RFC 288 APPROVED: Exhaustive Search

Editor: keegan@sourcegraph.com
Status: APPROVED
Requested reviewers:
Approvals: Rob Rhyne Loïc Guychard Stefan Hengl
Team: Search

1 Background

Exhaustive search is a search which returns all results matching an expression. Sourcegraph's search is optimised for fast interactive searching, as such we have time and match limits which stop a search before it is exhaustive. Additionally, there are several sources of non-determinism which can interact with the limits leading to different subsets of results when retrying queries.

Exhaustive search is a common request from customers who want to solve security, compliance, code health and other automated use cases based on the output of a search. Additionally exhaustive search would be used by batch operations done by Sourcegraph. These include Campaigns and Code Insights.

2 Problems

2.1 Large result sets

A user can input a query which matches everything (eg regex .). This is not a reasonable query to run, but is useful to handle since it illustrates what time and match limits we will still have.

2.2 Accurate reporting of missing results

Given the above, we accept we won't always return everything. However, in that case we need accurate reporting to give confidence.

2.3 Determinism in result counts

Related to the above, we need determinism in how we report aggregate counts such that we solve concerns around compliance. If we say a token appears 100 times, if the search is rerun it should still say 100.

2.4 Cold caches

There are several slow paths which prevent searching until complete: uncloned repository, unindexed repository, cold searcher cache, slow backends (commit/diff). This conflicts with user expectations on exhaustive search.

2.5 Overloading Sourcegraph

Exhaustive searches are slower so can overload the system in a few ways. The concerns here are:

Noisy neighbour
The resources used during search (CPU, Mem, Network, Semaphores/Locks) will impact interactive searches latency.
OOM
Large result sets can both OOM any service from the client to the core search backend services.

A subtle problem above is our use of coarse semaphores in several places. For example we have a global semaphores to rate limit all searches. Another is a coarse read lock in zoekt which when held prevents updating the list of shards to search.

3 Proposal

Our proposal is several smaller recommendations and changes which work together to achieve exhaustive searches:

- Exhaustive searches are done via the streaming API.
- Exhaustive searches have a very large match limit.
- Introduce a display limit (vs match limit).
- Overhaul and polish of how limits and counts work. #18297
- Increase timeout for searcher fetching archives (currently 500ms).
- Allow users to override max unindexed repositories searched (searcher, diff/commit).
- Introduce separate semaphores for interactive vs batch use cases.

We introduced a display vs match limit so that clients which care about statistics can avoid the need to fetch all results. Both the webapp (small number of results) and Code Insights (no results) would use this. src-cli will be used for displaying full and large result sets.

The streaming API has not been finalised yet, so we will ask existing uses of the API by users to use src-cli instead. For internal uses (code insights, campaigns) they can use our streaming API since we can update them.

Briefly we solve the problems listed above:

Large result sets
streaming avoids us needing to hold large result sets into memory.
Accurate reporting of missing results
this is part of the work done in streaming.
Determinism in result counts
display vs match limit will increase the accuracy in the default case. Additionally we have lots of changes proposed by #18297.
Cold caches
uncloned and unindexed repos are reported as such. Large systems converge uncloned and unindexed to 0. Unindexed searches will have significantly higher limits.
Overloading Sourcegraph
streaming avoids us holding large result sets in memory. Separate semaphores reserves capacity for interactive use cases.

3.1 Tasks

See Search team :: exhaustive project board.

3.1.1 Introduce a display limit vs match limit

Right now we return and display all matches we find. The purpose of a display limit is to only return the display limit results. This can then be used to allow search to continue to look for results to update statistics.

An example is Code Insights only wants statistics. By setting the display limit to 0, it will still receive the limit. The webapp can also use this to give more accurate statistics without unnecessary network IO and JSON parsing.

Prior art for this is Zoekt's search options MaxDocDisplayCount and TotalMaxMatchCount. We will take a similiar approach and have display live outside of the query language. I propose the HTTP API layer takes in a display query parameter. Initially the webapp can hardcode this at a number higher than any user will scroll (eg 500). If a user does reach the bottom of the page we can direct them to the src-cli.

3.1.2 Differentiate between interactive and batch queries #18305

This doesn't feel like it should be part of the query language. We could make it an option we pass to the backend. However, we already have the count: parameter.

In that regards we should rather make the system gracefully handle queries which have a large count:. The UX can guide the user to using large counts, but from a backend perspective we shouldn't change behaviour.

We propose introducing a naive scheduler which:

- Co-operative to simplify implementation.
- Chunks up work by time.
- Reserves capacity for new requests.
- Differentiates requests by how long they have run.

We believe we can implement a simple algorithm that achieves the above goals. We will then tune based on emperical data. It will hook into the existing semaphores we have in search. Additionally we will need to update Zoekt to co-operatively hand back control to the scheduler while running. For other search backends the semaphore will act on a per-repo level.

