Google

Superroot

Twiddler Quick Start Guide

! Review outdated · Reviewed by smunteanu on 2017-11-30 · Updated 2018-01-26

go/twiddler-quickstart

Introduction

The twiddler framework is the part of Superroot (http://go/sr) responsible for re-ranking of results from a single corpus. (The other major ranking component in Superroot is the universal packer, which combines results from multiple corpora, i.e., for universal search.)

This article is a summary of what the twiddler framework can do and of guidelines to using its various features. It is by no means an exhaustive discussion, but rather a starting point for further study. If you haven't used twiddlers before, we hope that this will give you enough to know whether you should look deeper: there are also links to larger documents and wikis scattered around this article and we are happy to chat. If you've done ranking since before there were twiddlers, we hope you'll still find some interesting perspective. In any case, please check the Twiddler YAQS queue if you have any questions.

A twiddler is a C++ object that makes ranking recommendations (twiddles) given a provisional search response from a single corpus. Twiddling differs from Ascorer ranking in that twiddlers act on a ranked sequence of results, rather than results in isolation.

There are two supported types of twiddler: predoc and lazy. Predoc twiddlers run on thin responses, which typically have several hundred results that don't contain any docinfo (snippets and other data). These twiddlers run over the full set of results returned from the backend.

After all predoc twiddlers have run, the framework reorders the thin results. It then makes an RPC that fetches docinfo for a prefix of the results, runs lazy twiddlers on that prefix, and attempts to pack a response. This attempt can fail if, for example, lazy twiddlers filter results from the top or push them down the ranking. In that case the framework fetches more docinfo, lazy twiddles the new results, and tries packing again.

More information about this packing flow can be found in the SuperrootBasicIntro.

Goals and design principles

- *Isolation:* In contrast to Ascorer, which has relatively few but complex algorithms developed over longer periods, the twiddler framework supports hundreds of twiddlers (>65 are currently active in production in WebMixer alone), each trying to optimize for certain signals. Under these conditions, letting each of these components depend on the behavior of the others would result in unmanageable complexity. Therefore, the twiddler framework conceptual model is of twiddlers in isolation (without knowledge of the others' decisions).
- *Interaction resolution:* Because twiddlers run in isolation, they can only provide constraints and recommendations of how to change the ranking. The framework then reconciles these constraints.
- *Provide context:* The framework provides safe read-only access to the context in which results are being twiddled.
- *Hide the complexities of docinfo fetching and pagination:* By constraining the operations that lazy twiddlers may perform, the twiddler framework prevents a broad range of pagination bugs—skipped or duplicated results across search result page boundaries. This is covered more in the SuperrootBasicIntro.
- *Ease of experimentation:* Because they run within Superroot it is often easier to run ranking experiments by writing a twiddler (you only need to bring up a few superroot jobs rather than building an Ascorer section or attachment, or bringing up 1,400 jobs). On the other side, if you need huge amounts of data, Ascorer is a better choice.

Writing a twiddler

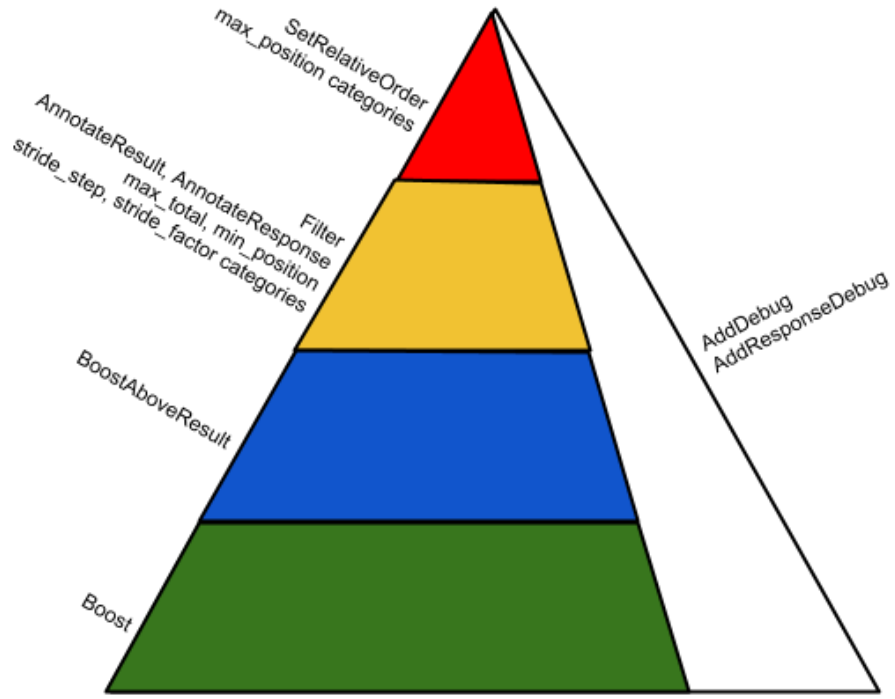
To create a new twiddler, simply derive from the base `Twiddler` class in `quality/twiddler/twiddler.h` and override the `Apply` method:

```
Twiddler::Apply(TwiddlerAPI* api, Rank start, Rank end, int debu
                Closure* done)
```

