On Precomputation and Caching in Information Retrieval Experiments with Pipeline Architectures

Sean MacAvaney[†], Craig Macdonald[†]

University of Glasgow, United Kingdom

Abstract

Modern information retrieval systems often rely on multiple components executed in a pipeline. In a research setting, this can lead to substantial redundant computations (e.g., retrieving the same query multiple times for evaluating different downstream rerankers). To overcome this, researchers take cached "result" files as inputs, which represent the output of another pipeline. However, these result files can be brittle and can cause a disconnect between the *conceptual* design of the pipeline and its *logical* implementation. To overcome both the redundancy problem (when executing complete pipelines) and the disconnect problem (when relying on intermediate result files), we describe our recent efforts to improve the caching capabilities in the open-source PyTerrier IR platform. We focus on two main directions: (1) automatic implicit caching of common pipeline prefixes when comparing systems and (2) explicit caching of operations through a new extension package, pyterrier-caching. These approaches allow for the best of both worlds: pipelines can be fully expressed end-to-end, while also avoiding redundant computations between pipelines.

Keywords

Information Retrieval Experiments, Caching

1. Introduction

Information retrieval systems are now often multi-stage architectures. In large-scale deployments, this allows separation of efforts: one team may work on the indexer or first-stage retriever architecture, while other teams may be responsible for rerankers (e.g. learning-to-rank, neural models), ad selection, answer generation or result presentation. While developed independently, it is imperative that these pipeline components successfully operate in tandem. For instance, if a reranker is not prepared to handle the distribution of results provided by the retriever, the engine's effectiveness will suffer.

When proposing a new method, it is important for researchers to demonstrate its robustness in different environments. Robustness is typically shown by evaluating the method on multiple benchmarks, but this is only one dimension of robustness. We argue that it is also important to show how well a method performs in various pipelines since this reflects the diversity of environments in which the method may be deployed.

However, the available data and tooling often make this type of experimentation challenging. For instance, some QA datasets (such as 2Wiki [1]) commonly provide retrieved documents for a single retriever, meaning that the impact of answer generation quality to different retrieval systems is not systematically ablated; precomputing the first-stage results for testing an answer generator or reranker can lead to an inflexible experimental workflow, that prevents the researcher from *dog-fooding*, i.e. testing the approach on their own queries.

A central goal of PyTerrier [2, 3][1] is to provide a shared platform that enables this kind of experimentation. Components in the platform can be composed into declarative pipelines that clearly define their constituent components, called Transformers[2], and how they interact. This design also allows individual components to be easily ablated (e.g., swapping out one retriever for another). However, this

Second International Workshop on Open Web Search (WOWS 2025)

[†] Listed alphabetically. These authors contributed equally.

sean.macavaney@glasgow.ac.uk (S. MacAvaney); craig.macdonald@glasgow.ac.uk (C. Macdonald)

0000-0002-8914-2659 (S. MacAvaney); 0000-0003-3143-279X (C. Macdonald)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1] http://github.com/terrier-org/pyterrier

[2] We note that the term *tranformer* is overloaded, also referring to a neural network architecture, an electrical component, etc.

approach can incur substantial redundant costs at experimentation time. For instance, when comparing two answer generators under the same retrieve-and-rerank pipeline, the preceding pipeline is fully executed twice (even though the results are the same). Similarly, if the researcher seeks to ablate the retriever choice within the same pipeline, the reranker will redundantly score any document retrieved by both systems twice, even though they are assigned the same score.

This paper presents our recent efforts to address these problems while retaining the platform's flexibility. We apply two complementary approaches. The first automatically detects common prefixes in pipelines when running an experiment and only executes the common prefix once (Section 3). The second approach is to let users explicitly define a caching strategy over individual components (Section 4). Together, approaches will help reduce the computational overhead of conducting experiments, abiding by the "reuse" principle of GreenIR [4].

Naturally, our approaches are not the only ways to perform caching for information retrieval research.[3] However, we feel that our approaches hit a sweet spot between functionality and ease-of-use for day-to-day experimentation. The traditional file-based workflow involves saving intermediate results to files (e.g., TREC-formatted "result" or "run" files). These result files can be used as starting points (and are hosted in places like ranxhub [5] or the TREC "Past TREC Results" page[4]), but result files on their own do not clearly define their provenance[5] and are relatively brittle (e.g., one could mistakenly use a run file that operated over a topic's "title" field when the remainder of the pipeline runs over its "description"). Meanwhile, the TIREx platform [7, 8] performs community-wide pipeline prefix caching, but requires containerising all components (which is a benefit in terms of reproducibility, but reduces flexibility). Finally, we note that there is a significant body of literature on the caching of search engine posting lists [9, 10] or results [11, 12], however, to the best of our knowledge, all existing work is concerned with caching in deployed information retrieval systems, rather than for the purposes of aiding experimentation.

In the remainder of this paper, we provide a background on PyTerrier (Section 2), followed by a description of our precomputation method for IR experiments (Section 3), explicit caching transformers (Section 4). Section 5 provides some demonstration experiments of the benefits of the precomputation and caching to an exemplar experiment; Section 6 discusses limitations and future work, while concluding remarks follow in Section 7.