This part of the proposal we are the least sure is the correct approach. We will validate this with a proof of concept the week of 1st of March.

3.1.3 Tune default display, match and time limits

This has already been done for streaming. We now have a much higher default match limit (500 vs 30) and we have removed the usual time limit. Display limit will be implemented once display has been implemented.

3.1.4 Higher limit waiting to fetch archives in searcher

Currently we wait 500ms before giving up searching an archive. With streaming we can make this much higher. This will in particular improve the experience of exhaustive searches since users will less likely need to retry.

3.1.5 Remove coarse RW lock in Zoekt

This lock is used to ensure we don't close a shard while it is being searched. However, it is a single lock for all shards. This means it is possible to starve it from being locked. Starvation prevents us from searching newly indexed shards. This is likely a minor issue but calling out.

3.1.6 Streaming RPC for Zoekt #18303

This is so we don't need to do batch requests against zoekt.

3.1.7 src-cli has streaming support

We expect exhaustive use cases which need large result sets or post-processing should be done via src-cli. We don't want to publish the streaming API yet, so adding support in src-cli is a good first step.

We propose adding a src search -stream. The current src search API is intimitely tied to our graphql layer. This means we can't replace src search with streaming without breaking other flags such as -json.We propose to introduce a switch at first, then in the future move to streaming by default with src search when we are confident users would prefer it.

3.1.8 Decide on UX for exhaustive in webapp #18306

This proposal hasn't mentioned any UX changes. See #18306 for our UX changes plan.

4 Definition of success

We can concurrently start 20 slow global searches on Sourcegraph.com with:

- no OOMing.
- All completing with accurate statistics.
- Interactive queries continue to return in under a second.

5 Addendum

https://docs.google.com/document/d/1dk309wEXA34b-LF66SBOzcNBJ6jZbWXT_eeBL9TZ2Ro/edit
https://about.sourcegraph.com/handbook/communication/rfcs

5.1 Motivating issues

Below find a list of issues providing more context on the need for exhaustive search.

- Metrics or indicator for search readiness (for massive scale instances) #14316
- Confirm $CUSTOMER exhaustive search instability fixed after 3.19 upgrade customer#71
- $CUSTOMER exhaustive search customer#69
- Multiple searches required to get full result set customer#82
- $CUSTOMER missing search results (search timeouts) customer#37
- Should Sourcegraph do stable search result ordering? #3791

5.2 Paginated Search

We could instead use paginated search. The main pro of pagination it would allow customers to more easily resume failed "chunks" of work. The downside are:

- Difficult to handle uncloned/unfetched repositories.
- More complicated client.
- No clear way to break up pages when you consider monorepos. (eg if you are a monorepo org there is only one page).

This is why we are proposing using streaming. Streaming's main downside is if the request is interrupted for some reason you don't have a way to resume without starting all over again. We believe an exhaustive search should still be in the range of minutes to run, so ensuring our backend is reliable and the client has a good connection is a reasonable trade-off. We also end up with the same API (streaming) we expect all searches to go through in the future.

5.3 Slowloris

An attack against a system is for the client to consume results very slowly tieing up locks. The streaming architecture should handle this better than batch based ones. If a client is slow it should apply back-pressure causing that request to stop consuming resources. To properly handle this we will need to make the request hand over control back to the scheduler with back-pressure.

5.4 Tasks [9/9]

- [X] Reach out about overlap with code insights.
- [X] Reach out about overlap with campaigns.
- [X] How do we differentiate between interactive vs batch?
- [X] Should we care about different limits. EG total results, per repository, per document, per line?
- [X] Cover paginated search
- [X] Cover exhausting/overloading backend system
- [X] Cover slowloris type attacks
- [X] Cover src-cli vs web
- [X] comprehensive searches

5.5 Internal use cases

5.5.1 Campaigns

https://sourcegraph.slack.com/archives/CMMTWQQ49/p1610463124037200

> We always want all of the results, except when the user manually says “count:10” or something. See here for our query: https://github.com/sourcegraph/src-cli/blob/d6eb9d2c4c99ec55da96a61dbbbfa8a7b2126b5d/internal/campaigns/service.go#L556-L579

5.5.2 Code Insights

https://sourcegraph.slack.com/archives/C014ZCKMCAV/p1610463076239900

> There certainly is (a need for exhaustive search)! Code insights run search queries through the GraphQL API and use the result count field for the charts.

Rob's notes:

At customer represented in customer#82, one of their exhaustive searches was a token search. In this search the expected and desired result count will be 0. If the token is found, it may only be present in a few files out of even a very large number of repositories. From my experience with network security search, I can confidently say this is the case with many security related searches. We had rules running constantly that returned no results.

I'm not sure there is any action or impact on that RFC, but I thought I would note that exhaustive does not always mean we would return too many results to be viewable in the UI.