- `api`: Twiddlers call methods on this object to perform their re-ranking operations. These methods are the focus of the majority of the rest of this article.
- `start`, `end`: The range of results that the twiddler is allowed act upon, though it can examine the state of all results.
- `debug`: The level of debugging requested. For a full discussion of debugging see Debugging Options for Superroot.
- `done`: Twiddlers must always call this closure when they are done.

There are a few other minor implementation details, such as registering the twiddler for construction, but they don't require much explanation.

So many flavors of twiddling. Which one do I use?

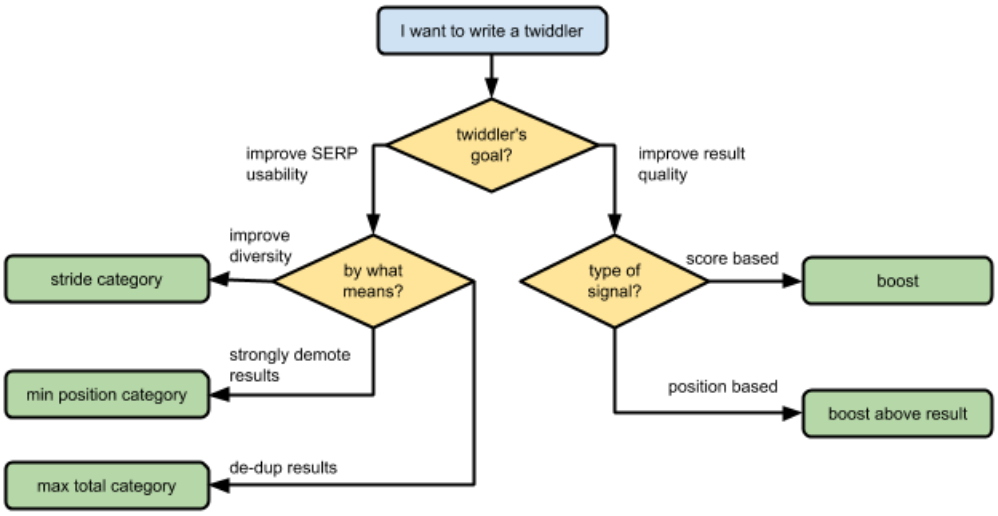


We will discuss the various `TwiddlerAPI` methods in more detail later, but first some advice: You don't need to understand all of twiddler.h to use the twiddler framework; in fact, a large fraction of the methods there have very specialized uses that only one or two projects need. Here is a way to group the `TwiddlerAPI` operations that we find to be a useful mind map:

- `Boost` and `BoostAboveResult`: The bread-and-butter APIs and what most ranking work uses. Almost all BU and most PQ wins from twiddlers come exclusively from these two methods.
- `Filter`, `max_total`, and `stride` categories: Used to increase diversity, remove duplicates, reduce unwanted results such as spam and foreign pages.
- `AnnotateResult`, `AnnotateResponse`: Don't change ranking directly; used to communicate with GWS and other parts of Superroot.
- `SetRelativeOrder` and `max_position` categories: Useful in very special cases; tricky as they don't interact well with the rest of the framework and need special considerations. Consult with superroot-team@ if you think you need them.
- `merge_cluster` and `stride_demotion_factor` categories: Don't use them. Seriously.
- There are also accessors and methods for debug, but they don't need any special consideration at this point.

The way we generally recommend looking at the twiddler operations is to *use twiddler methods and category types to express semantic intent*, rather than focusing on the operational details of what they do and how they interact. For example, don't use `Boost` when what you want is to demote a result to the second page, and don't use `max_position` if you've only determined that a result is better than than the current top result (if you don't yet know what `Boost` or `max_position` mean, it's ok, we'll explain further down). This is the best way to ensure that your wins will stay around as new twiddlers and framework changes are introduced.