2. Information Retrieval Pipelines and Experiments in PyTerrier

We provide an overview of the PyTerrier data model and operator language (Section 2.1), as well as the manner in which evaluation is conducted in PyTerrier (Section 2.2). Together these provide these necessary background for our understanding the functional aspects of our solutions described in Sections 3 and 4.

2.1. Declarative Pipelines

We now provide a short summary of the PyTerrier [2] data model, the families of transformers, and operators for their combination. Firstly, let $Q(qid, query)$ be a relation type for a set of queries, and similarly $D(docno, text, ...)$ a relation type for a set of documents, with additional possible attributes. From these principle types, we can derive types such as for (i) documents ranked in response to a query $R(qid, docno, score, rank, ...)$; (ii) relevance assessments $RA(qid, docno, label)$. All relation types are extensible, in that extra columns can be added. In PyTerrier, these relation types can be instantiated as Pandas DataFrames, or as lists of dictionaries, both with the required and any optional attributes. Indeed, the exact choice of instantiation is left to the preferences of the developer of a particular class, and PyTerrier maps between the DataFrames and lists as needed.

[3] In fact, PyTerrier previously had a general-purpose caching operator ($\sim$). This operator is now deprecated in favor of these more robust and flexible caching approaches.

[4] https://trec.nist.gov/results.html

[5] We note that efforts have been undertaken to improve the provenance of result files, however [6].

Transformations between these relation types are called *transformers*: for instance a transformer class fulfilling a retrieval role should expect data of type Q, and return type R. Typical families of pipeline stages can be expressed as mappings between these relations, for instance:

- *Retrieval*: $Q \rightarrow R$
- *Reranking*: $R \rightarrow R$
- *Query rewriting*: $Q \rightarrow Q$
- *Document rewriting*: $D \rightarrow D$
- *Pseudo-relevance feedback*: $R \rightarrow Q$
- *Indexing*: $D \rightarrow \emptyset$ (a terminal operation)

Each transformer object, t, operates as a function, i.e. we can obtain a set of results for a given input by invoking t on that input, $t(input)$.

As argued above, IR systems are more commonly phrased as pipelines. To combine different transformers, PyTerrier offers a number of operators defined on transformers, that allow them to be succinctly expressed. For instance $\gg$ is known as "then" or "compose", and is defined as:

$$(t_1 \gg t_2)(input) := t2(t1(input)) \tag{1}$$

This allows many pipelines of transformers to be created, and easily understood. Indeed, rather than an imperative programming style, where a series of steps may be executed for a given set of data in sequence, here the entire pipeline is constructed before execution on a set of queries. The $\gg$ notation has been seen to be easily understandable and has been appearing as notation in many IR papers in recent years. Table 1 summarises all of the operators defined in PyTerrier. Each operator has relational algebra semantics, as explored in Macdonald and Tonellotto [2].

Of note from Table 1, $\gg$ and $\%$ (rank cutoff) are the most commonly used operators. For example, we might want to apply rank cutoffs before different reranking stages:

```
pipe = bm25 % 100 >> MonoT5() % 10 >> DuoT5()
```

Within Table 1, $\sim$ is a caching operator, which is used to cache the results of a retrieving transformer to disk. This operates such that when a query is executed repeatedly, the results for that query can be retrieved directly from the cache, in order to speed up retrieval, particularly for experiments where a first-stage retrieval may be invoked repeatedly for the same query(ies). In brief, this is the only operator we have been unhappy with, and it is now deprecated for removal in a future release of PyTerrier. The semantics of how the caching operator is currently defined does not offer sufficiently fine-grained control - in particular, the type of caching that is most appropriate likely varies according to the family of transformer, which is not immediately obvious to the cache implementation. For example, a cache for a retriever should cache on the qid attribute (the primary key of Q datatype), and return the score for that

Table 1
PyTerrier operators for combining transformers.

Op.	Name	Description
>>	*then*	Pass the output from one transformer to the next transformer
%	*rank cutoff*	Shorten a retrieved results list to the first K elements
+	*linear combine*	Sum the query-document scores of the two retrieved results lists
*	*scalar product*	Multiply the query-document scores of a retrieved results list by a scalar
**	*feature union*	Combine two retrieved results lists as features
\| | *set union* | Make the set union of documents from the two retrieved results lists |
& | *set intersection* | Make the set intersection of the two retrieved results lists |
^ | *concatenate* | Add the retrieved results list from one transformer to the bottom of the other |
$\sim$ | *cache* | Keep the results of this transformer on disk [now deprecated] |

query. However, caching on only the qid would result in a rewritten query retrieving the same results as the original query, so $\langle qid, query \rangle$ would be a safer cache key. On the other hand, the cache for a cross-encoder reranker should cache on at least $\langle qid, docno \rangle$, and, more safely $\langle qid, docno, query, text \rangle$, (in order to rescore when the query or text have been expanded). More generically, the primary keys and the functional dependencies of the input and output types of a transformer are not exposed by a given transformer, and would be required for caching to operate properly. Such fine-grained control over caching cannot be achieved using only a unary caching operator to signify a cached transformer (indeed, the current caching operator assumes that all cached transformers are retrievers). For this reason, this paper discusses alternative caching strategies, addressing common experimental use cases easily, while also offering fine-grained control over caching behaviour.

