Garbage collection for the disk cache

Please read Bazel Code of Conduct before commenting.

- **Authors**: tjgq@google.com

- **Status**: Draft | **In review** | Approved | Rejected | In progress | Implemented

- **Reviewers**: chiwang@google.com

- **Created**: 2024-01-16
- **Updated**: 2024-03-13
- **Discussion thread**: https://github.com/bazelbuild/bazel/issues/5139

Overview

This document describes how we plan to implement automatic garbage collection for the Bazel disk cache.

Background

The Bazel disk cache, controlled by the `--disk_cache` flag, makes it possible to store build results in a well-known location in the local filesystem and share them across space (workspaces) and time (invocations).

Internally, the disk cache comprises two stores: the Content Addressable Store (CAS) stores action inputs and outputs, and the Action Cache (AC) stores `ActionResult` protos in wire format, which may in turn reference CAS entries. Both stores are content-addressed; each entry is stored as a file whose name is the content hash.

The disk cache does not currently implement an expiration policy; in the absence of a manual cleanup, any entry added to the disk cache will be kept forever. This has been a long-standing pain point.

Starting with Bazel 7.0.0, the filesystem modification time (mtime) of a cache entry is always updated on access.[1] This makes it possible for a manually triggered external process to do garbage collection by sorting files by mtime and deleting the oldest ones. The goal of this project is to teach Bazel to do so all by itself.

We plan to merge these changes into the 7.x tree, tentatively in the 7.2.0 release.

[1] The decision to use mtime instead of atime is intentional: the atime is more likely to be externally modified, and unreliable in the presence of the `noatime` or `relatime` mount options on Linux.

Requirements

Goals

We aim to achieve the following:

- Provide an optional knob to set the user-supplied target size for the disk cache.
- Keep the disk cache under the target size by deleting the least recently accessed entries to make space for newly inserted ones.
- Keep the disk cache in a self-consistent state at all times.
- Preserve the ability to use a disk cache stored in a **local filesystem**, possibly used by multiple Bazel instances running on that machine, either concurrently or at different times.

Non-goals

We do **not** aim to achieve the following at this time:

- Preserve the current performance of a disk cache when garbage collection is enabled. We will endeavor to keep the synchronization overhead low, but a slight slowdown might be unavoidable, even when at most a single Bazel instance is accessing the cache at any given time (since concurrent actions within an invocation must also synchronize). We expect this to be mitigated by an eventual `--experimental_remote_cache_async` flag flip.
- Support garbage collection for a disk cache stored in a **network filesystem** (e.g. NFS or SMB). Given the uncertainties around network filesystem support for locking primitives, we judge it to not be worth the additional complexity. Use of a disk cache stored in a network filesystem remains possible with garbage collection disabled.
- Support setting a cache size smaller than the total output yield of a single build. Bazel assumes that no cache entry depended upon by the current build is evicted from the cache before the build is over, and lifting this restriction is nontrivial.

Proposed implementation

Command-line flag

We'll add a new `--experimental_disk_cache_max_size` flag accepting a value matching the regex `[0-9]+[MGT]` and setting its target size to the given number of MiB, GiB or TiB, depending on the last character.

The `--experimental_disk_cache_max_size` flag may be set in the absence of `--disk_cache`, but it will be ineffectual. This makes it more convenient to include it in a shared `.bazelrc`.

Reserved files

The disk cache is laid out in a 3-level directory hierarchy:

- Hash function (directory name is the hash function name)
- Store (directory name is one of `cas` or `ac`)
- Hash prefix (directory name is in the range `00`…`ff`, for the first 8 bits of the hash)

For legacy reasons, some of the hash functions are stored in top-level `cas` and `ac` directories, and distinguished only by hash length.