Here is another way to look at the lower parts of the pyramid, as leaves of a decision diagram:



You will also need to decide whether your twiddler should run predoc or lazily. Predoc twiddlers run on the full range of thin results, while lazy twiddlers run on monotonically increasing ranges of fat results (which have snippets and other docinfo data).

Your twiddler should run predoc if:

- It modifies result IR scores.
- It promotes results.
- It performs RPCs to services such as TwiddlerServers, SSTables, or FastMap.

Your twiddler should run lazily if:

- It requires snippets or other docinfo.
- It needs to see the outcome of predoc twiddlers actions.

IR-score modifying methods

There are several distinct flavors of reranking methods. The first flavor is delicious score boosting, which we highly recommend. These methods directly manipulate result IR scores.

`Boost`

```
Boost(Rank result, float boost)
```

Use `Boost` when you have an IR-score based signal indicating that the score of the document should be increased or decreased by a certain

factor. The call instructs the framework to multiply the IR score of the result by boost.

Example: `YoutubeDensityTwiddler` promotes channel results when multiple video results from that channel are a good match for the query.

The framework combines all Boosts by multiplying them together. `Boost` is the most widely used and oldest of the twiddler APIs. You can think about it as the analogue of a score adjuster in Ascorer. If your underlying signal is position-based, not score-based, then `BoostAboveResult` may be a better choice.

`BoostAboveResult`

```
BoostAboveResult(Rank a, Rank b, float tie_breaker)
```

Use `BoostAboveResult` when the underlying signal indicates that result A should rank above (or around the position of) another result B. The framework will compute an equivalent boost factor for you.

Example: `YoutubeMovieTwiddler` boosts movie results that a top ranking entity to position 0.

The framework resolves all `BoostAboveResult` actions made on a result by converting them to an equivalent multiplicative boost. As opposed to calculating a `Boost` manually to achieve a similar effect, `BoostAboveResult` has the following advantages:

- The framework performs score interpolation for you, including handling edge cases such as boosting to the first or last position, and multiple results at the same position (that's what the `tie_breaker` argument is for).
- When multiple twiddlers use `BoostAboveResult` to promote or demote the same result, the framework combines their actions as follows:
    - If any call to `BoostAboveResult` would cause a ranking promotion, the strongest promotion is chosen.
    - If all calls would result in a ranking demotion, the strongest demotion is chosen.

The framework is also smart about the way it combines `Boost` and `BoostAboveResult` actions to avoid double promotions or demotions.

Here's a sketch of how it works:

```
const float boost_above_result = GetResolvedBoostAboveResult(r  t)
float promotion = 1.0;
float demotion = 1.0;
for (float boost : GetAllBoosts(result)) {
  if (boost > 1.0) {
    promotion *= boost;
  } else {
    demotion *= boost;
  }
}
const float combined_boost =
    max(promotion, boost_above_result) *
    min(demotion, boost_above_result);
```

For the full details, see: Design doc | Code

A useful guideline to choose between `Boost` and `BoostAboveResult` is: If your signal could, in some possible scenario, move down to Ascorer (because it can be expressed as a function of a single result, perhaps with the addition of some aggregate information about all the top results), then you should probably use `Boost`; otherwise, you should probably use `BoostAboveResult`. Another useful guideline is that if you routinely `Boost` by a factor of more than 5–10, there's a good chance you're not doing it right.

Constraint methods

Constraint methods are piquant, and can be an acquired taste.

This large class of twiddler actions has no analogue in the one-result-at-a-time Ascorer world: they constrain result positioning and limit the number of results of a given type that are allowed in the final response. Nearly all constraints are specified via categorization: a twiddler creates a new category, with the desired constraints, and assigns one or more results to that category. Using category constraints requires two methods:

`NewCategory`

```
NewCategory(const CategoryParams& params)
```

Creates a new category with the given parameters.

`Categorize`

```
Categorize(Rank result, TwiddlerCategoryId id)
```

Assigns a result to the category with the given id.

To create a category, a twiddler must first populate a `CategoryParams` struct with the constraints it wishes to apply; all `CategoryParams` constraints are initialized with no-op values, so only the fields of interest need to be set. The twiddler must also set the id field in `CategoryParams`, so that the category can be identified in subsequent calls to `Categorize`. (Each Twiddler has its own id space, so there can be no accidental use of the same category by two different twiddlers.)