2.2. Declarative Experiments

PyTerrier also defines an experimentation abstraction for the purposes of evaluating different retrieval systems (including composed pipelines). In particular, the `pt.Experiment()` function takes four key arguments: (i) a list of systems to be compared; (ii) the set of queries on which they should be evaluated (type Q); (iii) the set of relevance assessments to use to evaluate them (type RA); and (iv) the evaluation measures to compute. This invokes each system on the specified queries, and applies the specified evaluation measures on the outcome. An example experiment is shown below, which would evaluate the impact on the number of documents retrieved by BM25 on the nDCG@10 effectiveness of a pipeline involving MonoT5 and DuoT5 (recall that `%` denotes application of a rank cutoff):

```
pt.Experiment(
   [bm25 % k >> MonoT5() % 10 >> DuoT5() for k in [20, 50, 100, 200]],
   dataset.get_topics('test'),
   dataset.get_qrels('test'),
   [nDCG@10]
)
```

The succinctness of the experimentation abstraction demonstrates its utility for researchers - indeed, we place considerable focus on developing memorable APIs that don't require researchers to regularly refer to documentation. Its also more succinct than an imperative workflow - its a single statement - no for loops, no different invocations for different stages of a ranking pipeline. Additional options for `pt.Experiment()` allow the calculation of significance tests wrt. a baseline, application of multiple-testing correction (as recommended by Fuhr [13], Sakai [14]), limiting batch sizes, etc.

However, the example above also illustrates one of the challenges with a declarative workflow, in that the BM25 retriever would be invoked for the topic set on each of the 4 pipelines. This is a considerable efficiency disadvantage compared to an imperative workflow, whereby a user may gather all BM25 results, before applying the cutoff. An alternative formulation is shown below - here BM25 is applied on the topics before ingestion into the experiment; an Identity transformer[6] is used in place of BM25 to allow the results to be passed-through to a rank cutoff:

```
pt.Experiment(
   [pt.Transformer.identity() % k >> MonoT5() % 10 >> DuoT5() for k in [20, 50, 100,
       200]],
   bm25(dataset.get_topics('test')), # <--- BM25 results as input, rather than queries
   dataset.get_qrels('test'),
   [nDCG@10]
)
```

So while this is explicit, it somehow feels less appropriate - a reader of the code would not be naturally drawn to line 3 where `bm25` is invoked on the test queries; the clear separation between queries and systems has been lost. The use of the Identity transformer also feels unnatural, and likely to confuse readers.

Instead, in the next section, we discuss how a different strategy whereby the same experiment is efficiently conducted, i.e. without repetitive invocations of BM25.

[6]The Identity transformer just returns its input, unchanged.

Figure 1: A visual depiction of the prefix precomputation approach for two pipelines: A » B and A » C. With prefix precomputation, the common A prefix is identified and the results are used for the computation of the remainder of both pipelines, i.e. B & C.

3. Prefix Precomputation in Comparative Experiments

PyTerrier's operator-based language for expressing pipelines can be seen as closer to the conceptual design that one might write in a paper.[7] However, the logical implementation may differ. For example, consider a transformer $Retriever(index, k)$ that has a rank cutoff operation ($\%k'$) applied. A more efficient pipeline formulation would be to apply the rank cutoff directly in the Retriever instance.[8] PyTerrier supports a number of such optional *compile* operations, which allow apply a rewriting of the conceptual pipeline into a more efficient logical variant [2] - i.e. a syntactically different but semantically equivalent reformulation of a pipeline that executes more quickly.

In this vein, let us consider an experiment comparing rerankers (B and C) applied to the results of a retriever A. This would be instantiated as an experiment involving two pipelines $A \gg B$ and $A \gg C$. When performing a side-by-side evaluation of these pipelines, as mentioned in Section 2.2, there may be efficiency gains in pre-computing the results of A.

To this end, PyTerrier now supports *prefix precomputation* when conducting experiments: here, any common prefix of all the evaluated pipelines is invoked once, and the results applied on the remainder of the pipelines. This is shown visually in Figure 1. To expose this functionality to the researcher, we simply add an optional argument to `pt.Experiment()`, namely `precompute_prefix`, as shown below:

```
pt.Experiment(
   [bm25 % k >> MonoT5() % 10 >> DuoT5() for k in [20, 50, 100, 200]],
   dataset.get_topics('test'),
   dataset.get_qrels('test'),
   [nDCG@10],
   precompute_prefix=True # <---- enable precomputation
)
```

Our current implementation identifies the longest common prefix (LCP)[9] of a set of pipelines - efficient implementations of this algorithm can be instantiated that only assume that transformers have an equality property (i.e. we can test to see if two transformers are equal).

