Improving Batch Performance when Migrating to Microservices with Chunking and Coroutines

Holger Knoche
Kiel University, Software Engineering Group
24118 Kiel, Germany
hkn@informatik.uni-kiel.de

Abstract

When migrating enterprise software towards microservices, batch jobs are particularly sensitive to communication overhead introduced by the distributed nature of microservices. As it is not uncommon for a single batch job to process millions of data items, even an additional millisecond of overhead per item may lead to a significant increase in runtime.

A common strategy for reducing the average overhead per item is called chunking, which means that individual requests for different data items are grouped into larger requests. However, chunking is difficult to integrate into existing batch jobs, which are traditionally executed sequentially.

In this paper, we present a chunking approach based on coroutines, and investigate whether it can be used to mitigate the potential penalty to batch performance during migrations to microservices.

1 Introduction

Microservices are currently a very popular style for software architectures. This popularity is due to several reasons, one of them being improved maintainability, which previous research found to be a particularly important driver for adopting microservices in software modernization settings [5]. Such modernizations are usually carried out in an incremental manner, where new features are implemented immediately as microservices, and old features are gradually reimplemented [3].

A potential pitfall of this strategy in terms of performance is that the existing implementation may have to invoke the new microservices as well. Due to the distributed nature of microservices, this usually means remote communication such as REST calls, for which we measured an overhead of roughly one millisecond in our environment.

While an overhead of this magnitude may be negligible for user transactions with few invocations, it can lead to a significant increase in runtime for high-volume batch operations with hundreds of thousands or even millions of invocations. This is particularly unfortunate as batch jobs may be confined to a designated time frame, the so-called *batch window*, as not to interfere with interactive applications. For a batch job with one million invocations, an additional millisecond per invocation accumulates to about 17 minutes of additional runtime, which can be a significant portion of the batch window.

A common strategy to reduce the average overhead per invocation is to group several individual invocations into one larger "chunk" invocation, which we refer to as *chunking*.[1] Thus, the communication overhead only applies once per chunk, thereby reducing the effective overhead per individual invocation.

However, as discussed later, making effective use of chunking in batch jobs is far from trivial. In this paper, we present an approach based on coroutines and evaluate to what extent this approach can help improving batch performance when migrating to microservices.

The remainder of this paper is structured as follows: Section 2 provides background information on batch processing and coroutines. Then, the approach is described in Section 3. In Section 4, a short experimental evaluation is presented. Related work and conclusions are discussed in Sections 5 and 6.

2 Background

The following paragraphs provide the necessary background information on batch processing and coroutines.

2.1 Batch Processing

Batch processing refers to bulk data processing tasks without user interaction. This type of processing is particularly suited for high-volume tasks or tasks that need to run at given points in time, such as the end-of-day processing at a bank or report generation. In the following paragraphs, we briefly summarize the relevant concepts based on the domain language from Spring Batch [6].

A *batch job* is the primary runnable entity in batch processing. Each job consists of one or more *steps*, in which the actual processing takes place. The steps are usually orchestrated by internal DSLs (e.g., Spring Batch) or special scripting languages (e.g., the Job

[1] Another common name for this strategy is *microbatches*.

Control Language on the mainframe). These allow for conditional execution of steps or the abortion of the entire job if an unrecoverable error occurs.

Internally, a batch step typically consists of a *main loop* iterating over an input data set, such as the result of a database query or an input file. In the body of the loop, each data item is processed, and the results are written to the appropriate location.

Batch jobs can have complex dependencies (e.g., one job needs to run before another) or time constraints, which is why their execution is usually governed by automated job scheduling facilities. Typically, numerous jobs are run in parallel, while the steps are executed sequentially, although newer frameworks like Spring Batch also allow the parallel execution of steps.

2.2 Coroutines

Coroutines [1] are subroutines with the ability to voluntarily suspend their execution ("yield") and to later continue at the point where they left off. When a coroutine suspends, it frees the current execution thread so that another coroutine may run in it. Thus, multiple coroutines can be executed concurrently using only one or just a few threads.

The major advantage of coroutines over preemptive threads is that switching from one coroutine to another can be much cheaper in terms of performance than a context switch from one thread to another. Furthermore, coroutines can be implemented completely in userspace. Thus, a large number of coroutines can exist without occupying large amounts of operating system resources. The major downside is that in order for this concept to work, all coroutines must act cooperatively at all times. If one coroutine enters an infinite loop or invokes a thread-blocking operation (e.g., blocking I/O), it may bring the whole system down.

Although coroutines were already described and implemented in the 1960s, they have only recently gained mainstream attention for implementing software systems that need to handle large numbers of concurrent tasks. As a consequence, many mainstream programming languages such as COBOL and Java do not support them natively, as opposed some old (e.g., Simula) and recent (e.g., Go and Kotlin) programming languages. Since we will use Kotlin's coroutines in our evaluation, the relevant concepts from Kotlin are described below.

