Using Remote Cache Service for Bazel

SAVE TIME BY SHARING AND REUSING BUILD AND TEST OUTPUT

ALPHA LAM

Software projects today are getting more and more complex. Code accumulates over the years as organization growth increases the volume of daily commits. Projects that used to take minutes to complete a full build now start with fetching from the repository and may require an hour or more to build.

A developer who maintains the infrastructure constantly has to add more machines to support the ever-increasing workload for builds and tests, at the same time facing pressure from users who are unhappy with the long submit time.

Running more parallel jobs helps, but this is limited by the number of cores on the machine and how parallelizable the build is. Incremental builds certainly help but might not apply if clean builds are needed for production releases. Having many build machines also increases maintenance.

Bazel (<https://bazel.build/>) provides the power to run build tasks remotely and massively parallel. Not every organization, however, can afford to have an in-house

remote execution farm. For most projects a remote cache is a great way to boost performance for build and test by sharing build outputs and test outputs among build workers and workstations. This article details the remote cache feature in Bazel (<https://docs.bazel.build/versions/master/remote-caching.html>) and examines options for building your own remote cache service. In practice, this can reduce the build time by almost an order of magnitude.

HOW DOES IT WORK?

Users run Bazel (<https://docs.bazel.build/versions/master/user-manual.html>) by specifying targets to build or test. Bazel determines the dependency graph of actions to fulfill the targets after analyzing the build rules. This process is incremental, as Bazel will skip the already completed actions from the last invocation in the workspace directory. After that, it goes into the execution phase and executes actions according to the dependency graph. This is when the remote cache and execution systems come into play.

An action in Bazel consists of a command, arguments to the command, and the environment variables, as well as the list of input files and list of output files. It also contains the description of the platform for remote execution, which is outside the scope of this article. The information about an action can be encoded into a protocol buffer (<https://developers.google.com/protocol-buffers/>), which works as a fingerprint of the action. It contains the command, arguments, and environment variables combined as a digest and a Merkle tree digest from the input files. The Merkle tree is generated as follows:

files are the leaf nodes and are digested using their corresponding content; directories are the tree nodes and are digested using digests from their subdirectories and children files. Bazel uses SHA-256 as the default hash function to compute the digests.

Before executing an action, Bazel constructs the protocol buffer using the process above. It is then digested to look up the remote action cache, known as the action digest or action key. If there is a hit, the result contains a list of output files or output directories and their corresponding digests. Bazel downloads the contents of a file using the file digest from the CAS (content-addressable store). Looking up the digest of an output directory from the CAS results in the contents of the entire directory tree, including subdirectories, files, and their corresponding digests. Once all the output file directories are downloaded, the action is completed without the need to execute locally.

The cost of completing this cached action comes from the computation of digests of input files and the network round trips for the lookup and transfer of the output files. This cost is usually substantially less than executing the action locally.

In case of a miss, the action is executed locally, and each of the output files is uploaded to the CAS and indexed by the content digests. Standard output and error are uploaded similarly to files. The action cache is then updated to record the list of output files, directories, and their digests.

Because Bazel treats build actions and test actions equally, this mechanism also applies to running tests. In this case, the inputs to a test action will be the test

executable, runtime dependencies, and data files.

The scheme does not rely on incremental state, as an action is indexed by a digest computed from its immediate inputs. This means once the cache is populated, running a build or test on a different machine will reuse all the already-computed outputs as long as the source files are identical. A developer can iterate on the source code; then build outputs from every iteration will be cached and can be reused.

Another key design element is that cache objects in the action cache and CAS can be independently evicted, as Bazel will fall back to local execution in the case of a cache miss or error reading from either one. The number of cache objects will grow over time since Bazel does not actively delete. It is the responsibility of the remote cache service to perform eviction.

REMOTE CACHE USAGE

Two storage buckets are involved in the remote cache system: a CAS that stores files and directories and an action cache that stores the list of output files and directories. Bazel uses the HTTP/1.1 protocol (<https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>) to access these two storage buckets. The storage service needs to support two HTTP methods for each of the storage buckets: the PUT method, which uploads the content for a binary blob, and the GET method, which downloads the content of a binary blob.

The most straightforward way to enable this feature with Bazel is to add the flags in the following example to the `~/.bazelrc` file:

```
build --remote_http_cache=http://build/cache
build --experimental_remote_spawn_cache
```

This enables remote cache with local sandboxed execution.

The first flag, `--remote_http_cache`, specifies the URL of the remote cache service. In this example Bazel uses the path `/ac/` [i.e., `http://build/cache/ac`] to access the action cache bucket and the path `/cas/` [i.e., `http://build/cache/cas`] to access the storage bucket for the CAS.

The second flag, `--experimental_remote_spawn_cache`, enables the use of remote cache for eligible actions with sandboxed execution in case of a cache miss. When downloading from or uploading to a bucket, the last segment of the path (also known as a slug) is a digest.