Formally, let P be a set of transformer pipelines for a given experiment, where each pipeline $p_i \in P$ consists of different stages of a pipeline $t_{i,1} \gg \ldots \gg t_{i,||p_i||}$ where the number of stages in a pipeline is denoted by $||\cdot||$. Further, let $p[j]$ denote the jth stage of a pipeline, and $p[j..k]$ denote a range of transformer stages. In applying the LCP algorithm, we identify a common prefix $\text{LCP}(P)$, as follows:

$$\text{LCP}(P) = \arg\max_{cp} \{||cp|| \text{ s.t. } cp[j] == p_i[j]\ \forall i, 1..j\} \tag{2}$$

Using $LCP(P)$, we can identify the *remainder* pipelines $\hat{P}$, where each constituent $\hat{p}_i$ is the remainder of the pipeline p starting after the common prefix, i.e. $\hat{p}_i = p_i[||LCP(P)||..||p_i||]$. Then, for a given set of queries, q, evaluation naturally takes place by obtaining the results on the common prefix, i.e. $interim\_res = \text{LCP}(P)(q)$, followed by evaluation of each pipeline remainder, i.e. $\hat{p}_i(interim\_res)$.

The astute reader may observe that there are possible experiments of $M > 2$ pipelines, where a common prefix is shared by only a subset, $2..M-1$. Our current implementation would not benefit

[7] Indeed, we've noted an increasing number of papers using the $\gg$ notation to indicate composition of pipeline stages.
[8] This is akin to a SQL query optimisation, where selection operations are moved earlier.
[9] It has to be prefix, rather than the more general longest common subsequence, as pipelines are affected by their leftmost constituent transformation.

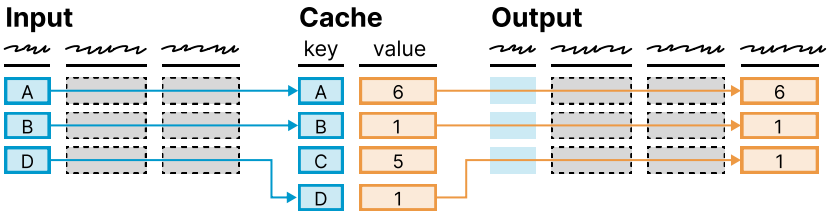

Figure 2: The `KeyValueCache` maps each key (consisting of one or more input column) to a value (one or more output columns). It assumes rows are treated independently and that the values only depend on the keys.

the efficiency of such experiments. We postulate that this may be addressed by counting the coverage of all possible pipeline prefixes, but leave this further development to future work.

That said, our experience with prefix precomputation thus far has been positive - the functionality works as expected, with no unexpected corner cases that required to be resolved. As a result, we may we set this to be default setting for `pt.Experiment()` in the future. Overall, we argue that the simplicity of prefix precomputation addresses a key efficiency disadvantage of the declarative workflow, while enabling end-to-end evaluation and retaining legibility of the experiments. It can also be seen as a marked progress between separation of a conceptual model of an IR experiment and its logical implementation. We are not aware of any previous work considering the programmatic decomposition of IR pipelines in this manner to benefit experimentation.

For more fine-grained control, in the next section we discuss a different type of caching, where the researcher wishes to retain full control over what is reused between pipelines or pipeline invocations.

4. Explicit Caching Strategies

We now describe the explicit caching strategies provided by the pyterrier-caching[10] package. Four strategies are provided to cover a variety of use cases: key-value caching (Section 4.1); caching for scorers/rerankers (Section 4.2); caching for retrievers (Section 4.3), and caching for indexing operations (Section 4.4).

4.1. `KeyValueCache`: Key-Value Caching

The `KeyValueCache` is a basic caching strategy that maps one or more "key" columns to one or more "value" columns. The cache operates on a row-by-row basis, working under the assumption that each row does not affect the results of other rows. A visual depiction of the cache is given in Figure 2. This formulation makes the `KeyValueCache` suitable for operations like Query Rewriting ($Q \rightarrow Q$) and Document Rewriting ($D \rightarrow D$). For example, a Doc2Query [15] transformer can be cached as follows:

```
from pyterrier_caching import KeyValueCache
from pyterrier_doc2query import Doc2Query
dataset = pt.get_dataset('irds:msmarco-passage') # some dataset
model = Doc2Query(append=True)
cache = KeyValueCache('doc2query.cache', model, key='text', value='querygen')

# First index with Terrier: (fills cache with Doc2Query results)
index = pt.terrier.TerrierIndex('doc2query.terrier')
pipeline = cache >> index.indexer()
pipeline.index(dataset.get_corpus_iter())

# Indexing with PISA is now faster, since Doc2Query results are cached
from pyterrier_pisa import PisaIndex
index = PisaIndex('doc2query.pisa')
pipeline = cache >> index.indexer()
pipeline.index(dataset.get_corpus_iter())
```

[10] https://github.com/terrierteam/pyterrier-caching

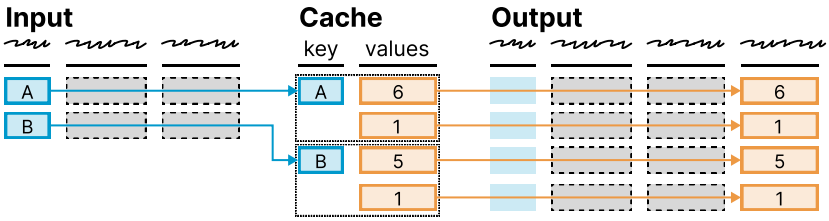

Figure 3: The `RetrieverCache` maps each key (consisting of one or more input column) to a value (many rows over one or more output columns). It assumes input rows are treated independently and that the values only depend on the keys.