In order to ensure that a cache populated by a future Bazel N+1 can be correctly cleaned up by Bazel N, we must be careful not to hardcode the set of known hash functions. At the same time, the need for coordination between Bazel servers sharing a cache requires the ability to create files that should not be targeted by garbage collection (see later sections). Therefore, we reserve the top-level `ctl` directory to store these files.

Garbage collection algorithm

Whenever an entry is about to be inserted into the cache, Bazel must first ensure there is sufficient space for it. This entails the following (conceptual) steps:

1. List all existing entries and obtain their size and mtime.
2. Compute the cumulative size of existing entries.
3. If the cumulative size leaves enough room for the new entry to be inserted, no garbage collection is required. Otherwise:
4. Sort existing entries by ascending mtime.
5. Delete existing entries one at a time in sort order, until the cumulative size leaves enough room for the new entry to be inserted.

AC and CAS entries are treated identically by garbage collection. Deleting entries by ascending mtime preserves referential integrity even if the process is interrupted at any time, because when Bazel refreshes an AC entry it also refreshes the CAS entries referenced by it (to a value slightly larger than the AC entry's). But the converse is not true; an incremental build may refresh a CAS entry without also refreshing AC entries referencing it. Therefore we cannot use AC entries to drive collection of CAS entries.

If all else fails, dangling references are still not capable of causing a build failure, because Bazel only considers an AC entry to be valid if all of the CAS entries it references are also present in the cache. However, Bazel does assume that a CAS entry that was determined to be present during a build remains so until the end of that build; as a corollary, to ensure correct operation, the cache must be at least large enough to contain the output yield of an entire build.

Performance challenges

There are two problems with a straightforward implementation of the algorithm described in the previous section.

First, it would require a large number of filesystem operations to run at arbitrary points during the build. For a very large cache, this would incur in significant I/O overhead and noticeable pauses. For example, on an M1 MacBook Pro, obtaining the sizes and mtimes of a cache with ~1M files takes about 8s, even if done in parallel while minimizing the number of system calls. This work can be moved into the background by setting `--experimental_remote_cache_async`, but the I/O overhead is still concerning.

Second, a filesystem scan is inherently racy when concurrent Bazel instances share a cache, which could cause the target size to not be scrupulously observed at all times. Using a locking mechanism to prevent an instance from accessing the cache while another is scanning it would make pauses even more noticeable.

Index

In order to address the challenges described in the previous section, we introduce an index to speed up garbage collection and provide coordination between concurrent accesses.

The purpose of the index is to make it possible to perform the following operations quickly (under ~1ms), with minimal I/O, and guarding against corruption by concurrent accesses:

- Get the current cumulative cache size
- Get the cache entry with the smallest mtime
- Insert, delete or update the index for an entry

The index is stored at `ctl/index`, which is expected to be very small in relation to the cache contents (ballpark of 200 bytes per cache entry). More importantly, all data is contained in a single file instead of scattered across thousands of directory entries.

The exact file format is yet to be finalized, but the most promising option is to use a SQLite database, which provides the required primitives and performance characteristics.

Initial index creation

Bazel should gracefully handle the case where a cache has been previously populated but doesn't yet contain an index. In such a situation, the index will be generated from scratch upon the first Bazel invocation capable of garbage collection. While this might introduce a noticeable pause for the first such invocation (ballpark of 10s per 1M entries), the index creation can proceed in parallel with analysis, and it will not recur in subsequent invocations.

If the cache is determined to already exceed the target size upon index creation, a garbage collection immediately occurs before the build proceeds.

Incremental index updates

Every cache access either inserts a new entry (possibly deleting older entries first) or updates the mtime for an existing entry, thereby requiring the index to be updated. When updating an mtime, both the file and the index must be updated: files remain the source of

truth and the index merely provides hints, which may become stale and require a recomputation (see discussion of mismatches below).

Concurrent updates to the index must be synchronized to avoid corruption; we will likely leverage SQLite transactions to do so, which restricts the number of writers to one at a time (see discussion of concurrent access below).

