Indexing

Here, we will look at a basic indexing workflow using the LangChain indexing API.

The indexing API lets you load and keep in sync documents from any source into a vector store. Specifically, it helps:

- Avoid writing duplicated content into the vector store
- Avoid re-writing unchanged content
- Avoid re-computing embeddings over unchanged content

All of which should save you time and money, as well as improve your vector search results.

Crucially, the indexing API will work even with documents that have gone through several transformation steps (e.g., via text chunking) with respect to the original source documents.

How it works

LangChain indexing makes use of a record manager (`RecordManager`) that keeps track of document writes into the vector store.

When indexing content, hashes are computed for each document, and the following information is stored in the record manager:

- the document hash (hash of both page content and metadata)
- write time
- the source id – each document should include information in its metadata to allow us to determine the ultimate source of this document

Deletion modes

When indexing documents into a vector store, it's possible that some existing documents in the vector store should be deleted. In certain situations you may want to remove any existing documents that are derived from the same sources as the new documents being indexed. In others you may want to delete all existing documents wholesale. The indexing API deletion modes let you pick the behavior you want:

Cleanup Mode	De-Duplicates Content	Parallelizable	Cleans Up Deleted Source Docs	Cl M o
None	✅	✅	❌	❌
Incremental	✅	✅	❌	✅
Full	✅	❌	✅	✅

`None` does not do any automatic clean up, allowing the user to manually do clean up of old content.

`incremental` and `full` offer the following automated clean up:

- If the content of the source document or derived documents has **changed**, both `incremental` or `full` modes will clean up (delete) previous versions of the content.

- If the source document has been **deleted** (meaning it is not included in the documents currently being indexed), the `full` cleanup mode will delete it from the vector store correctly, but the `incremental` mode will not.

When content is mutated (e.g., the source PDF file was revised) there will be a period of time during indexing when both the new and old versions may be returned to the user. This happens after the new content was written, but before the old version was deleted.

- `incremental` indexing minimizes this period of time as it is able to do clean up continuously, as it writes.
- `full` mode does the clean up after all batches have been written.

Requirements

1. Do not use with a store that has been pre-populated with content independently of the indexing API, as the record manager will not know that records have been inserted previously.
2. Only works with LangChain `vectorstore`'s that support:
   - document addition by id (`add_documents` method with `ids` argument)

  - delete by id (`delete` method with `ids` argument)

Compatible Vectorstores: `AnalyticDB`, `AstraDB`, `AwaDB`, `Bagel`, `Cassandra`, `Chroma`, `DashVector`, `DatabricksVectorSearch`, `DeepLake`, `Dingo`, `ElasticVectorSearch`, `ElasticsearchStore`, `FAISS`, `HanaDB`, `Milvus`, `MyScale`, `PGVector`, `Pinecone`, `Qdrant`, `Redis`, `ScaNN`, `SupabaseVectorStore`, `SurrealDBStore`, `TimescaleVector`, `Vald`, `Vearch`, `VespaStore`, `Weaviate`, `ZepVectorStore`.

Caution

The record manager relies on a time-based mechanism to determine what content can be cleaned up (when using `full` or `incremental` cleanup modes).

If two tasks run back-to-back, and the first task finishes before the clock time changes, then the second task may not be able to clean up content.

This is unlikely to be an issue in actual settings for the following reasons:

1. The RecordManager uses higher resolution timestamps.

2. The data would need to change between the first and the second tasks runs, which becomes unlikely if the time interval between the tasks is small.
3. Indexing tasks typically take more than a few ms.

Quickstart

```
from langchain.indexes import
SQLRecordManager, index
from langchain.schema import Document
from langchain_community.vectorstores import
ElasticsearchStore
from langchain_openai import OpenAIEmbeddings
```

Initialize a vector store and set up the embeddings:

```
collection_name = "test_index"

embedding = OpenAIEmbeddings()

vectorstore = ElasticsearchStore(
    es_url="http://localhost:9200",
index_name="test_index", embedding=embedding
)
```

Initialize a record manager with an appropriate namespace.

