utilization.

To this end, administrative transition commands (e.g. stop-job, restart-job, and migrate-job) and some unrecoverable errors during Job Execution will cause job partitions to begin a coordinated process to open circuit breakers, quiesce their data pipeline, close out connections, and cancel any scheduled threads.

List of exceptions that bypass pipeline quiesce to avoid deadlocks:

- io.lettuce.core.RedisException
- InvalidChangeEventException
- RedisTransactionFailureException
- RedisWaitCommandTimeoutException

- UnrecoverableTargetConnectionException
- NullPointerException

Production Readiness

Capacity Estimation

Compute Resources

Redis Connect has a cloud-native and shared-nothing architecture which allows any cluster node/member/instance to perform both Job Management and Job Execution. Each Redis Connect instance is multi-threaded and segregates the control plane (Job Management) from the data plane (Job Execution) so administration does not impact performance.

At the minimum, each Redis Connect instance with Job Management enabled, will require 2 vCores to operate its core services, and an additional vCore if Metrics Reporter is enabled.

For maximum throughput, each Redis Connect instance with Job Execution enabled, will require a vCore per stage, including sinks, for each partition that can be claimed up to job.claim.max.capacity.

NOTE	Any Redis Connect instance, with Job Execution enabled, can eventually claim any job partition, therefore, for maximum performance it's recommended to assume that the jobs with the most stages will all end up on the same instance.

Example vCore Estimation per Redis Connect Instance

```
job.claim.max.capacity=2
cluster.timeseries.metrics.enabled=true

Job Management Services = 2 vCores
Metrics = 1 vCore
Job 1 (w/ sink only) = 1 vCore
Job 2 (w/ custom stage + sink) = 2 vCores

Total = 6 vCores per Redis Connect Instance
```

NOTE	Redis Connect instances can be collocated on the same VM / cloud instance, however, each will require fixed vCores for Job Management. Therefore, for efficiency, it's recommended to vertically scale Redis Connect instances by allowing jobs to be executed multi-tenant instead of deploying many JVMs on the same infrastructure.

Memory Resources

Redis Connect is very memory efficient however it's still worth nothing which factors might require additional RAM as instances scale vertically. For example, each job partition maintains its own pipelineBufferSize and an in-memory queue within its connector. As job.claim.max.capacity increase so too will these in-memory buffers.

See Environment Variables on how to configure the JVM for additional RAM

Minimum Resources

Production Environments Minimum Resources

```
4 vCores
4 GB RAM
20 GB of free disk space
1 Gbps networking
```

Development Environments Minimum Resources

```
4 vCores
2 GB RAM
20 GB of free disk space
1 Gbps networking
```

Operating System

Redis Connect can be deployed on physical servers, virtual machines, or using Docker on any K8s-based environment.

Redis Connect is supported on JAVA versions 11+.

For information on deploying to Kubernetes environments, see the Redis Connect Kubernetes documentation.

Environment Variables

Redis Connect requires configuration of several environment variables. Examples of them can be found in the

Example taken from startup script included in the Redis Connect distribution.

```
REDISCONNECT_MIN_JAVA_VERSION="11"
REDISCONNECT_HOME="${REDIS_CONNECT_HOME_DIR}"
REDISCONNECT_JOB_MANAGER_CONFIG_PATH="$REDISCONNECT_HOME/config/jobmanager.properties"
REDISCONNECT_LOGBACK_CONFIG="$REDISCONNECT_HOME/config/logback.xml"
REDISCONNECT_LOGBACK_CLI_CONFIG="$REDISCONNECT_HOME/config/logback-cli.xml"
REDISCONNECT_JAVA_OPTIONS="-XX:+HeapDumpOnOutOfMemoryError -Xms2g -Xmx4g"
REDISCONNECT_EXTLIB_DIR="$REDISCONNECT_HOME/extlib"
REDISCONNECT_LIB_DIR="$REDISCONNECT_HOME/lib/\*:$REDISCONNECT_EXTLIB_DIR/\*"
```

Logging

Redis Connect uses Logback for logging.

Environment variables to set logging configurations

```
REDISCONNECT_LOGBACK_CONFIG="$REDISCONNECT_HOME/config/logback.xml"
REDISCONNECT_LOGBACK_CLI_CONFIG="$REDISCONNECT_HOME/config/logback-cli.xml"
```

Redis Connect has been designed to provide descriptive logs to make troubleshooting easier.

If you need to change your application log level at runtime, you can do this using the REST API. See the loglevel REST endpoint documentation for details.

Note that log level changes are not global; they apply only to the instance whose REST API you are connected to. To change the log level for a given instance, connect directly to that instance's REST API.

If you need to change your application log level at runtime, you can so with the REST API & SWAGGER UI or CLI. See the REST endpoint documentation for details.

NOTE	Log levels are instance specific. To change them globally would require performing the change on each.

Monitoring

Metrics are enabled by default, but can be disabled by setting **cluster.metrics.enabled** to False.

Prometheus metrics are enabled by setting the type of metric (**cluster.metrics.type**) to 'OTEL'. This is the default, the other option being 'REDIS', which will log metrics as Redis' TimeSeries objects in the connect configuration database, thus taking advantage of Redis' time series capabilities, in which case additional data about keys and TTL are available.

The port used by OTEL metrics is set by **prometheus.port**, it is set to 19090 by default.

The OpenTelemetry metrics are reported for Create/Update/Delete operations for every table configured in the payload file. The metrics generated are the overall throughput, as well as the lag

and latency for each operation, which are reported as histograms. They are tagged so that table and operation information can be used to report individual operations or operations on a specific table, or they can be aggregated.

Tags: job, database, table, partition, operation, stage

The sample otel/grafana dashboard looks like this:

Redis Connect exports OpenTelemetry metrics via a Prometheus endpoint. A simple Prometheus configuration would look like this:

```
- job_name: "connect"
  scrape_interval: 5s
  scrape_timeout: 5s
  metrics_path: /
  scheme: http
  static_configs:
  - targets: ["localhost:19090"]
```

Redis Connect provides a dashboard for monitoring the system. After installing Grafana and connecting it to Prometheus, i.e., adding a datasource, you can install the Redis Connect dashboard by navigating to the Grafana dashboard page and clicking New → Import.

The Redis Connect dashboard reports the following metrics:

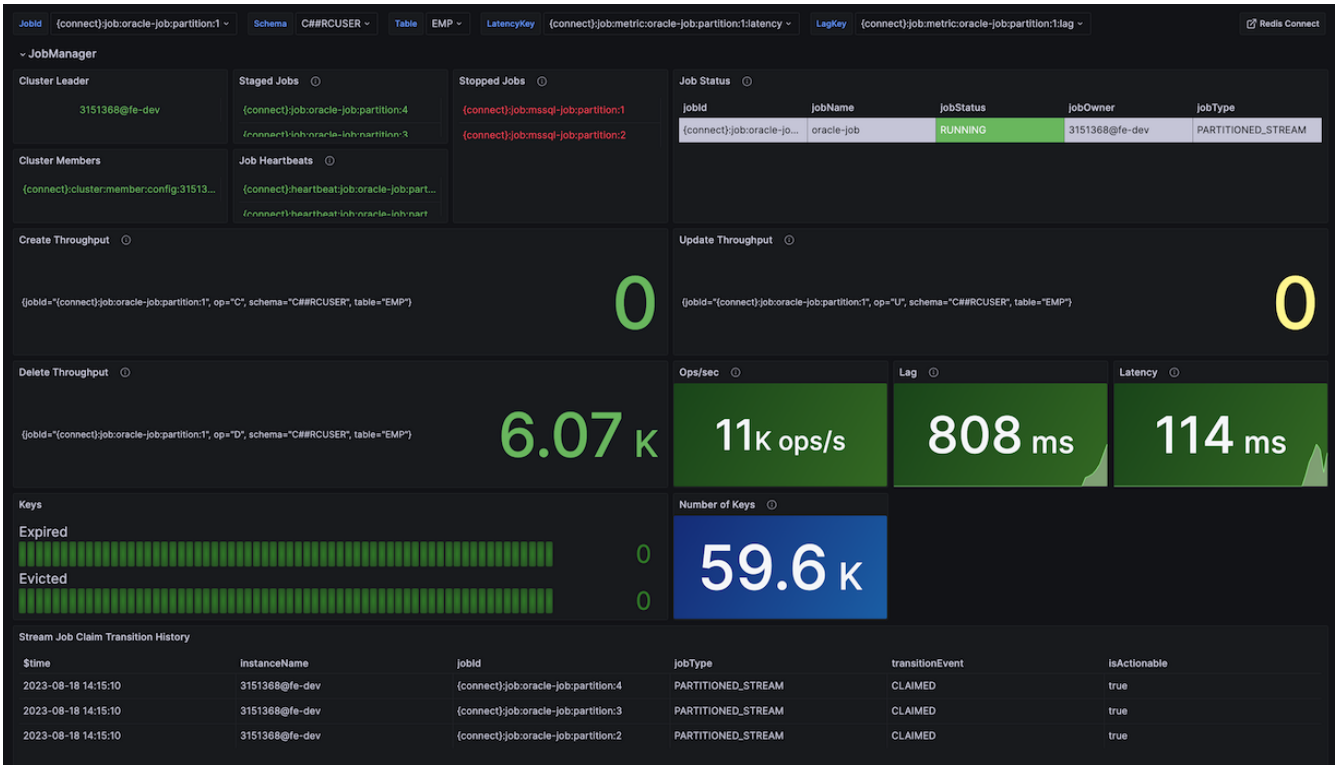
metric	label	type	description
event_job_starts_total	job starts	count	number of times job has been started
event_job_stops_total	job stops	count	number of times job has been stopped

