Universe Scientific Publishing

DOI:10.18686/ahe.v7i19.9422

Group-batching Algorithm for High-throughput Network Requests

Hongke Sun, Xin Shan

Shandong Vocational Collage of Information Technology, Weifang, 261061, Shandong

Abstract: It is common in distributed cloud systems to enhance the throughput of the system by grouping multiple API requests into a single batch API request to complete tasks in bulk. It is easy to batch them when the tasks are already managed by a single synchronous thread. But it is hard to group independent asynchronous tasks into several batches, each having its own unique attributes. This paper designed a leader election algorithm to effectively group high-throughput requests into several batches based on the requests' attributes. So that the upstream service is less likely to be throttled by the downstream service, and the downstream service has lower workload.

Keywords: Distributed system; High-throughput; Leader election algorithm; Batch processing

1. Introduction of the Problem

To balance between the availability and protection, some services may provide some batch APIs which can process multiple tasks in a single API request. For example, the batch API of some getResource API may be named as batchGetResources.The problem we are going to solve in this article is, in a high throughput distributed system, based on certain grouping conditions, how to batch concurrent asynchronous workflows into one single workflow, so it can invoke a batch API, while avoiding data loss caused by race conditions.

2. The Data Persistence assisting the Group-Batching Algorithm

As a possible example of implementation, in this paper we describe how we built the system infrastructure using Amazon Web Service (AWS). We used AWS DynamoDB (DDB) as the data base, and AWS Simple Queue Service (SQS) as the message queue to pass information between the asynchronous steps. First, let's take take a look at the persistent data that assists the workflow orchestration. For data persistence, we used three DDB tables:

2.1 Task Table - the record of each task

- Id: A globally unique ID of tasks. The DDB table's primary partition key.

- Status: The status of the task. Possible values include: created, batched, in-progress, succeeded, failed. The initial status is created. This field is the DDB table's GSI sort key.

- Target: Describing the parameterized operation supposed to be taken on the task. Tasks with the same target are eligible to be (but not necessarily) batched into a same group. It will be the DDB table's GSI partition key.

2.2 BatchLock Table - each record is shared by the task workflows that have the same Target

- Target: It plays the role similar as a foreign key of the Target in the Task Table, though foreign key is not an official concept in NoSQL.

- LockHolder: The workflow id that is holding this lock. Writing to this field will use DDB conditional check to make sure concurrent threads won't successfully write the data in a race condition.

- AcquiredAt: The timestamp that the lock was acquired. This is used to automatically expire the lock in the unexpected case of dead leader workflow.

2.3 Batch Table - This stores batches of tasks

- BatchId: A globally unique ID of the batch. It is the primary key of the DDB table.

- Target: The common target of the member tasks.

- Members: An array of task ids, indicating what tasks have been grouped into this batch.

For workflow processing, we used AWS Lambda as the runtime and AWS SQS as the the glue to pass information in the asynchronous system. In general, some steps of the workflow may have physical I/O operations. To avoid I/O operations from holding the computation resources for a long time idly, the steps should be asynchronous, hence need SQS to glue the asynchronous runtime.

3. The Design and Implementation of the Group-Batching Algorithm

The system consists a batch grouper module and a batch handler module.In the batch grouper module, our algorithm groups task workflows into a Batch, handing over the batch to the batch handler module.The flowing explains how each step of the grouping algorithm works in the batch grouper module, as shown in Figure 1.

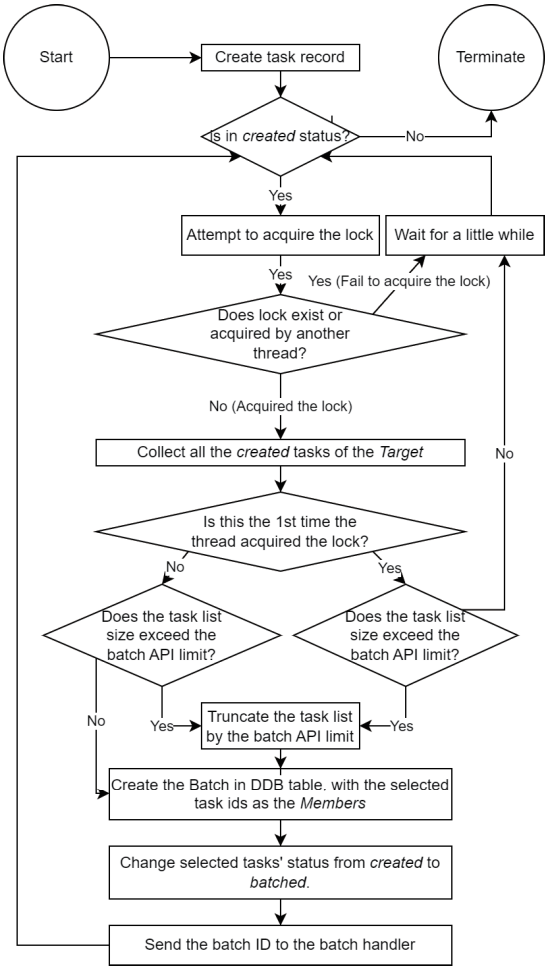

Figure 1 - The batching algorithm in the “batch grouper” module, with the details of leader election