Implementation Details. `KeyValueCache` is implemented as a SQLite database. Keys and values are encoded as blobs using Python's built-in `pickle` package. Rows with key misses in the database are passed along to the wrapped transformer to calculate their corresponding values, which are then inserted into the database.

4.2. `ScorerCache`: Caching Scorer Results

Typical scorers (rerankers) operate by independently assigning a new relevance score for each document under the probability ranking principle [16].[11] Given the prominence of this pattern and the need to re-assign the *rank* column based on these new scores, the `ScorerCache` implements this special case of the general-purpose `KeyValueCache`. The *query* and *docno* columns serve as the key, and the *score* column serves as the value, though this functionality can be overridden. An example demonstrating the value of the `ScorerCache` when caching the results of a MonoElectra [20] model follows:

```
from pyterrier_dr import ElectraScorer
from pyterrier_pisa import PisaIndex
dataset = pt.get_dataset('irds:msmarco-passage/dev/small')
index = PisaIndex.from_hf('macavaney/msmarco-passage.pisa')
scorer = dataset.text_loader() >> ElectraScorer()
cached_scorer = ScorerCache('electra.cache', scorer)

# Use the ScorerCache cache object just as you would a scorer
cached_pipeline = index.bm25() >> cached_scorer
cached_pipeline(dataset.get_topics())

cached_pipeline(dataset.get_topics()) # <-- all values are cached

# Will only compute scores for docnos that were not returned by bm25
another_cached_pipeline = index.qld() >> cached_scorer
another_cached_pipeline(dataset.get_topics())
```

Implementation Details. By default, `ScorerCache` uses the same SQLite strategy as `KeyValueCache`, while also re-assigning the *rank* column based on the new scores. In some cases, a large proportion of a corpus is scored – for instance, when exploring the effect of an exhaustive search strategy of cross-encoders [21]. In this case, a SQLite cache is inefficient because of the high repetition of document identifiers and other overheads. Therefore an alternative implementation (`DenseScorerCache`) is available. `DenseScorerCache` uses a HDF5 backend, with a separate npids[12] file serving as a mapping between the docnos and their corresponding indexes in the storage array.

4.3. `RetrieverCache`: Caching Retriever Results

Other operations — most notably retrievers — map each input row to multiple output rows. The `RetrieverCache` (shown visually in Figure 3) handles caching in this situation. The following code snippet shows how this cache can save on repetitive calls to a retriever:

```
from pyterrier_caching import RetrieverCache
dataset = pt.get_dataset('irds:msmarco-passage/dev/small')
index = pt.terrier.TerrierIndex.from_hf('macavaney/msmarco-passage.terrier')
bm25_cache = RetrieverCache('path/to/cache', index.bm25())

bm25_cache(dataset.get_topics())
bm25_cache(dataset.get_topics()) # <-- all values are cached
```

Implementation Details. `RetrieverCache` is implemented using Python's `dbm` package. The `dbm` is a generic key-value database interface that is included in Python's standard library. The keys to the DBM database are SHA256 hashed pickles of the keys, and the values are LZ4-compressed pickles of the value frames.

4.4. `IndexerCache`: Caching Indexing Streams

In some cases it is beneficial to store an entire sequence of inputs. This is especially true for indexing operations, such as document encoding with Learned Sparse Retrieval [22]. The following example shows how to cache SPLADE document representations [23]. Note that unlike other caching operations, the `IndexerCache` acts as an indexer:

```
from pyt_splade import Splade
from pyterrier_caching import IndexerCache
dataset = pt.get_dataset('irds:msmarco-passage')
splade = Splade()
cache = IndexerCache('splade.cache')
cache_pipeline = splade >> cache

# The following line will save the results of splade to splade.cache
cache_pipeline.index(dataset.get_corpus_iter())

# Now you can build multiple indexes over the results of splade without
# needing to re-run it each time
indexer1 = pt.terrier.TerrierIndex('splade.terrier').indexer()
indexer1.index(cache)
indexer2 = pyterrier_pisa.PisaIndex('./path/to/index.pisa').toks_indexer()
indexer2.index(cache)
```

Note that unlike the other caching components, `IndexerCache` captures a sequence of documents, where the order of records is potentially important [24]. Consequently, it does not wrap a pipeline the way that other components do (i.e., `Splade() >> IndexerCache('path')` instead of `IndexerCache('path', Splade())`). The user may decide that the order is not important; in these cases, it can also be used as a basic forward index, as it allows for efficient row lookups based on the *docno* column (if present).

Implementation Details. `IndexerCache` stores the sequences as LZ4-compressed pickles of each input row. When the cache object is iterated over, the sequence is decompressed row-by-row as a row generator. The index also captures the *docno* column (if present), and stores it in an npids file to facilitate the forward index functionality.

[11] Note that this does not apply to some types of scorers, e.g., adaptive rerankers [17] (which add new documents to the pool), or pairwise [18] and listwise [19] rerankers (for which each score depends on the others present in the pool).