Now, the twiddler can assign results to their category via `Categorize`. The general pattern for applying a single category is as follows:

```
CategoryParams params;
params.id = 1234;
// Set desired fields in params.
api->NewCategory(params);
for (Rank rank = start; rank < end; ++rank) {
  if (ShouldApplyCategory(rank)) {
    api->Categorize(rank, 1234);
  }
}
```

Category constraints

There are many types of category constraints, some more surprising than others. Here is a selection of the ones we recommend. A detailed list of all the category constraints can be found in quality/twiddler/category_params.h, but please consult with superroot-team@ before you try out ones that aren't on this list.

`max_total`

```
max_total = N
```

Prevents more than N results in a category being packed.

Example: `BlogCategorizer` places all the results from a blog in a `max_total` category to prevent too many being shown. It also provides an escape hatch via annotations, a framework feature we'll talk about later. `max_total` constraints are enforced when results are being packed into the final response: any `max_total` constraints applied by predoc twiddlers is simply propagated through to final response packing.

`predoc_limit`

```
predoc_limit = N
```

Filters all but the first N results in a category at the end of predoc twiddling.

Let's look at a pathological case of applying a `max_total` constraint of 1 to the first 100 results. Since `max_total` constraints are enforced during final response packing, the twiddler framework will end up having to fetch docinfo for all 100 results, even though only 1 can be packed. To guard against this, a twiddler can add a `predoc_limit` constraint to a `max_total` category.

Unlike most other category constraints, `predoc_limits` are enforced after the results have been sorted at the end of predoc twiddling: the first N results are kept and the rest are filtered, so the framework will not fetch docinfo for them. The `predoc_limit` should chosen to be somewhat larger than the corresponding `max_total` constraint, to allow some slack for some results being filtered or reordered by other twiddlers.

`min_position`

```
min_position = N
```

Prevents results being packed earlier than the Nth rank (rank 0 is the first result, rank 1 is the second, etc., so that `min_position = 20` means below the 20th result).

Example: `BadURLsCategorizer` adds a pseudo random (but deterministic per-query) `min_position` constraint to results that are marked for demotion, pushing them off the first couple of pages.

`stride_step` and `stride_factor`

```
stride_step = X
stride_factor = Y
```

Require a certain minimum spacing between consecutive results of the same category.

The second result from the category is packed no less than X+Y spaces after the first, the third no less than X+2Y after the second, the fourth no less than X+3Y after the third, and so on.

Example: `ImageHostCategorizer` uses stride constraints to prevent too many images from the same host being clustered together.

`max_position`

```
max_position = N
```

Prevents results from being packed later than the Nth rank.

Example: `OfficialPageTwiddler` applies a `max_position` constraint of 0 to the official page relating to a query when it has very high confidence in its "officialness" signal.

`max_position` constraints should be used with care, since they will override any demotions that another twiddler may wish to apply. Since `max_position` constraints can promote results, they may only be applied by predoc twiddlers, otherwise we would run into pagination bugs.

Intermezzo: The category packing algorithm, stress, and priorities

Constraints are enforced by the category packer (not to be confused with the universal packer), the algorithm at the last stage of twiddling that computes the final response. The number of results packed depends on the user request:

- `&num=10`: pack 10 results.
- `&num=50`: pack 50 results.
- `&start=20&num=10`: pack 30 results (the universal packer will discard the first 20).

The category packer first applies lazy twiddlers to a range of fat results; it then puts these results into a priority queue and attempts to pack them one-at-a-time into the response. The constraints are used to determine the priority of results in the queue—the basic priority being score order, then modified by the presence of `min_position` and stride categories as more results from a category are packed.

You may be wondering what happens in cases where the result set is overconstrained. The category packer maintains a concept of stress, which roughly corresponds to how many results are stuck in the pending queue waiting to be packed. As the stress increases, the category packer begins to relax constraints, allowing results to be packed. Categories can be assigned a priority value in the range [0, 1], which controls the rigidity of their constraints.

`SetRelativeOrder`

```
SetRelativeOrder(Rank a, Rank b)
```

Use the `SetRelativeOrder` method to specify that result A must be packed above B if B appears in the packed response.

Example: `YoutubeDuplicatesRemovalTwiddler` uses `SetRelativeOrder` to combat the problem of people uploading multiple copies of the same video. It tries to identify the original video, and reorders the results such that the original video ranks the highest of all the duplicates.