3.1 Register task

Create the initial record in the Task Table.

3.2 Register task target

The system calculate the Target of the task based on the content of task record, and writes the Target into the Task Table. Only tasks that have the same Target are eligible to be grouped into a same batch. But they are not necessarily be grouped into a same batch, due to the batch size upper limit and the time that the task arrives.

3.3 Elect leader workflow

Multiple concurrent workflows handling tasks of a given Target compete to acquire a lock in the BatchLock Table. This can be achieved by leveraging DDB Conditional Expression, to guarantee exactly one workflow can successfully acquire the lock: A workflow can successfully acquire the lock if and only if the BatchLock item of the given Target does not exist or the LockHolder is exactly the id of the workflow.

Before a workflow attempts to acquire the lock, it should first check whether the task Status is still created. If not, it means the Status has already been modified by another workflow in the step 4 when adding tasks to a batch. Therefore, the current task is already in a batch. The task will be properly handled by the leader workflow of the batch. So, the current workflow should simply terminate, and do not continue competing for the lock.

If a workflow successfully acquires the lock, it creates a Batch Table item. Then it sends an SQS message to the next step, containing the BatchId. The Batch Table item does not have the Members set yet. But its Target is set to be the Target of the leader task. This Target of the batch item will be used to group tasks in the next step. If a workflow fails to acquire the lock (DDB throws a Cond itionalCheckFailedException), then the system leverages the SQS message delayed delivery feature or the message visibility timeout feature to wait for sometime (eg. 1s), then retry.

3.4 Leader groups tasks into a batch

The input of this step is the BatchId. According to the BatchId, we retrieve the Target stored in the Batch Table item. Then using the Task Table's GSI (Target and Status), we can quickly find out all the tasks that are in the created Status and that have their Target equal to the Batch's Target. These selected tasks are those that are eligible to be grouped into one batch. The system changes the Status of these tasks as batched, then put their task ids into the Members field of the Batch Table item. Now, the system has determined and persists all the tasks that should be batched, which is also the item_id_list to be passed to the downstream service. Finally, through SQS, the system notifies the next step to process the batch of the given BatchId.

Based on the pattern of concurrency and the strictness of the downstream service's throttling, if the Members array has not reached to the maximum allowed length, the system waits for a little while (eg. one second) to batch, expecting more new tasks having the given Target will arrive and registered by the step 1 and 2. The benefits of doing this is to include more tasks in a single batch.

3.5 Leader dispatches the batch

To be clear, the leader is the workflow that acquired the lock and grouped tasks into a batch. At this step, the system sends the BatchId in the message to the batch handler. The system also sets the Status of tasks as in-progress. Finally, the leader releases the lock in the BatchLock Table.

4. Batch Handler

In happy case, when the batch handler receives the BatchId sent by the batch grouper, it can directly use the BatchId to get the Batch Table item, then get all the tasks in the batch from the Members array. Then the batch handler puts the array as one of the arguments in the batch API call.

However, in unhappy cases, the batch API call can fail. In the case of retriable failures, the batch handler can leverage SQS's delay queues feature (by sending a copy of the message back to the queue) or the visibility timeout feature (by updating the visibility of the current message) to retry the failed message handling. By doing so, we are retrying asynchronously. Of course, based on different use cases, it may be helpful to implement both of the asynchronous retry with in-place synchronous retry. In the case of unretriable failures, the batch handler sends a failure notification callback to the initiator of the tasks of the batch.

5. Conclusion

We designed a group-batching algorithm that can be used in a distributed system to batch multiple concurrent asynchronous workflows into a single workflow. The system is decoupled so the generic batch grouper module can be reused to group for different batch APIs. One of the example use cases is to group shopping order workflows into a single workflow that distribute parcels to a delivery truck.

References:

[1] AWS documentation of DynamoDB Global Secondary Index: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html

[2] AWS documentation of DynamoDB read consistency: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html

[3] AWS documentation of SQS delay seconds and visibility timeout: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-delay-queues.html