[12] https://github.com/seanmacavaney/npids

4.5. Other Caching Features

Cache objects conform to PyTerrier's Artifact API, which allows them to be shared using HuggingFace, Zenodo, or other platforms. This can enable the sharing of computational resources across research groups.

```
cache.to_hf('username/some-cache') # upload to HuggingFace
cache.to_zenodo() # upload to Zenodo
cache = pt.Artifact.from_hf('username/some-cache') # download from HuggingFace
cache = pt.Artifact.from_zenodo('1234') # download from Zenodo
```

All caches support a "temporary" cache mode, where a temporary cache directory is created and cleaned up when the object is deleted. This setting is applied by omitting the index path when creating the cache. We recommend using these as context managers so that the lifetime and cleanup of the temporary cache is well-defined. For example:

```
from pyterrier_caching import RetrieverCache
dataset = pt.get_dataset('irds:msmarco-passage/dev/small')
index = pt.terrier.TerrierIndex.from_hf('macavaney/msmarco-passage.terrier')

# construct a temporary retriever cache
with RetrieverCache(retriever=index.bm25()) as bm25_cache:
   bm25_cache(dataset.get_topics())
   # second time faster due to caching
   bm25_cache(dataset.get_topics())
# (temporary cache deleted when the context manager exits)
```

All the caches also support a mode where the transformer is not provided. If there is a cache miss and the transformer is not provided, an exception is raised. Alternatively, the transformer can be constructed on-demand using the `Lazy` utility transformer, which only constructs the actual transformer once if/when it is invoked. Both are helpful for situations where the user wants to avoid constructing the transformer object due to the resources that it would consume, e.g., a GPU for a neural scorer. For example:

```
from pyterrier_caching import ScorerCache, Lazy
from pyterrier_dr import ElectraScorer

# scorer omitted (raises error on cache miss)
cached_scorer = ScorerCache('electra.cache')

# lazy scorer (only constructed once called)
lazy_scorer = Lazy(lambda: dataset.text_loader() >> ElectraScorer())
lazy_cached_scorer = ScorerCache('electra.cache', lazy_scorer)
```

5. Demonstration Experiments

To demonstrate the benefit of the PyTerrier techniques described in this paper, we examine the response times of four different experimental settings. Our chosen experiment is based on the example pipelines discussed in Section 2.2, where we vary the number of documents retrieved by BM25 to determine the impact on MonoT5 and DuoT5. Such an experiment would use a list of pipelines expressed as follows: `[bm25 % k >> MonoT5() % 10 >> DuoT5() for k in [20, 50, 100, 200]]`.

The four different experimental settings are: (1) without using any caching, such that BM25 is executed four times on each query; (2) using precomputation (Section 3), such that BM25 is only executed once; (3) using a cold ScorerCache around MonoT5, such that MonoT5 is not executed more than once for a given query/document pair within the experiment; and (4) a re-run of the same setting where the ScorerCache is hot, such that MonoT5 reuses the results computed in (3). Finally, recall that DuoT5 is not amenable to caching in these pipelines, as the overall score of a document depends on the other retrieved documents for that query.

#	Precomputation	Cached MonoT5	MSMARCO v1 Execution Time	Δ	MSMARCO v2 Execution Time	Δ
(1)	✗	✗	3min 11s	-	5min 48s	-
(2)	✓	✗	2min 55s	92%	4min 13s	72%
(3)	✓	✓ (cold)	2min 19s	73%	3min 27s	59%
(4)	✓	✓ (hot)	1min 36s	50%	2min 25s	42%

Table 2
Execution times for an experiment comparing `[bm25 % k » MonoT5() % 10 » DuoT5() for k in [20, 50, 100, 200]]` on queries from the TREC 2019 & 2021 Deep Learning tracks. Relative decreases in experiment execution time compared to row (1) are denoted by Δ.

We execute these experiments on a machine with Intel Xeon Gold 5222 CPU @ 3.80GHz (16 cores) and an NVIDIA RTX 3090 GPU. We use a Terrier backend for BM25, with the index stored on disk. We use two indices: MSMARCO v1 passage corpus using 43 queries from the TREC 2019 Deep Learning track; and MSMARCO V2 passage corpus using 53 queries from the TREC 2021 Deep Learning track. The Jupyter notebooks for executing these experiments can be found on the PyTerrier GitHub repo: https://github.com/terrier-org/pyterrier/blob/master/examples/notebooks.md.

The obtained timings are shown in Table 2, allowing the following observations: precomputation allows reducing the experimental execution time by 8% on MSMARCO v1 and 28% on MSMARCO v2 – the small relative benefit on MSMARCO v1 is due to the small size of the MSMARCO v1 index and resulting fast BM25 retrieval (only 3 seconds for 43 queries); Caching of MonoT5 results shows a benefit of 27-41%; Rerunning using a hot ScorerCache reduced overall time to 50-68% - essentially executing BM25 once, and the reexecutions of DuoT5. Overall, the table supports the expected benefits of prefix precomputation and appropriate transformer caching, which are easily accessible through PyTerrier's `pt.Experiment` API and the the pyterrier-caching package. Prefix precomputation is more beneficial for more expensive shared prefixes of transformer pipelines in an experiment.

6. Limitations and Future Work

We have described two approaches for caching in PyTerrier – a precomputation approach that can be automatically applied to experiments, and explicit caching components that can be incorporated into indexing and retrieval pipelines. Although these additions provide a major improvement over the prior operator-based caching strategy, we see these as promising starting points, rather than final products, due to several shortcomings.