`SetRelativeOrder` is another packing constraint, although it is a `TwiddlerAPI` method, not a category type. Superficially similar to `BoostAboveResult`, the constraint is actually much stronger and is enforced during category packing, overriding other twiddlers' requests, including `max_position`. It should be reserved for special circumstances only. Since `SetRelativeOrder` can promote results, it may only be called by predoc twiddlers, otherwise we would run into pagination bugs.

Result filtering methods

Filtering methods allow twiddlers to separate the wheat from the chaff.

`Filter`

```
Filter(Rank result)
```

`Filter` logically removes a result from the response.

Example: `EmptySnippetFilter` filters results that have no snippet. If your twiddler is attempting to perform de-duping operations, using a `max_total` category constraint is normally a better choice. `Filter` doesn't immediately remove results from the response, they are simply marked as filtered.

`Hide`

```
Hide(Rank result, const MessageSet& annotation)
```

Hide is a specialized method mostly used to implement legal removals.

Example: `DMCAFilter` hides results for which Google has received and reviewed DMCA notices, and adds annotations later used by GWS to show ChillingEffects links.

If we are obligated by law to censor a result that would otherwise have shown, we want to be able to display a warning to our users. Filtering too early would prevent the exact calculation of that event. Hidden results flow through the twiddler framework untouched and are only removed at the final stage of result packing.

`Filtered`

```
Filtered(Rank result)
```

Calling `Filtered` with a result's rank will return true if that result was filtered by a twiddler running in an earlier twiddling round.

It is recommended that lazy twiddlers check the `Filtered` state of results, to avoid processing results that were filtered during predoc twiddling. Additionally, twiddlers can use `Filtered` to see which results

they themselves have removed, though because twiddlers run concurrently, calls to `Filtered` will not reflect any `Filter` calls made by other twiddlers running in the same round.

Annotating methods

And we now come to the annotating methods, those rich with the elusive flavor-taste, umami.

These methods let twiddlers add protocol messages to the response or to specific results. They are used to pass information up the stack, for example to twiddlers running in later phases, the universal packer or to GWS, to influence later ranking or UI decisions.

`AnnotateResult`

```
AnnotateResult(Rank, const MessageSet& annotation)
```

Annotates a result with messages in the given `MessageSet`.

Example: `SocialLikesAnnotator` annotates social results with the number of +1s they have received.

`AnnotateResponse`

```
AnnotateResponse(const MessageSet& annotation)
```

Annotates the response with messages in the given `MessageSet`.

Example: `SymptomSearchTwiddler` annotates the response with possible medical conditions and symptoms.

Debug methods

There are two methods that twiddlers can use to add debug information to the result page.

`AddDebug`

```
AddDebug(Rank rank, const string& message)
```

Associates some unstructured debug data with the result at a particular rank.

`AddResponseDebug`

```
AddResponseDebug(const string& message)
```

Associates some unstructured debug data with the response.

Further topics

There are some things in the twiddler framework that we haven't mentioned at all, but we hope that this brief introduction gives you a place to start. Some of the major things we did not cover, and that you can read about by following links in this document, or asking us, are:

- **Remote calls**

  Predoc twiddlers can make RPCs to FastMap, SSTables, or to arbitrary remote code running in so-called TwiddleServers.

- **Multi-phase twiddling**

  There are in fact multiple rounds of predoc twiddling, with a serialization point after each where all score modifications are applied and results are reordered. Phases before the last are intended for twiddlers affecting large fractions of the query stream that would otherwise create too many interactions. Categorize and SetRelativeOrder are not allowed in early phases.

- **Merge cluster categories**

  Essentially used for megasitelinks. We only mention them because occasionally we've got questions from someone trying to use them to obtain some strange ranking effect. If you think you understand how they work, trust us: you don't. We're not sure that we do either.

We hope that you got a flavor for what the twiddler framework can do for you. You should now be able to guess what were the right APIs for the two examples earlier on:

How to demote a result to the second page? A `min_position` category is probably a good starting point.

How to promote a result when your signal only does pairwise comparisons and says it's better than the current top? Use `BoostAboveResult`. If your signal is very very high precision, consider `SetRelativeOrder` as well, but ask first.

For further questions, we are available at superroot-team@.

Comments

Before you can create or read g3doc comments, you need to grant g3doc access to Buganizer and Google people information. Learn more

Get access to g3doc comments

View all unresolved or resolved comments