event_input_buffer_histogram	buffer	histogram	number of events received
event_input_buffer_count	buffer count	count	number of measurements
event_input_buffer_sum	buffer total	count	sum of all measured quantities
event_operation_lag	lag	histogram	time it took connect to receive event
event_operation_lag_milliseconds_count	lag count	count	number of measurements
event_operation_lag_milliseconds_sum	lag total	count	sum of all measured quantities
event_operation_latency	latency	histogram	time it took to process the event
event_operation_latency_milliseconds_count	latency count	count	number of measurements
event_operation_latency_milliseconds_sum	latency total	count	sum of all measured quantities
event_operation_elapsed	elapsed	histogram	time it took to write event to redis
event_operation_elapsed_milliseconds_count	elapsed count	count	number of measurements
event_operation_elapsed_milliseconds_sum	elapsed sum	count	sum of all measured quantities
event_job_operation_throughput_total	throughput	count	total number of events processed

With regards to Redis' TimeSeries metrics, you can view these metrics in Grafana using the Redis Datasource for Grafana.

The Redis Connect distribution includes a pre-configured grafana dashboard for viewing key operational metrics in the ../config/samples/dashboard subdirectory. It can be modified to fit the user's monitoring requirements/preferences.

The sample redis/grafana dashboard looks like this:



Metrics include the following:

Staged jobs

Set of all job partitions that have been assigned to the Job Claim Assignment Stream and have not been stopped.

Job Heartbeats

Heartbeats for the all currently active job partitions.

Ops per second

The number of create, update, and delete operations against the <<target?? database completed per second.

Lag

The average elapsed time, in milliseconds, between the commit to the source transactional log and the commit to the target database. High lag values may indicate that your Redis Connect cluster is failing to keep up with the volume of changes at the source. Note that this metric only applies to stream jobs.

Latency

The average elapsed time between, in milliseconds, between the producer publishing the changed-data event to the data pipeline and the commit to the target database.

NOTE	In order to enable lad and latency, sourceTransactionTimeSequenceEnabled must be enabled.

Security

TLS/SSL Support

TLS authentication is often required in production environments whether by one-way or mutual TLS.

Redis Connect manages certificate-based authentication using Java's KeyStore and TrustStore support. Java TrustStore and KeyStore can be configured with truststore.file.path and keystore.file.path, respectifully.

To configure Redis Enterprise with TLS, see the documentation:

- TLS with Redis Cloud
- TLS with Redis Enterprise Software

Job Management Database ACLs

Administrators should **never** directly manipulate metadata within the Job Management Database without using either the Redis Connect CLI or REST API & SWAGGER UI.

To avoid innocent mistakes, ACLs should be enabled to limit access and updatability of the metadata to only Redis Connect instances and privileged administrators.

The following policies are recommended:

1. Create separate Redis database users for your Redis Connect instances, Redis Connect administrators, and developers.
2. Developers should be provided with read-only access to these Redis databases.
3. Ensure that any application user connecting Redis Connect's Redis databases has `dangerous` commands disabled.
4. Users should not have permission to delete keys starting with "{connect}". This prevents accidental deletion of important configuration.

Authentication Credentials

Redis Connect gets its database authentication credentials from properties files stored on the filesystem. The file `redisconnect_credentials_jobmanager.properties` is common to all Redis Connect instances.

This file provides the following authentication credentials:

- Username and password for the Redis database used to store Redis Connect's configuration
- Password for the certificate TrustStore (when applicable)
- Password for the certificate KeyStore (when applicable)
- Password for an SMTP mail server, when email alerts are enabled

In addition to the job manager's credentials, Redis Connect requires two credential files for each job: one for the source database and another for the target Redis database.

Source database credential files are named according to this scheme:

```
redisconnect_credentials_[SOURCE_DB_NAME]_[JOB_NAME].properties
```

For example, if you have a job where PostgreSQL is the source database, and the job is called "user-replication", then you will need a properties file called:

```
redisconnect_credentials_postgresql_user-replication.properties
```

The **target database credentials files** are named as follows:

```
redisconnect_credentials_redis_[JOB_NAME].properties
```

For job called "user-replication", the Redis target database credentials file will be named:

```
redisconnect_credentials_redis_user-replication.properties
```

You can see examples of credentials files in the `config/samples/credentials` directory of the Redis Connect distribution.

Securing Credentials Files

Because credentials files store sensitive information in plain text on the filesystem, these files must have strict permissions. Only authorized users should be able to read and write to these files.

As with private key files, these credentials files should be owned by the redis connect user be set read/write only (e.g., chmod 600).

Credential Rotation

Redis Connect instances listen for changes to the credentials files. When the contents of these files changes, Redis Connect will read the changes to ensure that database connectivity is maintained.

To enable support for credential rotation in Redis Connection, open `jobmanager.properties` and

ensure that the credential rotation directives are uncommented. The following configuration enables the credential file listener and sets it to check for changes every 60 seconds:

- `credentials.rotation.event.listener.enabled=true`
- `credentials.rotation.event.listener.interval=60000`

Secrets Management

You can use a secrets management framework such as Hashicorp Vault to securely store and rotate credentials. To use a secrets management framework:

1. Ensure that framework can write property files in a given mounted path on your filesystem.
2. In `jobmanager.properties`, set the `credentials.dir.path` to this mounted path.

High availability

For high-availability, we recommend that you employ *n*+1 redundancy as part of your Redis Connect cluster. In this case, *n* is defined as the minimum number of Redis Connect instances required to provision all of the jobs that you intend to run.

We also recommend that you deploy each Redis Connect instance on a separate VM. When running in the cloud, you should ensure that your Redis Connect instances are distributed evenly across cloud availability zones.

Instance failure behavior

In the event of a Redis Connect instance (JVM) failure, the heartbeat lease of each job partition owned by that instance expires. Once expired, the job reaper will identify each partition without a heartbeat and immediately publish a job claim request.

Even if a JVM was restarted on the same machine, on which it just failed, it would still need to compete with the other instances with available capacity to claim the job partitions.

It's important to understand each job's capacity requirements and related settings to calculate how many instances are needed. For example, if a job with 4 partitions and a setting of `maxPartitionsPerClusterMember=2` is deployed across a 2-instance cluster, then even if there are 20 available capacity on each Redis Connect instance, they would be blocked from claiming more of this job's unclaimed partitions in the event of a node failure.

It's also important to set the Job Manager property `job.claim.max.capacity` appropriate to the desired redundancy requirement with respect to all job partitions planned for deployment across the cluster. For example, if there are 3 cluster nodes each with a goal to deploy a job with 4 partitions, then in order to achieve n+1 redundancy, each node would have to configure job.claim.max.capacity=2, or more, since the loss of one node would still allow for all 4 partitions to be claimed.

Instance failures

For high-availability, it is recommended to deploy each Redis Connect instance on a separate VM or cloud availability zone. For on-premises deployments, extra care should be taken to ensure that each VM is deployed on different underlying server racks to avoid a single point of failure.

Each VM / cloud availability zone should have access to each job's source, job's target, and the Redis Connect Job Manager database.

For deployment on Kubernetes, see the Redis Connect Kubernetes Docs.

Network partitions

If a network partition occurs, then one side or the other (or both) of the partition needs to stop responding to requests to maintain the consistency guarantee. If both sides continue to respond to reads and writes while they are unable to communicate with each other, they will diverge and no longer be consistent. This state where both sides of the partition remain available is called "split brain".

To avoid "split brain", Redis Connect jobs make a call to the job manager database as the initial stage of their producer's source event-loop to check if their claim is still valid. Here's why that matters. A job cannot be claimed unless the Job Reaper identifies a staged job without a heartbeat, so until that lease expires, the claim is intact. Once the heartbeat expires due to the network partition, the Job Reaper will first remove the claim from the job manager database before publishing a new job claim request. So even if the original Redis Connect instance has not died, it cannot continue to process additional batches once its claim is removed nor can it regain its ownership by default when the network partition is remedied.