The next example shows two possible URLs that Bazel might use to access the cache service:

```
http://build/cache/cas/cf80cd8ae-
d482d5d1527d7dc72fceff84e6326592848447d-
2dc0b0e87dfc9a90 http://build/cache/
ac/cf80cd8aed482d5d1527d7dc72fcef-
f84e6326592848447d2dc0b0e87dfc9a90
```

To more finely control the kinds of actions that will use the remote cache without local sandboxed execution, you can use the flags shown in the following example. Individual actions can be opted in to use the remote cache service by using the `--strategy=<action_name>=remote` flag.

```
build --remote_http_cache=http://build/cache
build --spawn_strategy=remote
build --genrule_strategy=remote
build --strategy=Javac=remote
```

The default behavior of Bazel is to read from and write to the remote cache, which allows all users of the remote cache service to share build and test outputs. This feature has been used in practice for a Bazel build on machines with identical configurations in order to guarantee identical and reusable build outputs.

Bazel also has experimental support for using a gRPC (gRPC Remote Procedure Call) service to access the remote cache service. This feature might provide better performance but may not have a stable API. The Bazel Buildfarm project (<https://github.com/bazelbuild/bazel-buildfarm>) implements this API.

IMPLEMENTING A CACHE SERVICE

An HTTP service that supports PUT and GET methods with URLs in forms similar to the second example in the previous section can be used by Bazel as the remote cache service. A few successful implementations have been reported.

Google Cloud Storage (<https://cloud.google.com/storage/>) is the easiest to set up if you are already a user. It is fully managed, and you are billed depending on storage needs and network traffic. This option provides good network latency and bandwidth if your development environment and build infrastructure are already hosted in Google Cloud. It might not be a good option if you have network restrictions or the build infrastructure is not located in the same region. Similarly, Amazon S3 (Simple Storage Service; <https://aws.amazon.com/s3/>) can be used.

For onsite installation, nginx (<https://nginx.org/en/>) with the WebDAV (Web Distributed Authoring and Versioning)

module (http://nginx.org/en/docs/http/nginx_http_dav_module.html) will be the simplest to set up but lacks data replication and other reliability properties if installed on a single machine.

Figure 1 shows an example system architecture implementation of a distributed Hazelcast (<https://hazelcast.com/>) cache service (<https://hazelcast.com/use-cases/caching/cache-as-a-service/>) running in Kubernetes (<https://kubernetes.io/>). Hazelcast is a distributed in-

FIGURE 1: **REMOTE CACHE SERVICE USING OPEN-SOURCE COMPONENTS**

memory cache running in a JVM (Java Virtual Machine). It is used as a CaaS (cache-as-a-service) with support for the HTTP/1.1 interface. In the figure, two instances of Hazelcast nodes are deployed using Kubernetes and configured with asynchronous data replication within the cluster. A Kubernetes Service (<https://kubernetes.io/docs/concepts/services-networking/service/>) is configured to expose a port for the HTTP service, which is load-balanced within the Hazelcast cluster. Access metrics and data on the health of the JVM are collected via JMX (Java Management Extensions). This example architecture is more reliable than a single-machine installation and easily scalable in terms of QPS (queries per second) and storage capacity.

You can also implement your own HTTP cache service to suit your needs. Implementing the gRPC interface for a remote cache server is another possible option, but the APIs are still under development.

In all implementations of the cache service it is important to consider cache eviction. The action cache and CAS will grow indefinitely since Bazel does not perform any deletions. Controlling the storage footprint is always a good idea. The example Hazelcast implementation in figure 1 can be configured to use a least recently used eviction policy with a cap on the number of cache objects together with an expiration policy. Users have also reported success with random eviction and by emptying the cache daily. In any case, recording metrics about cache size and cache hit ratio will be useful for fine-tuning.

BEST PRACTICES

Following the best practices outlined here will avoid

incorrect results and maximize the cache hit rate. The first best practice is to write your build rules without any side effects. Bazel tries very hard to ensure hermeticity by requiring the user to explicitly declare input files to any build rule. When the build rules are translated to actions, input files are known and must present during execution. Actions are executed in a sandbox by default, and then Bazel checks that all the declared output files are created. You can, however, still write a build rule with side effects using `genrule` or a custom action written in the Skylark language (<https://docs.bazel.build/versions/master/skylark/language.html>), used for extensions. An example is writing to the temporary directory and using the temporary files in a subsequent action. Undeclared side effects will not be cached and might cause flaky build failures regardless of whether remote cache is used.

Some built-in rules such as `cc_library` and `cc_binary` have implicit dependencies on the toolchain installed on the system and on system libraries. Because they are not explicitly declared as inputs to an action, they are not included in the computation of the action digest for looking up the action cache. This can lead to the reuse of object files compiled with a different compiler or from a different CPU architecture. The resulting build outputs might be incorrect.