Although the precomputation approach covers many practical experimental settings, it is not comprehensive. For instance, consider an ablation experiment where components are progressively added to a pipeline: (1) A, (2) A » B, (3) A » B » C. The precomputation approach will only precompute A (since it's common to all three pipelines), even though also precomputing A » B would be able to benefit both pipelines 2 and 3. Furthermore, the current strategy only supports precomputation across sequential pipelines; operations that occur as part of linear combinations, set operations, or feature combinations are not supported. We aim to assess which of these cases can be practically addressed and efficiently precomputed in future iterations of this feature – potentially in combination with the `pipeline.compile()` functionality [2].

The explicit caches also have limitations. Most notably, they rely on direct application by the researcher. This is by design, since current transformer implementations do not provide sufficient information to automatically infer the correct caching strategy. In the future, we may enhance the Transformer API to include this kind of information, e.g. the input and output columns, to ease the process of identifying the caching strategy to apply.[13]

Both precomputation and explicit caching make determinism assumptions; that is, the same input

[13] An added benefit is this information would also allow the automatic type-checking of pipelines.

will yield the same output. This is not always the case, especially for components running on GPUs.[14] In some sense, caching reduces the variability in experiments due to this noise. On the other hand, it means that such variations are not encountered through the course of experimentation, which could lead the researcher to infer a false sense of stability.

Finally, precomputation is an first study of optimisation across multiple IR pipelines, and can be seen as similar to multi-query optimisation [25] in database management systems - we believe there may be other multi-query optimisation techniques from databases that can result in improved IR experimentation, which we leave to future work.

7. Conclusion

We presented recent additions to the open-source PyTerrier platform to better facilitate caching. This involves two approaches: the precomputation of common pipeline prefixes when executing an experiment and new explicit result caching components. Demonstration experiments on MSMARCO v1 and v2 passage corpora concerning a pipeline with BM25, monoT5 and DuoT5 showed how precomputation and caching could reduce the execution time of a particular experiment by upto 41% in a cold-cache setting. We hope that these approaches are intuitive for researchers to use, and will help reduce the computational cost of running experiments, promote GreenIR principles, and ease collaboration through shared caches.

Acknowledgments

We thank Jan Heinrich Merker for helpful feedback and suggestions on the pyterrier-caching package. We also thank Andrew Parry for help implementing the longest common prefix algorithm for prefix computation. Finally, we thank the anonymous reviewers for their detailed and thoughtful feedback.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly for: Grammar and spelling check. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] X. Ho, A. D. Nguyen, S. Sugawara, A. Aizawa, Constructing A multi-hop QA dataset for comprehensive evaluation of reasoning steps, in: D. Scott, N. Bel, C. Zong (Eds.), Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020, International Committee on Computational Linguistics, 2020, pp. 6609–6625. doi:10.18653/v1/2020.coling-main.580.

[2] C. Macdonald, N. Tonellotto, Declarative experimentation in information retrieval using PyTerrier, in: K. Balog, V. Setty, C. Lioma, Y. Liu, M. Zhang, K. Berberich (Eds.), ICTIR '20: The 2020 ACM SIGIR International Conference on the Theory of Information Retrieval, Virtual Event, Norway, September 14-17, 2020, ACM, 2020, pp. 161–168. doi:10.1145/3409256.3409829.

[3] C. Macdonald, N. Tonellotto, S. MacAvaney, I. Ounis, PyTerrier: Declarative experimentation in Python from BM25 to dense retrieval, in: G. Demartini, G. Zuccon, J. S. Culpepper, Z. Huang, H. Tong (Eds.), CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021, ACM, 2021, pp. 4526–4533. doi:10.1145/3459637.3482013.

[14] The non-determinism on GPUs arises from the non-deterministic order in which operations are assigned to the GPU, which can ultimately cause cascading differences in floating point operations.

[4] H. Scells, S. Zhuang, G. Zuccon, Reduce, reuse, recycle: Green information retrieval research, in: E. Amigó, P. Castells, J. Gonzalo, B. Carterette, J. S. Culpepper, G. Kazai (Eds.), SIGIR '22: The 45th International ACM SIGIR Conference on Research and Development in Information Retrieval, Madrid, Spain, July 11 - 15, 2022, ACM, 2022, pp. 2825–2837. doi:10.1145/3477495.3531766.

[5] E. Bassani, ranxhub: An online repository for information retrieval runs, in: H. Chen, W. E. Duh, H. Huang, M. P. Kato, J. Mothe, B. Poblete (Eds.), Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2023, Taipei, Taiwan, July 23-27, 2023, ACM, 2023, pp. 3210–3214. doi:10.1145/3539618.3591823.

[6] T. Breuer, J. Keller, P. Schaer, ir_metadata: An extensible metadata schema for IR experiments, in: E. Amigó, P. Castells, J. Gonzalo, B. Carterette, J. S. Culpepper, G. Kazai (Eds.), SIGIR '22: The 45th International ACM SIGIR Conference on Research and Development in Information Retrieval, Madrid, Spain, July 11 - 15, 2022, ACM, 2022, pp. 3078–3089. doi:10.1145/3477495.3531738.