By relying on Redis Enterprise as its source of truth, this cluster architecture is guaranteed to always have a single job owner at any one point-in-time. This is because the metadata stored within Redis Enterprise has its own split-brain protection.

Source Connection Failures

Connections fail for many reasons, both intermittent drop-outs and prolonged outages. Since Redis Connect does not know the cause of a connection issue, the cluster will always assume that the connection failure is intermittent and attempt to re-establish the connection.

For this purpose, it's important to set the following configurations within the job source:

- `sourceConnectionMaxRetryAttempts`
- `sourceConnectionMaxRetryDuration`
- `sourceConnectionRetryDelayInterval`
- `sourceConnectionRetryMaxDelayInterval`
- `sourceConnectionRetryDelayFactor`

Here's an example of their use:

- `sourceConnectionMaxRetryAttempts=3`
- `sourceConnectionMaxRetryDuration=10`
- `sourceConnectionRetryDelayInterval=60`
- `sourceConnectionRetryMaxDelayInterval=600`
- `sourceConnectionRetryDelayFactor=3`

Based on the above, if the first attempt to connect fails, then the process waits 60 seconds before attempting its first retry.

If the retry fails, then the process waits 60 * 3 = 180 seconds for its second retry, and so on, up until the max delay interval between retries.

Target Redis Connection Failures

Just like sources, Redis target connections have a retry mechanism. However, these retries are built into the underlying Redis client (a.k.a., Lettuce). For production environments, we recommend using the default Redis connection settings.

Target Redis Slow Consumer

It's possible that the rate of incoming changed-data events and outgoing sink commits becomes unbalanced due to a slowdown on the target database, usually because of network congestion. To avoid overwhelming the job pipeline queue, the combination of the following settings helps to provide back-pressure support:

- slowConsumerMaxRetryAttempts
- intermittentEventSleepDuration

Once the pipeline queue becomes full, a circuit breaker will be triggered to disable the producer's polling event-loop until the requisite pipeline capacity becomes available. While the pipeline is iterating through its queued events, the job's producer will periodically attempt to publish the remainder of its current batch.

In between each attempt, the producer will sleep (based on intermittentEventSleepDuration), which allows the pipeline time to catch up. This loop will continue until either the required capacity becomes available or the `slowConsumerMaxRetryAttempts` limit is reached. Once reached, the job will stop.

In production, we recommend that you set `slowConsumerMaxRetryAttempts` to a reasonable setting instead of making it unlimited or disabling it. This is where proper capacity estimation and partitioning will become critical since you don't want to stop the job unnecessarily due to a network disruption nor large spike in volume. In other words, even if a single partition can handle the entire throughput on a regular basis, it might be worthwhile to partition the deployment so that a sudden spike in throughput or a slow consumer does not bring down the job.

Recovery Time Considerations

Redis Connect's job management processes are scheduled threads that periodically wake up to perform their service. While it's not critical to understand the internals of the cluster architecture, it is relevant to be aware of the balance between job claim heartbeats and the Job Reaper service.

As mentioned previously, when a job partition is claimed it begins a heartbeat lease which has a TTL set by `job.claim.heartbeat.lease.renewal.ttl`. If the lease cannot be renewed before expiry, the Job Reaper service will remove the current owner's claim based on the assumption that its Redis Connect instance is no longer alive.

This service is performed based on a fixed interval set by job.reap.attempt.interval. Once the job claim is removed, each eligible Redis Connect instance will attempt to claim the job partition. This Job Claim service is also based on a fixed interval set by job.claim.attempt.interval.

Since each interval is started at a different point-in-time (and some services have more work to do than others), it's impossible to synchronize all the moving parts. However, it is critical to avoid scenarios that can cause unexpected behavior by giving enough buffer between associated services. For example, `job.reap.attempt.interval` must be at least 2 seconds more than `job.claim.attempt.interval` to avoid sending multiple job claim requests for the same partition. This relates to recovery because each default interval can potentially add up to over a minute in recovery time inclusive of the actual start job process. Therefore, if instant recovery is critical to production SLAs, our recommendation is to consider the balance between separate services and as well as the stability of your network. If these settings are set too low, it's possible that a network disruption can unnecessarily start the recovery process on a healthy job.

See Cluster Configuration for details on setting these properties.

Multi-region deployment

Redis Enterprise lets you replicate databases across regions using active-active and active-passive deployment topologies.

If you deploy your Redis Enterprise databases across regions, you also need to consider how you will deploy Redis Connect and how you will recover if you lose a region. Let's consider a real-world scenario.

Suppose your data infrastructure is divided between two regions: West and East.

- Your primary data center resides in the West.
- You source database for CDC is Postgres, and this database is hosted in the West.
- Your target database is a Redis Enterprise active-active or active-passive database with replicas in both West and East data centers. Because the target database stores checkpoints for CDC, Redis Connect will be able to recover from where it stopped in the event of a failover.
- Your Redis Connect job manager database is a Redis Enterprise database deployed in the West region. You have also provisioned a job manager database in the East region. This East database will not contain any configuration until you fail over.

In a scenario like this one, we recommend running a Redis Connect cluster in the West and maintaining a cold standby cluster in the East. Running Redis Connect entirely in one region ensure a low-latency CDC pipeline.

As usual, we also recommend deploying at least two Redis Connect instances in each region, which will provide high availability within that region.

Disaster recovery

Redis Connect provides high availability within a single region; however, failover to a secondary region requires some planning and manual intervention. Continuing with the example above, you need a plan to fail over to the East data center and start a cold standby Redis Connect cluster.

First, ensure that you can start an equivalent Redis Connect cluster in the East. The means deploying the same number of Redis Connect instances you have in the West. Let's call this the "cold standby cluster".

The cold standby cluster instances in the East should maintain a local configuration nearly identical to the instances in the West, with a few important caveats:

1. Instances in the East will need to connect to a job manager database in the East. This means that you will need to pre-provision a job manager database in your Redis Enterprise East cluster.
2. Instances in the East will use the East target database for CDC. This means that all job configurations running in the East should point to the Redis Enterprise as the target database in the East.
3. You need to determine how Redis Connect jobs will connect their source, in this case Postgres. Will you have a secondary Postgres server in the East that your instances can connect to? If so, your CDC jobs will need to be configured to point to this database. In addition, this database must have equivalent transaction logs to the database in the West so that Redis Connect can continue reading from the same offset checkpoints.

Here's a summary of the configuration differences between the live Redis Connect cluster in the West and the cold standby in the East:

Table 2. Cluster properties

Region	Instances deployed	jobmanager.properties	CDC source	CDC target
West	>= 2	`redis.connection.url` points to **West** jobmanager database	Job configuration JSON points to CDC source in **West**	Job configuration JSON points to **West** target database
East	>= 2 (and equal to West)	`redis.connection.url` points to **East** jobmanager database	Job configuration JSON points to CDC source in **East**	Job configuration JSON points to **East** target database

Failover procedure

If your West region becomes unavailable, and you decide to manually fail over, you can fail over to

East as follows:

1. Stop all Redis Connect instances in the West (if they're still running).
2. Ensure that source, target, and job configuration databases are accessible in the East.
3. Verify that transaction logs for the source are still available in the East.
4. Start your cold standby Redis Connect instances in the East.
5. Start your CDC jobs by submitting your CDC JSON configuration files using Redis Connect's REST API.

Configuration Overview

- Cluster Instance Configuration
- Job Configuration

Cluster Instance Configuration

The most basic Redis Connect instance configuration includes the URL of the Redis database used to store the cluster's state. This URL is specified with the property `redis.connection.url` in `jobmanager.properties`.

Here's a sample `jobmanager.properties` file:

```
#####     Cluster properties                                    ①
cluster.name=default

#####     Job Manager Database properties                       ②
redis.connection.url=redis://127.0.0.1:14001
redis.connection.auto.reconnect=true

#####     Credentials properties                                ③
credentials.dir.path=/usr/local/redis-connect/credentials
credentials.rotation.event.listener.enabled=true

#####     REST properties                                       ④
rest.api.enabled=true
rest.api.port=8282

#####     Job Manager Services properties                       ⑤
job.manager.services.enabled=true
job.manager.services.threadpool.size=2
```

① The **cluster properties** define general Redis Connect cluster configuration.

② The **job manager database properties** determine which database Redis Connect uses to store its configuration.

③ The **credentials properties** tell Redis Connect where to find credentials files.

④ The **REST properties** determine whether the REST API is enabled and how it runs.

⑤ The **job manager services properties** lets you specify the resources to allocate for job managements on this JVM.

Job Configuration

Redis Connect runs one or more **change-data capture jobs**. Each job reads data from a source database and ultimately writes that data to a Redis target database. A job is defined by a JSON document. To create a job, you submit this JSON document using the REST API or CLI. Once a job is

submitted, the Redis Connect cluster leader will schedule the job among available instances.