Docker containers (<https://www.docker.com/what-container>) can be used to ensure that all build workers have exactly the same system files, including toolchain and system libraries. Alternatively, you can check in a custom toolchain to your code repository and teach Bazel to use it, ensuring all users have the same files. The latter approach

comes with a penalty, however. A custom toolchain usually contains thousands of files such as the compiler, linker, libraries, and many header files. All of them will be declared as inputs to every C and C++ action. Digesting thousands of files for every compilation action will be computationally expensive. Even though Bazel caches file digests, it is not yet smart enough to cache the Merkle tree digest of a set of files. The consequence is that Bazel will combine thousands of digests for each compilation action, which adds considerable latency.

Nonreproducible build actions should be tagged accordingly to avoid being cached. This is useful, for example, to put a timestamp on a binary, an action that should not be cached. The following `genrule` example shows how the `tags` attribute is used to control caching behavior. It can also be used to control sandboxing and to disable remote execution.

```
genrule(  
    name = "timestamp",  
    srcs = [],  
    outs = ["date.txt"],  
    cmd = "date > date.txt",  
    tags = ["no-cache"],  
)
```

Sometimes a single user can write erroneous data to the remote cache and cause build errors for everyone. You can limit Bazel to read-only access to the remote cache by using the flag shown in the next example. The remote cache should be written only by managed machines such as the build workers from a continuous integration system.

```
build --remote_upload_local_results=false
```

A common cause of cache miss is an environment variable such as `TMPDIR`. Bazel provides a feature to standardize environment variables such as `PATH` for running actions. The next example shows how `.bazelrc` enables this feature:

```
build --experimental_strict_action_env
```

FUTURE IMPROVEMENTS

With just a few changes, the remote cache feature in Bazel will become even more adept at boosting performance and reducing the time necessary to complete a build.

Optimizing the remote cache

When there is a cache hit after looking up the remote cache using the digest computed for an action, Bazel always downloads all the output files. This is true for all the intermediate outputs in a fully cached build. For a build that has many intermediate actions this results in a considerable amount of time and bandwidth spent on downloading.

A future improvement would be to skip downloading unnecessary action outputs. The result of successfully looking up the action cache would contain the list of output files and their corresponding content digests. This list of content digests can be used to compute the digests to look up the dependent actions. Files would be downloaded only if they are the final build artifacts or are needed to execute an action locally. This change should help reduce bandwidth and improve performance for clients with weak network connections.

Even with this optimization, the scheme still requires many network round trips to look up the action cache for

every action. For a large build graph, network latency will become the major factor of the critical path.

Buck has developed a technique to overcome this issue (https://buckbuild.com/concept/what_makes_buck_so_fast.html). Instead of using the content digests of input files to compute a digest for each action, it uses the action digests from the corresponding dependency actions. If a dependency action outputs multiple files, each can be uniquely identified by combining the action digest from its generating action and the path of the output file. This mechanism needs only the content digests of the source files and the action dependency graph to compute every action digest in the entire graph. The remote cache service can be queried in bulk, saving the network round trips.

The disadvantage of this scheme is that a change in a single source file—even a trivial one such as changing the code comments—will invalidate the cache for all dependents. A potential solution is to index the action cache with the action digests computed using both methods.

Another shortcoming in the implementation of remote cache in Bazel is the repeated computation of the Merkle tree digest of the input files. The content digests of all the source files and intermediate action outputs are already cached in memory, but the Merkle tree digest for a set of input files is not. This cost becomes evident when each action consumes a large number of input files, which is common for compilation using a custom toolchain for Java or C and C++. Such build actions have large portions of the input files coming from the toolchain and will benefit if parts of the Merkle tree are cached and reused.

Local disk cache

There is ongoing development work to use the file system to store objects for the action cache and the CAS. While Bazel already uses a disk cache for incremental builds, this additional cache stores all build outputs ever produced and allows sharing between workspaces.

Related articles

- ➡ Borg, Omega, and Kubernetes
Lessons learned from three container-management systems over a decade
Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes
<https://queue.acm.org/detail.cfm?id=2898444>
- ➡ Nonblocking Algorithms and Scalable Multicore Programming
Exploring some alternatives to lock-based synchronization
Samy Al Bahra
<https://queue.acm.org/detail.cfm?id=2492433>
- ➡ Unlocking Concurrency
Multicore programming with transactional memory
Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha
<https://queue.acm.org/detail.cfm?id=1189288>

CONCLUSION

Bazel is an actively developed open-source build and test system that aims to increase productivity in software development. It has a growing number of optimizations to improve the performance of daily development tasks.

Remote cache service is a new development that significantly saves time in running builds and tests. It is particularly useful for a large code base and any size of development team.

Alpha Lam is a software engineer. His areas of interest are video technologies and build systems. Most recently he worked at Two Sigma Investments. He currently works at Google.

Copyright © 2018 held by owner/author. Publication rights licensed to ACM.