[7] M. Fröbe, J. H. Reimer, S. MacAvaney, N. Deckers, J. Bevendorff, B. Stein, M. Hagen, M. Potthast, The information retrieval experiment platform, in: M. Leyer, J. Wichmann (Eds.), Lernen, Wissen, Daten, Analysen (LWDA) Conference Proceedings, Marburg, Germany, October 9-11, 2023, volume 3630 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 175–178. URL: https://ceur-ws.org/Vol-3630/LWDA2023-paper16.pdf.

[8] M. Fröbe, J. H. Reimer, S. MacAvaney, N. Deckers, S. Reich, J. Bevendorff, B. Stein, M. Hagen, M. Potthast, The information retrieval experiment platform, in: H. Chen, W. E. Duh, H. Huang, M. P. Kato, J. Mothe, B. Poblete (Eds.), Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2023, Taipei, Taiwan, July 23-27, 2023, ACM, 2023, pp. 2826–2836. doi:10.1145/3539618.3591888.

[9] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, F. Silvestri, The impact of caching on search engines, in: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, 2007, pp. 183–190. doi:10.1145/1277741.1277775.

[10] C. Macdonald, R. L. Santos, I. Ounis, B. He, About learning models with multiple query-dependent features, ACM Trans. Inf. Syst. 31 (2013). doi:10.1145/2493175.2493176.

[11] Q. Gan, T. Suel, Improved techniques for result caching in web search engines, in: Proceedings of the 18th international conference on World Wide Web, 2009, pp. 431–440. doi:10.1145/1526709.1526768.

[12] E. P. Markatos, On caching search engine query results, Computer Communications 24 (2001) 137–143. doi:10.1016/S0140-3664(00)00308-X.

[13] N. Fuhr, Proof by experimentation? Towards better IR research, in: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 2. doi:10.1145/3397271.3402426.

[14] T. Sakai, On Fuhr's guideline for IR evaluation, SIGIR Forum 54 (2021). doi:10.1145/3451964.3451976.

[15] R. F. Nogueira, W. Yang, J. Lin, K. Cho, Document expansion by query prediction, CoRR abs/1904.08375 (2019). URL: http://arxiv.org/abs/1904.08375. arXiv:1904.08375.

[16] S. E. Robertson, The probability ranking principle in IR, Journal of documentation 33 (1977) 294–304. doi:10.1108/eb026647.

[17] S. MacAvaney, N. Tonellotto, C. Macdonald, Adaptive re-ranking with a corpus graph, in: M. A. Hasan, L. Xiong (Eds.), Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022, ACM, 2022, pp. 1491–1500. doi:10.1145/3511808.3557231.

[18] R. Pradeep, R. Nogueira, J. Lin, The expando-mono-duo design pattern for text ranking with pretrained sequence-to-sequence models, CoRR abs/2101.05667 (2021). URL: https://arxiv.org/abs/2101.05667. arXiv:2101.05667.

[19] W. Sun, L. Yan, X. Ma, S. Wang, P. Ren, Z. Chen, D. Yin, Z. Ren, Is chatgpt good at search? investigating large language models as re-ranking agents, in: H. Bouamor, J. Pino, K. Bali (Eds.),

Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023, Association for Computational Linguistics, 2023, pp. 14918–14937. doi:10.18653/V1/2023.EMNLP-MAIN.923.

[20] R. Pradeep, Y. Liu, X. Zhang, Y. Li, A. Yates, J. Lin, Squeezing water from a stone: A bag of tricks for further improving cross-encoder effectiveness for reranking, in: M. Hagen, S. Verberne, C. Macdonald, C. Seifert, K. Balog, K. Nørvåg, V. Setty (Eds.), Advances in Information Retrieval - 44th European Conference on IR Research, ECIR 2022, Stavanger, Norway, April 10-14, 2022, Proceedings, Part I, volume 13185 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 655–670. doi:10.1007/978-3-030-99736-6_44.

[21] S. MacAvaney, X. Wang, Online distillation for pseudo-relevance feedback, CoRR abs/2306.09657 (2023). URL: https://arxiv.org/abs/2306.09657. doi:10.48550/ARXIV.2306.09657. arXiv:2306.09657.

[22] T. Nguyen, S. MacAvaney, A. Yates, A unified framework for learned sparse retrieval, in: J. Kamps, L. Goeuriot, F. Crestani, M. Maistro, H. Joho, B. Davis, C. Gurrin, U. Kruschwitz, A. Caputo (Eds.), Advances in Information Retrieval - 45th European Conference on Information Retrieval, ECIR 2023, Dublin, Ireland, April 2-6, 2023, Proceedings, Part III, volume 13982 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 101–116. doi:10.1007/978-3-031-28241-6_7.

[23] T. Formal, B. Piwowarski, S. Clinchant, SPLADE: sparse lexical and expansion model for first stage ranking, in: F. Diaz, C. Shah, T. Suel, P. Castells, R. Jones, T. Sakai (Eds.), SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021, ACM, 2021, pp. 2288–2292. doi:10.1145/3404835.3463098.

[24] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, A. Shalita, Compressing graphs and indexes with recursive graph bisection, in: B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, R. Rastogi (Eds.), Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016, ACM, 2016, pp. 1535–1544. doi:10.1145/2939672.2939862.

[25] P. Roy, S. Sudarshan, Multi-Query Optimization, Springer US, Boston, MA, 2009, pp. 1849–1852. doi:10.1007/978-0-387-39940-9_239.