The following section shows how to construct a JSON-based configuration for a job. To see several real-world Job configurations, open the `config/samples/payloads` folder in the Redis Connect distribution.

Job configuration format

Every job configuration has at least three top-level keys: partitions, source, and pipeline.

```
{
  "partitions" : NUMBER_OF_PARTITIONS,    ①
  "source": { SOURCE_JSON_DEFINITION },   ②
  "pipeline": { TARGET_JSON_DEFINITION }  ③
}
```

① The **partitions** field defines the level of parallelism to use when running the job.

② The **Job Source Properties** field stores configuration for the job's source database (e.g., MySQL, Oracle, etc.)

③ The **Job Pipeline Properties** field stores configuration for the target of the job. This includes custom transformations and the target Redis database.

Here's a more detailed job configuration with some of these valued filled in:

```
{
  "partitions" : 1,
  "source": {
    "database": {                     ①
      "databaseType": "POSTGRES",
      "databaseURL": "jdbc:postgresql://127.0.0.1:5432/RedisConnect",
      "customConfiguration": {        ⑤
        "database.dbname" : "RedisConnect",
      }
    },
    "tables": {                       ②
      "public.emp": {
        "columns": [
          { "targetColumn": "empno", "sourceColumn": "empno", "targetKey": true},
          { "targetColumn": "hiredate", "sourceColumn": "hiredate", "type":
"DATE_TIME"}
        ],
        "initialLoad": {
          "partitions": 4
        },
        "autoConfigColumnsEnabled": true
      }
    }
  },
```

```
  "pipeline": {
    "stages": [                          ③
      {
        "index": 1,
        "stageName": "REDIS_HASH_SINK",
        "database": {                    ④
          "credentialsDirectoryPath" : "../config/samples/credentials",
          "databaseURL": "redis://127.0.0.1:14000",
          "databaseType": "REDIS",
          "customConfiguration": {       ⑤
            "redis.connection.sslEnabled": false,
            "truststore.file.path": "../config/samples/credentials/client-
truststore.jks"
          }
        }
      }
    ]
  }
}
```

① A **database configuration** for the database that Redis Connect reads from.

② A **tables configuration** specifying which tables to read from.

③ The list of **stages**, which always ends with the Redis target database.

④ The **database configuration** for the target database. The final stage in the pipeline is always the Redis target database.

Note that all **database configuration** objects include a **customConfiguration** field. <5> The **customConfiguration** field lets you pass in database-specific configurations. For example, Postgres has a `database.dbName` field that is unique to Postgres.

Submitting a job configuration

To save a job configuration, construct your JSON payload and then submit it using the REST API's create job endpoint. When you submit this configuration, you will also provide a unique job name. This job name will be used in the naming of your credentials files.

Credentials files

Credentials files store authentication credentials for the databases used by Redis Connect.

The location of these credentials files is specified at the `credentials.dir.path` properties in `jobmanager.properties`. To learn about which credentials are required, see the authentication credentials subsection.

Configurations

Cluster Configurations

Table 3. Cluster properties

Property name	Type	Description	Default
cluster.name	String	Metadata purposes only. Non-functional	default
cluster.leader.heartbeat.lease.renewal.ttl	Integer	TTL (Time-to-Live) which is renewed upon each `cluster.election.attempt.interval` iteration by the cluster leader. Measured in milliseconds with a minimum of 1 second (1000 ms).	5000
cluster.election.attempt.interval	Integer	Fixed rate scheduled thread which either renews or elects a new cluster leader. Runs on each Redis Connect Instance (JVM) when `job.manager.services.enabled=true`. Measured in milliseconds with a minimum of 1 second (1000 ms)	5000
cluster.timeseries.metrics.enabled	Boolean	Enables creation of a scheduled thread for job metrics reporting to RedisTimeSeries within the job manager database.	false

Job manager services properties

Table 4. Job manager services properties

Property name	Type	Description	Default
job.manager.services.enabled	Boolean	Enables creation of scheduled thread(s) to participate in cluster leader elections, facilitate REST API / CLI (Job Manager service), and identify staged jobs without a heartbeat lease (Job Reaper service). When this property is disabled, the Redis Connect instance may still participate in job execution and job claim attempts (Job Claimer service).	true

Property name	Type	Description	Default
job.manager.services.threadpool.size	Integer	For non-production deployments, one thread is adequate. In production, we recommend two threads.	2
job.reap.attempt.interval	Integer	The interval between attempts to identify staged jobs without a heartbeat lease. Implemented as a scheduled thread that runs on each Redis Connect Instance (JVM) when `job.manager.services.enabled=true`. Measured in milliseconds with a minimum of 1 second (1000 ms)	7000
job.claim.service.enabled		Enables creation of scheduled thread(s) to attempt to claim ownership for UNASSIGNED staged jobs (Job Claimer Service), job execution, and job-level metrics reporting (Metrics Reporter service). When this property is disabled, the Redis Connect instance may still participate in cluster leader election, facilitate REST API / CLI, and perform Job Reaper services.	true
job.claim.attempt.interval	Integer	Interval at which this scheduled thread attempts to claim ownership for UNASSIGNED staged jobs. Runs on each Redis Connect Instance (JVM) when `job.claim.service.enabled=true`. Measured in milliseconds with a minimum of 1 second (1000 ms)	5000
job.claim.batch.size.per.attempt	Long	Specifies how many jobs can be claimed per attempt interval. If a sparse topology across many Redis Connect instances is desired, then lowering this interval is recommended.	4
job.claim.max.capacity	Integer	Specifies the maximum number of jobs that a single Redis Connect instance can claim at any given time.	4
job.claim.heartbeat.lease.renewal.ttl	Integer	TTL (Time-to-Live) which is renewed upon each iteration of a fixed rate scheduled thread that shares its value. Measured in milliseconds with a minimum of 1 second (1000 ms)	10000

REST API Properties

Table 5. REST API properties

Property name	Type	Description	Default
rest.api.enabled	Boolean	Instantiates an embedded Spring Boot Application to host the REST API and CLI.	true
rest.api.port	Integer	Specifies the port used for the REST API (and SWAGGER) powered by an embedded Spring Boot Application. If you are running multiple Redis Connect instances on the same server, each instance will require a different port for its REST API.	8282

Job Manager Database Properties

Table 6. Job Manager Database Properties

Property name	Type	Description	Default
redis.connection.url	String	A Redis URI indicating which Redis server to use for job management. For the Redis URI spec, the Lettuce documentation.	n/a
redis.connection.insecure	Boolean	Passed to Lettuce's `RedisURI.verifyPeer`. If true then verifyMode=FULL. Otherwise, if false, then verifyMode=NONE. When peer verification is disabled, Lettuce uses Netty's InsecureTrustManagerFactory.INSTANCE as the trust manager factory. Its javadoc notes that it should never be used in production and that it is purely for testing purposes.	false
redis.connection.timeout.duration	Integer	The timeout is canceled upon command completion/cancellation. Measured in seconds.	1
redis.connection.auto.reconnect	Boolean	Determine whether the driver will attempt to automatically reconnect to Redis. When enabled, then on disconnect, the client will try to reconnect, activate the connection and re-issue any queued commands.	true

Property name	Type	Description	Default
redis.connection.suspend.reconnect. on.protocol.failure	Boolean	When set to `true`, reconnect will be suspended on protocol errors. The reconnect itself has two phases: Socket connection and protocol/connection activation. In case a connection timeout occurs, a connection reset, or host lookup fails, this does not affect the cancellation of commands. In contrast, where the protocol/connection activation fails due to SSL errors or PING before activating connection failure, queued commands are canceled.	true
redis.connection.sslEnabled	Boolean	Enables SSL for one-way or mutual authentication. If this flag is set to `false`, TrustStore and KeyStore will not be passed to the client.	false

Job Manager Credentials Properties

Table 7. Job Manager Credentials Properties

Property name	Type	Description	Default
truststore.file.path	String	File path of the Java TrustStore (containing certificates trusted by the client)	n/a
keystore.file.path	String	File path of the Java KeyStore, which stores private key entries, certificates with public keys, or any other secret keys used for various cryptographic purposes.	n/a
credentials.dir.path	String	The name of the directory containing the Redis Connect credentials file. This directory path must include a properties file named `redisconnect_credentials_jobmanager.properties` [1].	../config/samples/credentials
credentials.rotation.event.listener.enabled	Boolean	When set to `true`, a listener will be created on the `redisconnect_credentials_jobmanager.properties` file within the `credentials.dir.path` to rotate credentials when they change. This lets you rotate credentials without restarting your Redis Connect instance.	false

Property name	Type	Description	Default
credentials.rotation.event.listener.interval	Integer	When `credentials.rotation.event.listener.enabled` is set to `true`, this flag sets the frequency at which is scanned for changes. Measured in milliseconds with a minimum of 60 seconds (60000 ms)	60000

Email Alerting Properties

Table 8. Email Alerting Properties

Property name	Type	Description	Default
mail.alert.enabled	Boolean	Enables email alerts when any error forces a job to stop.	false
mail.smtp.host	String	Hostname of the outgoing mail server.	smtp.gmail.com
mail.smtp.port	Integer	Set the non-SSL port number of the outgoing mail server.	587
mail.smtp.start.tls.enable	Boolean	Set or disable STARTTLS encryption[2].	true
mail.smtp.start.tls.required	Boolean	Set or disable the required STARTTLS encryption.	false
mail.to	String	The email address to send alerts to. This email address will also be used as the personal name. Multiple recipients can be added by delimiting them with a comma.	n/a
mail.debug	Boolean	Set session debugging on or off.	false

[1] Redis Connect never caches or persists credentials. Therefore, on each connection with the source, target, or job manager database, the credentials are read from a file. This enhances security and allows for seamless credential rotations and integration with secret management frameworks such as HashiCorp Vault.

