Caching

Context

There are advantages and disadvantages when fetching data from remote storage (e.g., buckets). Remote storage offers conveniences, such as easy data sharing between applications and users. Additionally, remote storage is often more cost-effective than local storage for long-term data repositories, which is particularly beneficial for large datasets.

Despite these advantages, there are potential drawbacks to fetching data from remote storage. One significant concern is slower data retrieval, which can impact the performance of your Squirrel application. Retrieving data over a network connection introduces additional latency over accessing data locally. Another factor to consider is the pricing structure associated with remote storage. Typically, remote storage costs are not solely based on how long which amount of data is stored but also on the amount of data transferred across the network and the number of requests made when retrieving the data. Consequently, you may incur extra costs, especially if you need to fetch a large number of shards across the network connection.

Caching to the Rescue

In Machine Learning workloads, models are often trained over multiple epochs, meaning you may need to fetch the same data multiple times during a run. To optimize this process, imagine if you could load the remote data only in the first pass (first epoch), store it locally, and subsequently access the data from the fast and inexpensive local disk. Precisely this functionality is offered through caching.

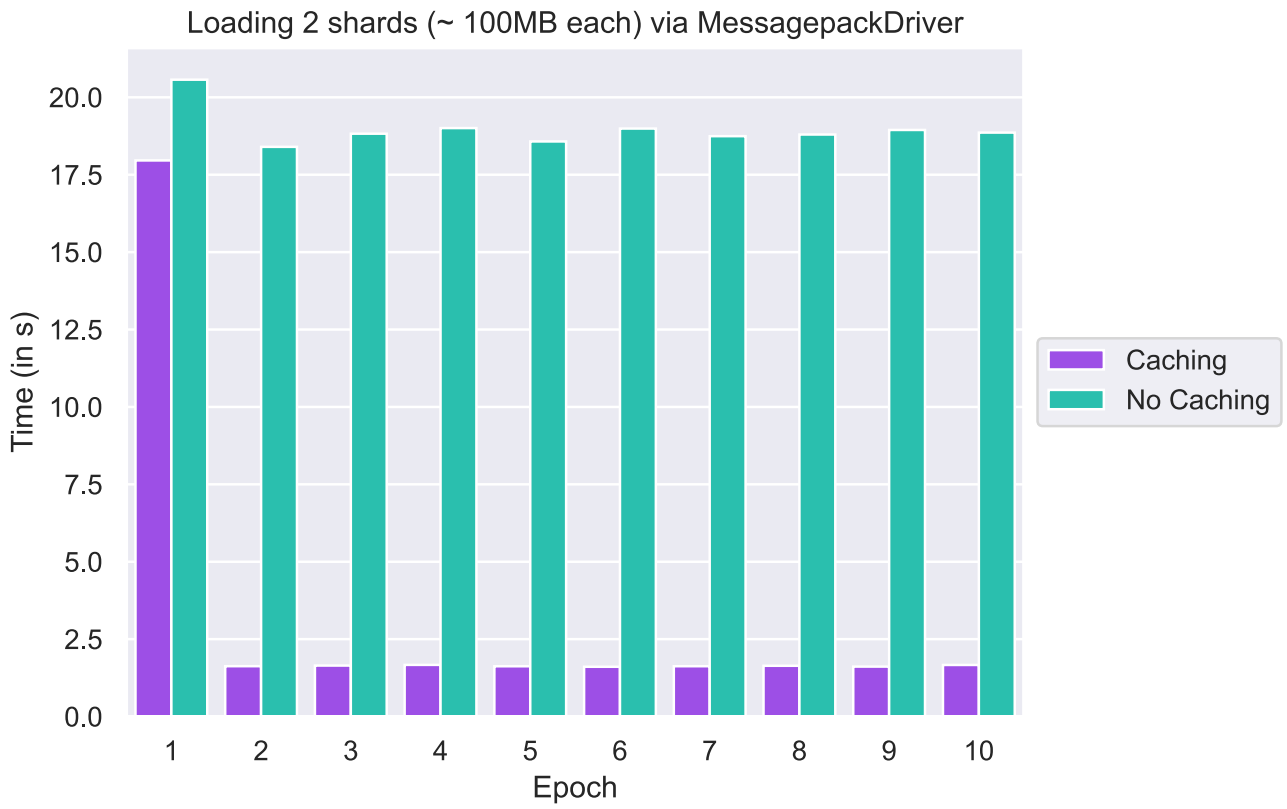
Squirrel leverages the capabilities of the `fsspec` library, which includes a powerful caching feature out of the box. Each default Squirrel `Driver` such as `DataFrameDriver`, `FileDriver`, or `StoreDriver` accepts a `storage_options` argument, which is a dictionary passed down to the `fsspec` filesystem. This dictionary allows you to configure caching, among other things. For more detailed information, please refer to the `fsspec` documentation on local caching.