Backwards compatibility

It remains possible to use a Bazel without garbage collection support, or with garbage collection disabled, against a cache whose index has been populated. This will cause the index to not be updated on access and to get out of sync with the actual contents. In such cases, we make no guarantees that the target size will be respected.

Index recreation

In the interest of preventing failures to incrementally update the index to become sticky, mismatches between the index and the actual cache contents will be detected on a best-effort basis.

Whenever a mismatch is detected during a build, the build will proceed normally (at the risk of not respecting the target size) but the index will be recreated from scratch at the end of the build, while the server is idle. This might trigger an immediate garbage collection, similar to the initial creation case.

Since not every mismatch is guaranteed to be detected on a best-effort basis, the index will also be periodically recreated from scratch, also while the server is idle.

Hysteresis

Whenever a garbage collection is required to make space for a new entry, it is advantageous to create more headroom than strictly needed for that one entry, so that the cache isn't constantly on the verge of overflowing.

If the cache target size is T, and the entry to be added has size S, a garbage collection should bring the cumulative size down to min(T - S, F * T), where F (the hysteresis factor) is a constant between 0 and 1. The value of F may be either hardcoded, if one that generally works well can be found through experimentation, or provided explicitly by the user.

Concurrent access

Since it's possible for concurrent Bazel instances to simultaneously attempt to create the index, creation must be atomic. A locking scheme on the index file will be used to ensure that at most one instance is attempting to create it at any given time, and that other instances will wait for it to finish.

Synchronization between cache accesses (either by concurrent Bazel instances or within a single Bazel instance running concurrent build actions) might also cause write contention on the index file as we incrementally update it during a build, potentially slowing it down. This is

considered an acceptable design tradeoff. Users who depend on heavy concurrency and care about the performance impact have the option of disabling garbage collection and running a manual cleanup script in between builds.

For a cache residing on a network share, the locking primitives required for safe concurrent access might not be available, and index corruption is possible. We will warn about network shares on a best-effort basis, but make no guarantees that garbage collection will work properly.

Abandoned alternatives

Offline collection with soft size limit

An earlier version of this proposal only contemplated "offline" garbage collection, i.e, by an idle task running in between builds. This was widely perceived as a significant limitation, because it would remain impossible to keep the cache size under control at all times. As a result, the proposal was redesigned into its current "online" form.

Storing the entire cache in a SQLite database

Given that the current proposal already entails using a SQLite database to store the index, we could go even further and store the entire cache contents in the database. However, there are some challenges with such an approach:

- Heavy contention would become much more likely, possibly requiring sharding.
- Huge blobs would have to be split to respect SQLite's 2GB limit on blob size.
- The entire database must fit in a single file, which may run into filesystem limitations.
- It would be backwards incompatible with the existing format (likely not a big deal).

We might reevaluate this alternative after gaining more experience with SQLite as the index implementation, as there is an advantage to implementing the entire cache using a single unified abstraction.

Incremental collection

The preceding discussion assumes that garbage collection requires knowledge of the entire cache contents, so that the total size can be computed, and the globally oldest entries can be found and deleted. However, since the cache space is partitioned into identically distributed buckets per hash function and hash prefix, it ought to be possible to garbage-collect each bucket independently in a round-robin fashion.

It's unclear that this would work out in practice. Even though scanning a single bucket would be much faster and could even obviate the need for a global index, there might be too many buckets (due to multiple hash functions stored in the same cache) to garbage-collect effectively and uniformly using this strategy.

Compatibility

Sharing a disk cache between a Bazel version with garbage collection support and another Bazel version without it (or with garbage collection disabled, or set to a different target size) might result in inability to keep the cache under the requested size at all times, regardless of whether the accesses are concurrent or disjoint in time.

Document History

Date	Description
2024-01-16	Initial version
2024-01-22	Addressed internal review comments
2024-03-13	Redesigned from “offline” to “online” garbage collection