[2] StartTLS is an extension of the SMTP protocol that tells the email server that the email client wants to use a secure connection using TLS or SSL.

Job Configurations

This section describes the fields in job configuration payload.

Job properties

Table 9. Job properties

Property name	Type	Description	Default
jobName	String	Unique name for a job. This name is used to derive all other Redis metadata keys related to the job execution workflow. Note: the jobName property is submitted separately from the job configuration when saving a job. In other words, jobName does not appear in the job configuration payload. Note: jobName should not be confused with jobId. jobIds are created as part of a job claim. They add-on a namespace to the jobName to identify the jobType and partitionId (if jobType=PARTITIONED_STREAM) [1].	n/a
partitions	Integer	Indicates how many partitions to create during startJob process. This attribute is ONLY used to partition a job with jobType=PARTITIONED_STREAM. Not jobType=LOAD. `CAUTION: Once a job has started, and job claims are created, a job cannot be repartitioned without deleting all job claims and existing checkpoints. Please reach out to Support to assist with the migration of checkpoints to avoid undesired outcomes.`	1
maxPartitionsPerClusterMember	Integer	The number of job partitions that can be claimed, and executed, on the same Redis Connect instance (JVM). If the limit forces partitions to span more instances than are currently deployed, then the job will not be able to start nor migrate [2].	0

Property name	Type	Description	Default
pipeline	JSON Object	See Job Pipeline Properties	n/a
source	JSON Object	See Job Source Properties	Not Null

Job Pipeline Properties

Table 10. Job Pipeline Properties

Property name	Type	Description	Default
pipelineBufferSize	Integer	Redis Connect's pipeline is powered by the LMAX Disruptor library (High Performance Inter-Thread Messaging). Must be a power of 2, minimum 1024 [3]	4096
preprocessorName	String	Functional interface (Consumer) that can be run before changed-data events are transformed and published to the pipeline. This is currently not extendable by end users.	n/a
postprocessorName	String	Functional interface (Consumer) that can be run after changed-data events are transformed and published to the pipeline. This is currently not extendable by end users.	n/a
stages	Job Pipeline Stage[]	See Job Pipeline Stage Properties	

Job Pipeline Stage Properties

Table 11. Job Pipeline Stage Properties

Property name	Type	Description	Default
stageName	String	Unique name which is used as an exact match reference to a custom-built target sink or a user-defined custom stage.	n/a
index	Integer	Specifies the sequence in which the stages of the pipeline should be orchestrated. Begins with 1 and each subsequent index should increment by 1	n/a

Property name	Type	Description	Default
metricsEnabled	Boolean	When enabled, the target sink stage will report throughput and latency related metrics for persistence in RedisTimeSeries. This can subsequently be visualized in Grafana.	false
metricsRetentionInHours	Long	Maximum duration for metrics samples as compared to the highest reported timestamp before they expire. Measured in hours; minimum is 1.	4
checkpointStageIndicator	Boolean	Indicates which sink will be responsible for committing the checkpoint to the target database. This is typically performed by the last stage of the pipeline and, often times, it is the only stage in the pipeline. Job pipeline can only have a single stage with checkpointStage `Indicator=true`	false checkpointTransactionsEnabled
checkpointTransactionsEnabled	Boolean	Although the producer's polling event loop enqueues changed-data events in batches, each event is processed individually through the pipeline. This is because Redis Connect updates the checkpoint at the changed-data event level and not the batch [4].	false
keyPrefix	String	Adds a prefix to the target Redis key before the tableName and composition of targetKey enabled columns.	
userDefinedType	String	To create a custom stage, a factory interface must be extended so that Redis Connect can have visibility to it from a class loading perspective. See section X.X.X. The interface will force the user to create a `getType()` method which returns a unique String to represent the custom factory. This property must exactly match that custom unique String so that Redis Connect can properly discover and handle it as a custom stage.	
database		See Database Configuration Configuration for all target database configuration.	

Property name	Type	Description	Default
checkpointDatabase		See Database Properties Checkpoint database configuration. This is required only if Redis is not the target destination (which is only supported when Splunk is the target).	

Job Source Properties

Table 12. Job Source Properties

Property name	Type	Description	Default
pollSourceInterval	Long	Fixed rate interval representing how long to pause the producer's polling event loop if no new change events were found in the batch. Measured in milliseconds; minimum is 5	50
batchSize	Integer	Maximum # of events to dequeue from the source-event-queue AND maximum # of events to query from the source transaction log/table/queue upon each interval of the producer's polling event loop. Minimum is 1.	500
sourceTransactionTimeSequenceEnabled	Boolean	When enabled, the source commit/transaction timestamp (and sequence# if the timestamp is the same) will be used to calculate latency metrics and passed along as metadata for Redis Streams sink(s).	false
slowConsumerMaxRetryAttempts	Integer	-1 = UNLIMITED 0 = DISABLED 1+ = MAX_ATTEMPTS Used as part of back-pressure support for the data pipeline in the event of a slow consumer. If the maximum attempts limit is reached, the job will be stopped for purposes of manual intervention.	50
intermittentEventSleepDuration	Integer	Used as part of back-pressure support for the data pipeline in the event of a slow consumer or the circuit breaker is open. Forces the event loop to pause for the configured duration of time. Measured in milliseconds, minimum is 0.	3

Property name	Type	Description	Default
sourceConnectionMaxRetryAttempts	Integer	0 = DISABLED 1+ = MAX_ATTEMPTS Maximum retry attempts to reconnect with the source in the event that a connection is lost. Minimum is 0.	3
sourceConnectionMaxRetryDuration	Integer	In addition to sourceConnectionMaxRetryAttempts, you can also add a max duration, after which retries will stop if the max attempts haven't already been reached. Measured in minutes; minimum is 1.	5
sourceConnectionRetryDelayInterval	Long	Fixed delay in between sourceConnectionMaxRetryAttempts. Measured in seconds; minimum is 0[5].	60
sourceConnectionRetryMaxDelayInterval	Long	Provides an upper bound to calculate the delay interval when sourceConnectionRetryDelayFactor is enabled. Measured in seconds, minimum is 0.	240
sourceConnectionRetryDelayFactor	Integer	0 = DISABLED 1+ = DELAY_FACTOR Factor by which delays are exponentially increased after each source connection retry attempt. Minimum is 0.	2
database	Object	See Database Configuration Configuration for all source databases. Not Null.	n/a
tables	Map<String, Table>	See Job Source Table Column Properties Configuration for all source tables/collections/regions/logs properties. Each table within the map requires a unique name which will be used as part of target key composition. Not Null.	n/a

Job Source Database Properties

See Database Configuration.