Suggestion: Use a namespace that takes into account both the vector store and the collection name in the vector store; e.g., ‘redis/my_docs’, ‘chromadb/my_docs’ or ‘postgres/my_docs’.

```
namespace =
f"elasticsearch/{collection_name}"
record_manager = SQLRecordManager(
    namespace,
db_url="sqlite:///record_manager_cache.sql"
)
```

Create a schema before using the record manager.

```
record_manager.create_schema()
```

Let’s index some test documents:

```
doc1 = Document(page_content="kitty",
metadata={"source": "kitty.txt"})
doc2 = Document(page_content="doggy",
metadata={"source": "doggy.txt"})
```

Indexing into an empty vector store:

```
def _clear():
    """Hacky helper method to clear content.
```

```
See the `full` mode section to to understand
why it works."""
    index([], record_manager, vectorstore,
cleanup="full", source_id_key="source")
```

None deletion mode

This mode does not do automatic clean up of old versions of content; however, it still takes care of content de-duplication.

```
_clear()
```

```
index(
    [doc1, doc1, doc1, doc1, doc1],
    record_manager,
    vectorstore,
    cleanup=None,
    source_id_key="source",
)
```

```
{'num_added': 1, 'num_updated': 0,
'num_skipped': 0, 'num_deleted': 0}
```

```
_clear()
```

```
index([doc1, doc2], record_manager,
vectorstore, cleanup=None,
source_id_key="source")
```

```
{'num_added': 2, 'num_updated': 0,
'num_skipped': 0, 'num_deleted': 0}
```

Second time around all content will be skipped:

```
index([doc1, doc2], record_manager,
vectorstore, cleanup=None,
source_id_key="source")
```

```
{'num_added': 0, 'num_updated': 0,
'num_skipped': 2, 'num_deleted': 0}
```

`"incremental"` deletion mode

```
_clear()
```

```
index(
    [doc1, doc2],
    record_manager,
    vectorstore,
    cleanup="incremental",
    source_id_key="source",
)
```

```
{'num_added': 2, 'num_updated': 0,
'num_skipped': 0, 'num_deleted': 0}
```

Indexing again should result in both documents getting **skipped** – also skipping the embedding operation!

```
index(
    [doc1, doc2],
    record_manager,
    vectorstore,
    cleanup="incremental",
    source_id_key="source",
)
```

```
{'num_added': 0, 'num_updated': 0,
'num_skipped': 2, 'num_deleted': 0}
```

If we provide no documents with incremental indexing mode, nothing will change.

```
index([], record_manager, vectorstore,
cleanup="incremental",
source_id_key="source")
```

```
{'num_added': 0, 'num_updated': 0,
'num_skipped': 0, 'num_deleted': 0}
```

If we mutate a document, the new version will be written and all old versions sharing the same source will be deleted.

```
changed_doc_2 =
Document(page_content="puppy", metadata=
{"source": "doggy.txt"})
```

```
index(
    [changed_doc_2],
    record_manager,
    vectorstore,
    cleanup="incremental",
    source_id_key="source",
)
```

```
{'num_added': 1, 'num_updated': 0,
'num_skipped': 0, 'num_deleted': 1}
```

`"full"` deletion mode

In `full` mode the user should pass the `full` universe of content that should be indexed into the indexing function.

Any documents that are not passed into the indexing function and are present in the vectorstore will be deleted!

This behavior is useful to handle deletions of source documents.

```
_clear()
```

```
all_docs = [doc1, doc2]
```

```
index(all_docs, record_manager, vectorstore,
cleanup="full", source_id_key="source")
```

```
{'num_added': 2, 'num_updated': 0,
'num_skipped': 0, 'num_deleted': 0}
```

Say someone deleted the first doc:

```
del all_docs[0]
```

```
all_docs
```

```
[Document(page_content='doggy', metadata=
{'source': 'doggy.txt'})]
```

Using full mode will clean up the deleted content as well.

```
index(all_docs, record_manager, vectorstore,
cleanup="full", source_id_key="source")
```

```
{'num_added': 0, 'num_updated': 0,
'num_skipped': 1, 'num_deleted': 1}
```

Source

The metadata attribute contains a field called `source`. This source should be pointing at the *ultimate* provenance associated with the given document.

