
Solid Cache

Upgrading from v0.3.0 or earlier? Please see upgrading to version v0.4.x and beyond

Solid Cache is a database-backed Active Support cache store implementation.

Using SQL databases backed by SSDs we can have caches that are much larger and cheaper than traditional memory only Redis or Memcached backed caches.

Usage

To set Solid Cache as your Rails cache, you should add this to your environment config:

```
1 config.cache_store = :solid_cache_store
```

Solid Cache is a FIFO (first in, first out) cache. While this is not as efficient as an LRU cache, this is mitigated by the longer cache lifespan.

A FIFO cache is much easier to manage: 1. We don't need to track when items are read 2. We can estimate and control the cache size by comparing the maximum and minimum IDs. 3. By deleting from one end of the table and adding at the other end we can avoid fragmentation (on MySQL at least).

Installation

Add this line to your application's Gemfile:

```
1 gem "solid_cache"
```

And then execute:

```
1 $ bundle
```

Or install it yourself as:

```
1 $ gem install solid_cache
```

Add the migration to your app:

```
1 $ bin/rails solid_cache:install:migrations
```

Then run it:

```
1 $ bin/rails db:migrate
```

Configuration

Configuration will be read from `config/solid_cache.yml`. You can change the location of the config file by setting the `SOLID_CACHE_CONFIG` env variable.

The format of the file is:

```
1 default:
2   store_options: &default_store_options
3     max_age: <%= 60.days.to_i %>
4     namespace: <%= Rails.env %>
5     size_estimate_samples: 1000
6
7 development: &development
8   database: development_cache
9   store_options:
10     <<: *default_store_options
11     max_size: <%= 256.gigabytes %>
12
13 production: &production
14   databases: [production_cache1, production_cache2]
15   store_options:
16     <<: *default_store_options
17     max_entries: <%= 256.gigabytes %>
```

For the full list of keys for `store_options` see Cache configuration. Any options passed to the cache lookup will overwrite those specified here.

Connection configuration You can set one of `database`, `databases` and `connects_to` in the config file. They will be used to configure the cache databases in `SolidCache::Record#connects_to`.

Setting `database` to `cache_db` will configure with:

```
1 SolidCache::Record.connects_to database: { writing: :cache_db }
```

Setting `databases` to `[cache_db, cache_db2]` is the equivalent of:

```
1 SolidCache::Record.connects_to shards: { cache_db1: { writing: :
    cache_db1 }, cache_db2: { writing: :cache_db2 } }
```

If `connects_to` is set it will be passed directly.

If none of these are set, then Solid Cache will use the `ActiveRecord::Base` connection pool. This means that cache reads and writes will be part of any wrapping database transaction.

Engine configuration There are three options that can be set on the engine:

-
- `executor` - the Rails executor used to wrap asynchronous operations, defaults to the app executor
 - `connects_to` - a custom connects to value for the abstract `SolidCache::Record` active record model. Required for sharding and/or using a separate cache database to the main app. This will overwrite any value set in `config/solid_cache.yml`
 - `size_estimate_samples` - if `max_size` is set on the cache, the number of the samples used to estimates the size.

These can be set in your Rails configuration:

```
1 Rails.application.configure do
2   config.solid_cache.size_estimate_samples = 1000
3 end
```

Cache configuration Solid Cache supports these options in addition to the standard `ActiveSupport::Cache::Store` options.

- `error_handler` - a Proc to call to handle any `ActiveRecord::ActiveRecordErrors` that are raises (default: log errors as warnings)
- `expiry_batch_size` - the batch size to use when deleting old records (default: 100)
- `expiry_method` - what expiry method to use `thread` or `job` (default: `thread`)
- `expiry_queue` - which queue to add expiry jobs to (default: `default`)
- `max_age` - the maximum age of entries in the cache (default: `2.weeks.to_i`). Can be set to `nil`, but this is not recommended unless using `max_entries` to limit the size of the cache.
- `max_entries` - the maximum number of entries allowed in the cache (default: `nil`, meaning no limit)
- `max_size` - the maximum size of the cache entries (default `nil`, meaning no limit)
- `cluster` - a Hash of options for the cache database cluster, e.g { `shards: [:database1, :database2, :database3]` }
- `clusters` - an Array of Hashes for multiple cache clusters (ignored if `cluster` is set)
- `active_record_instrumentation` - whether to instrument the cache's queries (default: `true`)
- `clear_with` - clear the cache with `:truncate` or `:delete` (default `truncate`, except for when `Rails.env.test?` then `delete`)
- `max_key_bytesize` - the maximum size of a normalized key in bytes (default 1024)

For more information on cache clusters see Sharding the cache

Cache expiry

Solid Cache tracks writes to the cache. For every write it increments a counter by 1. Once the counter reaches 50% of the `expiry_batch_size` it adds a task to run on a background thread. That task will:

1. Check if we have exceeded the `max_entries` or `max_size` values (if set). The current entries are estimated by subtracting the max and min IDs from the `SolidCache::Entry` table. The current size is estimated by sampling the entry `byte_size` columns.
2. If we have, it will delete `expiry_batch_size` entries.
3. If not, it will delete up to `expiry_batch_size` entries, provided they are all older than `max_age`.