Job Source Table Properties

Table 13. Job Source Table Properties

Property name	Type	Description	Default
autoConfigColumnsEnabled	Boolean	When enabled, source metadata is queried during the (re)start process to determine sourceColumn names so users do not need to enumerate each within the column's configuration. The columns configuration can be used to override source metadata (i.e., targetName, type, etc.). However, targetKey designation cannot be overridden since only the source table's primary key will be used. This is a common configuration in POCs and development environments since the design of Redis key names are less important than in production. It also allows for less knowledge about the source table schema. This is only supported for RDBMS sources.	false
dynamicSchemaEnabled	Boolean	When enabled, columns that are not provided in the columns configuration will be passed through, as-is, to the target. This is currently only supported for MongoDB, Redis Streams Broker, and Files.	false
prefixTableNameToTargetKeyEnabled	Boolean	When enabled, adds the tableName (defined in the tables configuration) as a prefix to the target Redis key before all other targetKey enabled columns are computed and applied.	true

Property name	Type	Description	Default
deleteOnPrimaryKeyUpdateEnabled	Boolean	When enabled, if the primary key is changed at the source, then an additional operation to DELETE the existing target key will accompany the UPDATE event. This is only supported for RDBMS sources since primary key changes require a delete and insert of a new row. The DELETE event shares an offset with the UPDATE event both at the source and checkpoint. Redis Connect will handle them within a single pipeline iteration.	true
changedColumnsOnlyEnabled	Boolean	When enabled, only allows changed (delta) column values to be replicated to the target. This does not include targetKey column(s) which cannot be bypassed. When disabled, all column values will be replicated to the target unless they are individually bypassed at the column-level using changedColumnOnlyEnabled. (See Section 4.2.2) [6].	false
columns	Job Source Table Column[]	See Job Source Table Column Properties.	Null
initialLoad	Initial Load	See Job Source Table Initial Load Properties.	n/a

Job Source Table Column Properties

Table 14. Job Source Table Column Properties

Property name	Type	Description	Default
targetKey	Boolean	Designates this column's value as part of the target's key composition process. When more than one column is designated, the order in which they are listed will impact the order in which they are appended to the key.	false
sourceColumn	String	Exact match identifier for source column name. Must not be empty.	n/a
targetColumn	String	Preferred field name to be used in the target. Must not be empty.	n/a

Property name	Type	Description	Default
type	String	Identifies the source column's data type which is used to transform the column value to a properly formatted String within the target. Supported types include: [STRING, VARCHAR, TEXT, INT, DATE, DATE_TIME, BYTE, DEC, NUMERIC, DECIMAL, DOUBLE, FLOAT, LONG, SHORT, RAW, BLOB, CLOB, HASHMAP, CUSTOM] CUSTOM data type is unique in that it bypasses column value transformation to a String which allows it to be converted manually within a Custom Stage. An example would be converting to a proprietary Oracle Timestamp format. Failure to convert this data type manually will cause errors in Redis-based sinks.	STRING
changedColumnOnlyEnabled	Boolean	When enabled, only allows changed (delta) column values to be replicated to the target unless targetKey is enabled. When changedColumnsOnlyEnabled=true at the table-level, this flag will be overridden. This is currently only supported for RDBMS sources.	false
passThroughEnabled	Boolean	When disabled, the source column value will not be published to the pipeline therefore it cannot be accessed within a custom stage nor any sink. The purpose of this flag is to allow source column values to be used for targetKey composition without adding the column's name/value pair as a field within the target. As an example, this is common for sources like MongoDB which generate a "_id" key which can be used as a targetKey but has no value as a field.	true
index	Integer	This is currently for metadata purposes only and has no functional value.	n/a

Property name	Type	Description	Default
dateFormat	String	Used by DATE and DATE_TIME type to override their default. Default formats are as follows: DATE = YYYY-MM-dd DATE_TIME = YYYY-MM-dd HH:mm:ss.S	n/a
nullFormat	String	Users can define how a column value=NULL will be represented in the target.	Default is an EMPTY String.

Job Source Table Initial Load Properties

Table 15. Job Source Table Column Properties

Property name	Type	Description	Default
partitions	Integer	Indicates the number of partitions to create during startJob process. This attribute is ONLY used to partition an initial load with jobType=LOAD. Each table should be partitioned based on its own size and release window SLAs. It's common practice to leverage more partitions for an initial load than when streaming. Please see the Production Readiness section for more detail. Disclaimer: If the source table has fewer than 500 rows, which is common in a dev environment, all but partition:1 will be stopped so all the rows are loaded from a single partition. Minimum is 1.	1

Property name	Type	Description	Default
maxPartitionsPerClusterMember	Integer	Limits how many task partitions can be claimed, and executed, multi-tenant on the same Redis Connect instance (JVM). If the limit forces partitions to span more nodes than are currently deployed, then the initial load will queue the instantiation of tasks until capacity is reallocated (e.g. earlier tasks complete their load partition). This is not a job-level limit; it is only specific at the table level. 0 represents no limit. Minimum is 0.	0
customWhereClause	String	Users can specify a WHERE clause to filter the rows required for initial load. Only the following sources are supported: - RDBMS sources support JDBC compliant WHERE statements - MongoDB supports a BSON filter - Gemfire supports an Apache Geode WHERE Clause	
rowIndexUsedAsTargetKeyEnabled	Boolean	RDBMS sources can have tables without primary keys. For those cases, rowIndex can be used as a unique identifier for partitioning purposes. This is only supported for RDBMS sources and only for initial load only / ETL jobs.	false

[1] When jobName is used in logging or administrative processes (i.e., stopJob), the jobName represents ALL job partitions. Must be between 4 and 50 characters.

[2] For example, if maxPartitionsPerClusterMember=1 and partitions=3, then the Redis Connect cluster will require at least 3 instances (JVMs) each with at least 1 available capacity to claim a job partition. This is not a global limit; it is only specific at the job level 0 represents no limit

[3] The buffer size sets the number of slots allocated within the Disruptor's internal ring buffer "queue". Increasing the buffer size will impact the JVM heap space required to store all transient changed data events within the queue. For most cases, this can be left as default.

[4] When enabled, the checkpoint will be committed as part of an atomic Redis transaction. This eliminates consistency issues and improves performance. Rollback capability is built in to handle any failure scenarios during the transaction so that no data will be lost. When disabled, the checkpoint will be committed after the the changed-data events are written. This adds another network round trip for each changed-data event. While distributed checkpoints have distinct advantages, there is a small tradeoff. Because Redis keys are bound by their hash-slots, distributed checkpoints require that we store one checkpoint per hash slot. When enabled, each job partition will have its own 16384 checkpoint keys created in the target database. With ~250 bytes per checkpoint, each partition's estimated overhead is ~4MB. When disabled, there is only a single checkpoint key in the target per partition. Distributed checkpoints require RediSearch. We use RediSearch to index checkpoint keys so that recovery from the latest checkpoint is immediate

[5] sourceConnectionRetryDelayInterval must be < than sourceConnectionRetryDelayInterval

[6] When enabled, the column-level changedColumnOnlyEnabled flag will be overridden for all columns other than those designated as targetKey(s). This is currently only supported for RDBMS sources

Database Configuration

These properties are available for all **database** configuration objects. You will find database configuration objects in a job configuration's source and in the job configuration's pipeline, which specified the target database.

Table 16. Database Properties

Property name	Type	Description	Default
connectionType	String	Distinguishes between Job Manager, Job Source, Job Target, and Job Checkpoint databases. This field is auto-generated.	
databaseType	Enumeration	The following database types are supported: [DB2, FILES, GEMFIRE, MONGODB, ORACLE, POSTGRES, REDIS, REDIS_STREAMS_MESSAGE_BROKER, SPLUNK, SQL_SERVER, VERTICA] NONE is used for custom stages. Also see userDefinedType. This is a required field.	
databaseURL	String	Database URLs specify the subprotocol (the database connectivity mechanism), the database, or server identifier, and a list of properties. Not required for Gemfire, Splunk, and Files databaseTypes.	
credentialsDirectoryPath	String	Each database type other than NONE (used for custom jobs) requires a credentials property file, even if the credentials are not required. Credentials property files must adhere to the following filename pattern: redisconnect_credentials_{source_type | target_type}_{job_name}.properties The only exception is for Job Manager which has fixed name: `redisconnect_credentials_jobmanager.properties`	

Property name	Type	Description	Default
credentialsRotationEventListenerEnabled	Boolean	When enabled, the credentialsDirectoryPath will be periodically scanned for changes that are specific to the property file associated with this database[1].	false
credentialsRotationEventListenerInterval	Integer	Fixed rate scheduled interval that scans the credentialsDirectoryPath for changes when credentialsRotationEventListenerEnabled is enabled. Measured in milliseconds; minimum is 60000.	60000

[1] If a change is identified, the listener will create a new connection without bringing down the Redis Connect instance nor stopping the job. There might be a momentary pause in pipeline processing while the connection is being reestablished. No data will be lost in this process. Disclaimer: if the new credentials cannot be used to create a connection, the job will be stopped for manual intervention.