The code below shows an example of configuring caching for several drivers. Note that, as per the `fsspec` documentation, only `simplecache` is “guaranteed thread/process-safe”.

```python
from squirrel.driver import CsvDriver, FileDriver, MessagepackDriver

so = {"protocol": "simplecache", "target_protocol": "gs", "cache_storage":
"/tmp/cache"}

CsvDriver("gs://bucket/data.csv", engine="pandas", storage_options=so)  # inherits
from DataFrameDriver
MessagepackDriver("gs://bucket/data-dir", storage_options=so)  # inherits from
StoreDriver
FileDriver("gs://bucket/file.txt", storage_options=so)
```

Let's observe the performance benefits of caching in action. The below code compares the performance of a `MessagepackDriver` with and without caching. The generated plot shows that the non-cached driver has a similar loading speed for all epochs. However, the cached driver stores the data on the local disk in the first epoch and reads it from the local disk in the subsequent epochs, making it much faster than the non-cached driver.

```python
import time
import typing as t

import matplotlib.pyplot as plt
import seaborn as sns
from pandas import DataFrame
from squirrel.driver import MessagepackDriver

remote_path = "gs://mxm-safetrain-data/msgpack-cache-demo-data"
so = {"protocol": "simplecache", "target_protocol": "gs", "cache_storage":
"/tmp/cache"}

driver_types = {
    "Caching": lambda: MessagepackDriver(remote_path, storage_options=so),
    "No Caching": lambda: MessagepackDriver(remote_path),
}


def time_epoch(driver_init: t.Callable[[], MessagepackDriver]) -> float:
    """Creates iterator and returns time to fetch all shards."""
    it = driver_init().get_iter().tqdm()
    start = time.time()
    it.join()
    return time.time() - start


def benchmark(driver_types: t.Dict[str, t.Any], num_epochs: int = 10) -> DataFrame:
    """Benchmarks loading speed of the `driver_types` over `num_epochs` epochs."""
    df = DataFrame()
    for k, v in driver_types.items():
        for i in range(num_epochs):
            data = {"Epoch": i + 1, "Time (in s)": time_epoch(driver_init=v), "Driver
Type": k}
            df = df.append(data, ignore_index=True)
    return df


df = benchmark(driver_types)

sns.set_theme(style="darkgrid")
sns.set_palette(["#9E36FF", "#11D8C1"])  # company colors

ax = sns.barplot(data=df, x="Epoch", y="Time (in s)", hue="Driver Type")
ax.set(title="Loading 2 shards (~ 100MB each) via MessagepackDriver")
ax.legend(loc="center left", bbox_to_anchor=(1, 0.5))

plt.savefig("advanced/msgpack_caching.svg", bbox_inches="tight")
```



Storage Options and Catalogs

When using the `Catalog`-API, some users will have written some `storage_options` to the catalog (e.g., that their Google Cloud Service (GCS) bucket is `requester_pays=True`). As a new user, you might now want to provide additional `storage_options` (e.g., for caching). As shown below in the code, you can do so when you call `get_driver()` on the `CatalogSource`. Squirrel ensures that the new `storage_options` passed to `get_driver()` are merged with the pre-existing `storage_options`.

```python
from squirrel.catalog import Catalog, Source

# user 1 creates a catalog, saves it, and shares it with user 2
cat = Catalog()
cat["source"] = Source(
    "csv",
    driver_kwargs={
        "url": "gs://some-bucket/test.csv",
        "storage_options": {"requester_pays": True},
    },
)
cat.to_file("catalog.yaml")

# user 2 loads catalog from file and inserts their storage_options
cat = Catalog.from_files(["catalog.yaml"])
driver = cat["source"].get_driver(
    storage_options={
        "protocol": "simplecache",
        "target_protocol": "gs",
        "cache_storage": "/tmp/cache",
    }
)

# storage_options from user 1 and user 2 are merged
assert driver.storage_options == {
    "requester_pays": True,
    "protocol": "simplecache",
    "target_protocol": "gs",
    "cache_storage": "/tmp/cache",
}
```