For example, if these documents are representing chunks of some parent document, the `source` for both documents should be the same and reference the parent document.

In general, `source` should always be specified. Only use a `None`, if you **never** intend to use `incremental` mode, and for some reason can't specify the `source` field correctly.

```
from langchain.text_splitter import
CharacterTextSplitter
```

```
doc1 = Document(
    page_content="kitty kitty kitty kitty
kitty", metadata={"source": "kitty.txt"}
)
doc2 = Document(page_content="doggy doggy the
doggy", metadata={"source": "doggy.txt"})
```

```
new_docs = CharacterTextSplitter(
    separator="t", keep_separator=True,
chunk_size=12, chunk_overlap=2
).split_documents([doc1, doc2])
new_docs
```

```
[Document(page_content='kitty kit', metadata=
{'source': 'kitty.txt'}),
 Document(page_content='tty kitty ki',
metadata={'source': 'kitty.txt'}),
 Document(page_content='tty kitty', metadata=
{'source': 'kitty.txt'}),
 Document(page_content='doggy doggy',
metadata={'source': 'doggy.txt'}),
 Document(page_content='the doggy', metadata=
{'source': 'doggy.txt'})]
```

```
_clear()
```

```
index(
    new_docs,
    record_manager,
    vectorstore,
    cleanup="incremental",
    source_id_key="source",
)
```

```
{'num_added': 5, 'num_updated': 0,
'num_skipped': 0, 'num_deleted': 0}
```

```
changed_doggy_docs = [
    Document(page_content="woof woof",
metadata={"source": "doggy.txt"}),
    Document(page_content="woof woof woof",
metadata={"source": "doggy.txt"}),
]
```

This should delete the old versions of documents associated with `doggy.txt` source and replace them with the new versions.

```
index(
    changed_doggy_docs,
    record_manager,
    vectorstore,
    cleanup="incremental",
    source_id_key="source",
)
```

```
{'num_added': 0, 'num_updated': 0,
'num_skipped': 2, 'num_deleted': 2}
```

```
vectorstore.similarity_search("dog", k=30)
```

```
[Document(page_content='tty kitty', metadata=
{'source': 'kitty.txt'}),
 Document(page_content='tty kitty ki',
metadata={'source': 'kitty.txt'}),
 Document(page_content='kitty kit', metadata=
{'source': 'kitty.txt'})]
```

Using with loaders

Indexing can accept either an iterable of documents or else any loader.

Attention: The loader **must** set source keys correctly.

```
from
langchain_community.document_loaders.base
import BaseLoader


class MyCustomLoader(BaseLoader):
    def lazy_load(self):
        text_splitter =
CharacterTextSplitter(
            separator="t",
keep_separator=True, chunk_size=12,
chunk_overlap=2
        )
        docs = [
            Document(page_content="woof
woof", metadata={"source": "doggy.txt"}),
            Document(page_content="woof woof
woof", metadata={"source": "doggy.txt"}),
        ]
        yield from
text_splitter.split_documents(docs)

    def load(self):
        return list(self.lazy_load())
```

```
_clear()
```

```
loader = MyCustomLoader()
```

```
loader.load()
```

```
[Document(page_content='woof woof', metadata=
{'source': 'doggy.txt'}),
 Document(page_content='woof woof woof',
metadata={'source': 'doggy.txt'})]
```

```
index(loader, record_manager, vectorstore,
cleanup="full", source_id_key="source")
```

```
{'num_added': 2, 'num_updated': 0,
'num_skipped': 0, 'num_deleted': 0}
```

```
vectorstore.similarity_search("dog", k=30)
```

```
[Document(page_content='woof woof', metadata=
{'source': 'doggy.txt'}),
 Document(page_content='woof woof woof',
metadata={'source': 'doggy.txt'})]
```