Custom Database Properties

RDBMS Properties

Table 17. Stream Job Type Custom Configurations

Property name	Type	Description	Default
column.exclude.list	List	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns to exclude from change event record values. Fully-qualified names for columns are of the form `databaseName.tableName.columnName`. To match the name of a column, Debezium applies the regular expression that you specify as an anchored regular expression. That is, the specified expression is matched against the entire name string of the column; it does not match substrings that might be present in a column name. If you include this property in the configuration, do not also set the column.include.list property.	[]
decimal.handling.mode	String	Specifies how the connector should handle values for DECIMAL and NUMERIC types: “string” encodes values as formatted strings, which is easy to consume but semantic information about the real type is lost. “precise” represents them precisely using java.math.BigDecimal values represented in change events in a binary form. “double” represents them using double values, which may result in a loss of precision but is easier to use.	String
heartbeat.interval.ms	Long	Controls how frequently the connector sends heartbeat messages. Heartbeat messages are useful for ensuring the source transaction/redo logs and checkpoint are updated on a predictable basis. This can avoid a long period without updates which may cause a gap between the latest checkpoint and an archived transaction log that has been deleted. Default (0) does not send heartbeat messages. Measured in milliseconds; minimum is 0.	0

max.batch.size	Integer	Specifies the maximum size of each batch of events that should be processed during each iteration of the producer's polling event loop; minimum is 1.	2048
max.queue.size	Integer	Positive integer value that specifies the maximum number of records that the blocking queue can hold. When Debezium reads events streamed from the database, it places the events in the blocking queue before it writes them to Kafka. The blocking queue can provide backpressure for reading change events from the database in cases where the connector ingests messages faster than it can write them to Kafka, or when Kafka becomes unavailable. Events that are held in the queue are disregarded when the connector periodically records offsets. Always set the value of max.queue.size to be larger than the value of max.batch.size. `Note: max.queue.size > max.batch.size`	8192
snapshot.mode	String	Specifies the criteria for running a snapshot when the connector starts. It is not recommended to use this debezium capability for initial load in Production. See Production Readiness for more information. MySQL, Postgres default to 'never'; Oracle, SQL Server default to 'schema_only' [1].	n/a

Table 18. Initial Load Job Type Custom Configurations

Property name	Type	Description	Default
jdbc.connection.timeout	Long	Maximum duration the pool will wait for a connection to become available. Measured in milliseconds.	30000
jdbc.keepalive.time	Long	Fixed interval to keep a connection alive, in order to prevent it from being timed out by the database or network infrastructure. A "keepalive" will only occur on an idle connection. Default: 0 (disabled) Measured in milliseconds.	0

Property name	Type	Description	Default
jdbc.idle.timeout	Long	Takes effect only when the jdbc.min.idle is less than jdbc.max.pool.size, and is removed when the number of idle connections exceeds the jdbc.min.idle and the idle time exceeds jdbc.idle.timeout. If (jdbc.idle.timeout + 1 second) > jdbc.max.lifetime and jdbc.max.lifetime > 0, it will be reset to 0. If jdbc.idle.timeout != 0 and less than 10 seconds, it will be reset to 10 seconds. If jdbc.idle.timeout=0, idle connections will never be removed from the connection pool. Measured in milliseconds.	600000
jdbc.max.lifetime	Long	Maximum duration a connection is allowed to remain open. When this timeout is exceeded, the next time the connection is released to the pool it is closed, and a new connection is opened to replace it. Measured in milliseconds.	1800000
jdbc.max.pool.size	Integer	Maximum number of connections in the pool.	#partitions + 2
jdbc.min.idle	Integer	Controls the minimum number of connection pool idle connections. When the connection pool idle connections are less than jdbc.min.idle and the total number of connections is not more than jdbc.max.pool.size, it will try its best to supplement new connections.	#partitions
truststore.file.path	String	Holds onto certificates that identify 3rd parties. truststore.type JKS String For Java keystore file format, this property has the value jks (or JKS). You do not normally specify this property, because its default value is already jks.	n/a
keystore.file.path	String	Stores private key entries, certificates with public keys, or just secret keys that we may use for various cryptographic purposes. keystore.type JKS String Depending on what entries the keystore can store and how the keystore can store the entries, there are a few different types of keystores in Java: JKS, JCEKS, PKCS12, PKCS11 and DKS.	N/A

MongoDB Properties

Table 19. Stream Job Type Custom Configurations

Property name	Type	Description	Default
connect.backoff.initial.delay.ms	Integer	Initial delay when trying to reconnect to a primary after the first failed connection attempt or when no primary is available. Measured in milliseconds.	1000
connect.backoff.max.delay.ms	Integer	Maximum delay when trying to reconnect to a primary after repeated failed connection attempts or when no primary is available. Measured in milliseconds.	120000
connect.max.attempts	Integer	Maximum number of failed connection attempts to a replica set primary before an exception occurs and task is aborted.	3
mongodb.authsource	String	Database (authentication source) containing MongoDB credentials. This is required only when MongoDB is configured to use authentication with another authentication database than admin.	
admin.connection.pool.max.size	Integer	Maximum number of connections opened in the pool.	100
connection.pool.min.size	Integer	Minimum number of connections opened in the pool	0
connection.timeout	Integer	Maximum duration to establish a connection before an error is thrown. Depending on network infrastructure and load on the server, the client may have to wait for a connection establishment. Possible scenarios where connection timeout can happen – the server gets shut down, network issues, wrong IP/DNS, port number etc. Measure in milliseconds.	10000
mongodb.ssl.invalid.hostname.allowed	Boolean	When SSL is enabled, this setting controls whether strict hostname checking is disabled during connection phase. If true, the connection will not prevent man-in-the-middle attacks.	False
read.timeout	Integer	Maximum duration to send or receive on a socket before an error is thrown. “0” default indicates disabling the timeout. Measured in milliseconds."	0
mongodb.ssl.enabled	Boolean	Enables TLS/SSL connection.	False

Files Connector Properties

Table 20. Initial Load Job Type Custom Configurations

Property name	Type	Description	Default
charset	String	Defines a mapping between sequences of sixteen-bit UTF-16 code units (that is, sequences of chars) and sequences of bytes.	UTF-16
column.names	String	If a header row is not provided, column names can be configured manually. Order matters for proper association. Supported for delimited types. CSV, PSV, TSV	n/a
continuation.identifier	String	Any arbitrary String (special characters) to concatenate multiple sequential lines.	\\
delimiter	String	Sequence of one or more characters for specifying the boundary between separated column-values.	,
google.cloud.storage.project.id	String	Unique identifier for a project within the GCP console.	n/a
gzip.file.format	Boolean	gzip is a file format and a software application used for file compression and decompression.	False
header.line	Integer	Specifies which row is the header line since some delimited file types have comments or other non-column related headers.	0
included.fields	String	Whitelist of fields/columns that should be included for replication. All others will be ignored.	n/a
includes.header	Boolean	Most delimited file types include a header row which can be used to name each column.	True
initial.offset	Integer	Specifies row/offset from which all scanning should begin inclusive of the header line in delimited types.	0
source.file.path	String	Location of the file to be loaded.	n/a
quote.character	Char	Avoids syntax errors when double quotes are used within field values. Any character can be used as a substitute.	"
type	String	Supported file formats include: CSV, JSON, PSV, TSV, XML.	N/A

Gemfire Properties

Table 21. Custom Configurations

Property name	Type	Description	Default
function.filter.batch.size			
pool.free.connection.timeout	Integer	Number of milliseconds (ms) that the client waits for a free connection if max-connections limit is configured and all connections are in use.	10000
pool.idle.timeout	Integer	Number of milliseconds to wait for a connection to become idle for load balancing	5000
pool.locator.host			
pool.locator.port			
pool.load.conditioning.interval	Integer	Interval in which the pool checks to see if a connection to a specific server should be moved to a different server to improve the load balance.	300000 (5m)
pool.max.connections	Integer	Maximum number of connections that the pool can create. If all connections are in use, an operation requiring a client-to server-connection is blocked until a connection is available or the free-connection-timeout is reached. If set to -1, there is no maximum. The setting must indicate a cap greater than min-connections [2].	-1
pool.min.connections	Integer	Number of connections that must be created initially.	5
pool.multiuser.authentication	Boolean	Sets the pool to use multi-user secure mode. If in multiuser mode, then app needs to get RegionService instance of Cache	False
pool.ping.interval	Integer	Interval between pinging the server to show the client is alive, set in milliseconds. Pings are only sent when the ping-interval elapses between normal client messages. This must be set lower than the server's maximum-time-between-pings.	10000
pool.pr.single.hop.enabled	Boolean	Setting used for single-hop access to partitioned region data in the servers for some data operations. See PartitionResolver. See note in thread-local-connections below.	True

Property name	Type	Description	Default
pool.read.timeout	Integer	Number of milliseconds to wait for a response from a server before the connection times out.	10000
pool.retry.attempts	Integer	Number of times to retry an operation after a time-out or exception for high availability. If set to -1, the pool tries every available server once until it succeeds or has tried all servers.	-1
pool.server.group	String	Server group from which to select connections. If not specified, the global group of all connected servers is used.	n/a
pool.socket.buffer.size	Integer	Size of the socket buffer, in bytes, on each connection established.	32768
pool.socket.connect.timeout	Integer	Amount of time (in seconds) to wait for a response after a socket connection attempt.	59
pool.statistic.interval	Integer	Default frequency, in milliseconds, with which the client statistics are sent to the server. A value of -1 indicates that the statistics are not sent to the server.	-1
pool.subscription.ack.interval	Integer	Number of milliseconds to wait before sending an acknowledgment to the server about events received from the subscriptions.	100
pool.subscription.message.tracking.timeout	Integer	Number of milliseconds for which messages sent from a server to a client are tracked. The tracking is done to minimize duplicate events.	90000
pool.subscription.redundancy	Integer	Redundancy for servers that contain subscriptions established by the client. A value of -1 causes all available servers in the specified group to be made redundant.	0
pool.subscription.timeout.multiplier			
gf:queue:seq			