Expiring when we reach 50% of the batch size allows us to expire records from the cache faster than we write to it when we need to reduce the cache size.

Only triggering expiry when we write means that if the cache is idle, the background thread is also idle.

If you want the cache expiry to be run in a background job instead of a thread, you can set `expiry_method` to `:job`. This will enqueue a `SolidCache::ExpiryJob`.

Using a dedicated cache database

Add database configuration to `database.yml`, e.g.:

```
1 development:
2   cache:
3     database: cache_development
4     host: 127.0.0.1
5     migrations_paths: "db/cache/migrate"
```

Create database:

```
1 $ bin/rails db:create
```

Install migrations:

```
1 $ bin/rails solid_cache:install:migrations
```

Move migrations to custom migrations folder:

```
1 $ mkdir -p db/cache/migrate
2 $ mv db/migrate/*.solid_cache.rb db/cache/migrate
```

Set the engine configuration to point to the new database:

```
1 # config/solid_cache.yml
2 production:
3   database: cache
```

Run migrations:

```
1 $ bin/rails db:migrate
```

Sharding the cache

Solid Cache uses the Maglev consistent hashing scheme to shard the cache across multiple databases.

To shard:

1. Add the configuration for the database shards to database.yml
2. Configure the shards via `config.solid_cache.connects_to`
3. Pass the shards for the cache to use via the cluster option

For example:

```
1 # config/database.yml
2 production:
3   cache_shard1:
4     database: cache1_production
5     host: cache1-db
6   cache_shard2:
7     database: cache2_production
8     host: cache2-db
9   cache_shard3:
10    database: cache3_production
11    host: cache3-db
```

```
1 # config/solid_cache.yml
2 production:
3   databases: [cache_shard1, cache_shard2, cache_shard3]
```

Secondary cache clusters

You can add secondary cache clusters. Reads will only be sent to the primary cluster (i.e. the first one listed).

Writes will go to all clusters. The writes to the primary cluster are synchronous, but asynchronous to the secondary clusters.

To specific multiple clusters you can do:

```
1 # config/solid_cache.yml
2 production:
3   databases: [cache_primary_shard1, cache_primary_shard2,
4               cache_secondary_shard1, cache_secondary_shard2]
5   store_options:
6     clusters:
7       - shards: [cache_primary_shard1, cache_primary_shard2]
7       - shards: [cache_secondary_shard1, cache_secondary_shard2]
```

Named shard destinations

By default, the node key used for sharding is the name of the database in `database.yml`.

It is possible to add names for the shards in the cluster config. This will allow you to shuffle or remove shards without breaking consistent hashing.

```
1 production:
2   databases: [cache_primary_shard1, cache_primary_shard2,
3               cache_secondary_shard1, cache_secondary_shard2]
4   store_options:
5     clusters:
6       - shards:
7         cache_primary_shard1: node1
7         cache_primary_shard2: node2
8       - shards:
9         cache_secondary_shard1: node3
10        cache_secondary_shard2: node4
```

Enabling encryption

Add this to an initializer:

```
1 ActiveSupport.on_load(:solid_cache_entry) do
2   encrypts :value
3 end
```

Index size limits

The Solid Cache migrations try to create an index with 1024 byte entries. If that is too big for your database, you should:

1. Edit the index size in the migration
2. Set `max_key_bytesize` on your cache to the new value

Development

Run the tests with `bin/rake test`. By default, these will run against SQLite.

You can also run the tests against MySQL and PostgreSQL. First start up the databases:

```
1 $ docker compose up -d
```

Next, setup the database schema:

```
1 $ TARGET_DB=mysql bin/rails db:setup
2 $ TARGET_DB=postgres bin/rails db:setup
```

Then run the tests for the target database:

```
1 $ TARGET_DB=mysql bin/rake test
2 $ TARGET_DB=postgres bin/rake test
```

Testing with multiple Rails version

Solid Cache relies on appraisal to test multiple Rails version.

To run a test for a specific version run:

```
1 bundle exec appraisal rails-7-1 bin/rake test
```

After updating the dependencies in the `Gemfile` please run:

```
1 $ bundle
2 $ appraisal update
```

This ensures that all the Rails versions dependencies are updated.

License

Solid Cache is licensed under MIT.