Redis Target Properties

Table 22. Event Handler Custom Configurations

Property name	Type	Description	Default
domain.strategy	Enumeration	Specifies the domain model that will be used to move data through the pipeline so that custom stages and targets are aligned. Domain Strategies include: DICTIONARY: passes all entities. KEY_ONLY: passes only key. STRING: passes key and single string blob. MESSAGE_BROKER: passes String[] with strict schema KEY_ONLY is used for invalidation and notification. STRING is used with REDIS_STRING_SINK. MESSAGE_BROKER is used for brokered deployments that stream data through Redis Streams. Only Splunk is currently supported for MESSAGE_BROKER.	n/a
databaseURL	String	See Lettuce's documentation for Redis URI syntax - https://lettuce.io/core/release/api/io/lettuce/core/RedisURI.html	N/A
redis.connection.insecure	Boolean	Passed to Lettuce's RedisURI.verifyPeer. If true then verifyMode=FULL, else if false then verifyMode=NONE. When peer verification is disabled, Lettuce uses Netty's InsecureTrustManagerFactory.INSTANCE as the trust manager factory. It's javadoc notes that it should never be used in production and that it is purely for testing purposes." FALSE redis.connection.timeout.duration Integer "Command timeout begins: When the command is sent successfully to the transport. Queued while the connection was inactive. The timeout is canceled upon command completion/cancellation. Measured in seconds.	1

Property name	Type	Description	Default
redis.connection.auto.reconnect	Boolean	Controls auto-reconnect behavior on connections. As soon as a connection gets closed/reset without the intention to close it, the client will try to reconnect, activate the connection and re-issue any queued commands. This flag also has the effect that disconnected connections will refuse commands and cancel these with an exception.	True
redis.connection.suspend.reconnect.on.protocol.failure	Boolean	If this flag is true the reconnect will be suspended on protocol errors. The reconnect itself has two phases: Socket connection and protocol/connection activation. In case a connection timeout occurs, a connection reset, host lookup fails, this does not affect the cancellation of commands. In contrast, where the protocol/connection activation fails due to SSL errors or PING before activating connection failure, queued commands are canceled.	True
redis.connection.sslEnabled	Boolean	Enables use of SSL for one-way or mutual authentication. If this flag is false, truststore and keystore will not be passed to the client.	False
redis.streams.max.length	Integer	Redis will trim the stream from the oldest entries when it reaches the number of entries specified in redis.streams.max.length. The stream could have more entries than specified in the command before Redis starts deleting entries.	0
redis.streams.passthrough.source.tx.time.enabled	Boolean	Redis Stream IDs are specified by two numbers separated by a - character. The first part is the Unix time in milliseconds of the Redis instance generating the ID. The second part is just a sequence number and is used in order to distinguish IDs generated in the same millisecond. When enabled, the source commit timestamp can be used to manually set the first part of the Redis Stream ID. Caution - time order is strictly enforced.	False

Property name	Type	Description	Default
redis.streams.sink.partitions	Integer	REDIS_STREAMS_SINK can partition the Redis Streams keys to which it will commit. This is complementary to job-level partitioning and does not replace it. If only the sink is partitioned a partition Id would be appended to the end of the Redis Streams key (e.g. key:2). If the sink is partitioned and the job was partitioned then first the job partition id would be appended to the Redis Streams key followed by the sink partition id (e.g. key:2:4).	0
redis.wait.enabled	Boolean	Blocks sink's commit from successfully completing until acknowledgment that the data was replicated to a backup Redis database shard. If checkpointTransactionsEnabled=true, the checkpoint will be rolledback to before this change occurred to avoid inconsistency and duplication. If checkpointTransactionsEnabled=false, the checkpoint won't be rolled back so on restart the operation would be duplicated.	False
redis.wait.timeout	Integer	If the timeout is reached, the sink's commit will fail and the consequence will be based on redis.wait.timeout.stop.job.enabled. Measured in milliseconds.	3000
redis.wait.timeout.stop.job.enabled	Boolean	If enabled, the job will be stopped and the quiesce process will be bypassed on the assumption that Redis is unavailable. If disabled, the error will be logged but the job will continue to operation. This introduces the risk that the data could be lost if the primary shard fails.	True

Redis Message Broker Properties

Table 23. Stream Job Type Custom Configurations

Property name	Type	Description	Default
redis.broker.eviction.strategy	Enumeration	If domain.strategy=MESSAGE_BROKER, Redis Streams is used as an event-stream(s) therefore requires cleanup to avoid running out of memory. To evict change-data events that are beyond the need for recovery, the following two options exist: THRESHOLD - evicts based on queue depth. SCHEDULER - evicts based on a scheduled thread.	THRESHOLD
redis.broker.eviction.scheduled.interval	Long	Fixed rate scheduled thread that evicts Redis Stream offsets that occurred before the oldest existing checkpoint. Measured in seconds.	10
redis.broker.eviction.threshold	Integer	Fixed queue depth threshold which triggers a separate thread to evict Redis Stream offsets that occurred before the oldest existing checkpoint. If redis.streams.eviction.strategy=THRESHOLD and redis.streams.eviction.threshold is not provided and job type is PARTITIONED_STREAM then redis.streams.eviction.threshold=redis.streams.max.length / partitions; If redis.streams.eviction.strategy=THRESHOLD and redis.streams.eviction.threshold is not provided and job type is STREAM then (redis.broker.max.queue.depth * 7) / 10;	n/a
redis.broker.max.queue.depth	Integer	Used to calculate redis.broker.eviction.threshold if it's not manually configured.	32768

Splunk Stream Properties

Table 24. Job Type Custom Configurations

Property name	Type	Description	Default
httpHeaders	String		
source.time.field.name	String		
source.time.sequence.field.name	String		

Property name	Type	Description	Default
authorization.stored.in.credentials.file	Boolean		
splunk.forwarder.destination.url	String		
http.post.max.retry.attempts	Integer		
http.post.max.retry.duration	Integer		
http.post.retry.delay.factor	Integer		
http.post.retry.delay.interval	Integer		
http.post.retry.max.delay.interval	Integer		

In-Memory Queue Properties

Table 25. Change Event Queue Custom Configurations

Property name	Type	Description	Default
poll.interval.ms	Long	Fixed rate interval that specifies the number of milliseconds the connector should wait for new changed-data events to appear before it starts processing a batch of events. Measured in milliseconds.	500
max.batch.size	Integer	Maximum events that can be dequeued by the source poll event loop within a single iteration.	16384
max.queue.size	Integer	Specifies the maximum number of records that the blocking queue can hold. [3].	32768
queue.persistence.enabled	Boolean	When enabled, batches of changed-data events are persisted to Redis Stream, before they are enqueued within the in-memory queue, which effectively mimics a change-data-capture (CDC) process within Redis Connect. [4].	True
queue.persistence.wait.enabled	Boolean	Blocks persistence from successfully completing until acknowledgment that the data was replicated to a backup Redis database shard completed. FALSE	queue.persistence.wait.timeout

[1] Possible settings are: initial - the connector runs a snapshot only when no offsets have been recorded for the logical server name. initial_only - the connector runs a snapshot only when no offsets have been recorded for the logical server name and then stops; i.e. it will not read change events from the binlog. when_needed - the connector runs a snapshot upon startup whenever it deems it necessary. That is, when no offsets are available, or when a previously recorded offset specifies a binlog location or GTID that is not available in the server. never - the connector never uses snapshots. Upon first startup with a logical server name, the connector reads from the beginning of the binlog. Configure this behavior with care. It is valid only when the binlog is guaranteed to contain the entire history of the database. schema_only - the connector runs a snapshot of the schemas and not the data. This

setting is useful when you do not need the topics to contain a consistent snapshot of the data but need them to have only the changes since the connector was started. schema_only_recovery - this is a recovery setting for a connector that has already been capturing changes. When you restart the connector, this setting enables recovery of a corrupted or lost database schema history topic. You might set it periodically to "clean up" a database schema history topic that has been growing unexpectedly. Database schema history topics require infinite retention

[2] If you use this setting to cap your pool connections, deactivate the pool attribute pr-single-hop-enabled. Leaving single hop activated can increase thrashing and lower performance.

[3] The blocking queue provides backpressure for reading changed data events from the source in cases where the connector ingests messages faster than they are consumed. Events that are held in the queue are disregarded when the connector periodically records offsets. Always set the value of maxQueueSize to be larger than the value of maxBatchSize

[4] Persistence allows for recovery of transient changed-data events which is critical for sources like Splunk that do not implement their own change-data-capture (CDC) process. Only supported by Gemfire and Splunk