In Kotlin, communication between coroutines is typically realized with *channels*, which behave similar to bounded message queues. Coroutines sending data into a full channel suspend automatically, as do coroutines receiving data from an empty channel. A coroutine may wait for data from multiple channels at the same time by means of the `select` statement; however, only data from one channel is processed at a time.

3 Approach

As already noted in the introduction, our approach aims at improving batch performance by grouping multiple service invocations into one in order to reduce the effective overhead per invocation. For instance, instead of invoking a service individually for each customer, we intend to invoke the service only once for a chunk of 10 or 100 customers, thus ideally reducing the number of invocations by a factor of 10 or 100, respectively. This intention leads to two major requirements: (1) The respective services must operate on chunks of parameters instead of single parameters (e.g., multiple customer ids instead of a single one), and (2) the batch jobs must group their individual invocations into chunks and perform the invocation once a chunk is full.

While the first requirement only affects the newly created microservices and can thus be immediately incorporated into their design, the latter affects the existing batch implementations. What makes this change particularly difficult is that the grouping of invocations is, so to speak, orthogonal to a sequential main loop. For instance, consider a main loop that first reads a customer by its id and then performs further actions. Executing this loop sequentially does not allow for chunking, as no read to a second customer is requested until the first one returns. Therefore, multiple iterations of the loop need to run concurrently in order to make effective use of chunking.

In order to achieve this concurrency, we propose the following approach. First, the body of the main loop is wrapped in a coroutine, which allows to run multiple loop iterations concurrently. Then, the invocations of the microservices are changed so that instead of invoking the service individually and synchronously, they issue a request task via a channel to a *collector coroutine*. Besides the necessary parameters, the request task contains a *return channel*, on which the invoker suspends until data becomes available. If the chunk is not yet full, the collector coroutine just registers the request; otherwise, it invokes the chunk-enabled REST service and distributes the appropriate (partial) results via the provided return channels. As a consequence, the suspended invokers are reactivated and proceed with their work. Listing 1 sketches the coroutine-enabled main loop; as apparent, the necessary changes to the code are limited.

To prevent partially filled chunks from waiting indefinitely, a *ticker* coroutine sends signals in regular intervals to the collector coroutine. If such a ticker signal is received, the current chunk is processed regardless of its fullness, provided that it is not empty.

It should be noted that this approach could also be implemented using threads that block instead of suspending. However, for a desired chunk size of n, at least n concurrent loop iterations are advisable. As shown in the evaluation below, a chunk size of 50 to 100 invocations may be necessary for decent perfor-

```
for (contract in contracts) {
    launch mainLoopBody(contract)
}

coroutine mainLoopBody(Contract contract) {
  returnChannel = Channel<Customer>()

  readCustomer(contract.owner, returnChannel)
  // Receive suspends until data is available
  customer = returnChannel.receive()
  // ... further actions ...
}
```

Listing 1: Coroutine-enabled main loop

mance. Therefore, the thread count might increase by a factor of 100, resulting in a considerable consumption of operating system resources.

The major downside of this approach is that due to the introduction of concurrency, the main loop is no longer executed in a strict, predictable order. This can break code that relies on such an order. Therefore, additional modifications may be necessary before this approach can be applied.

4 Experimental Evaluation

For the experimental evaluation of the presented approach, we created a small batch job that creates letters informing customers of a fictional insurance company about their current premium. For this task, the batch job iterates over a given number of contracts from the database and then reads the customer's basic data (first name and last name) as well as its primary postal address. The batch job provides three types of access to acquire this data: (1) Direct access to the customer and address data using one SQL statement per contract, (2) individual REST calls to a customer and address service per contract, and (3) chunked REST calls to a customer and address service using our coroutine-based approach.

The batch job was implemented in Java and Kotlin without specific frameworks; the REST services were built using Spring Boot. The experiments were run on a Raspberry Pi 4 Model B with 4 GB of RAM running Raspbian Buster and the Azul Zulu JDK based on OpenJDK 1.8.0_222. For the underlying database, we used PostgreSQL 11.5 running in a Docker container on the same machine, with the data being stored on an external Toshiba STOR.E ALU 2S hard drive.

In total, we ran the batch job in six different configurations: direct SQL access, individual REST calls, and chunked REST calls with a chunk size of 10, 20, 50, and 100 requests, respectively. Each configuration was run 10 times with 100,000 contracts. After each run, the REST services and the database container were restarted, and the operating system's filesystem buffers were cleared. Table 1 lists the average runtimes for the different configurations.

As apparent from the table, chunking can achieve

Access type	**Avg. runtime (sec.) (99% CI)**
Direct SQL access	[42.035;46.669]
Individual REST	[895.351;947.513]
Chunked REST (10)	[131.459;138.640]
Chunked REST (20)	[79.479;81.901]
Chunked REST (50)	[47.366;50.475]
Chunked REST (100)	[35.801;37.330]

Table 1: Evaluation batch runtimes

runtimes similar to or even better than individual SQL accesses with sufficiently large chunk sizes. These results suggest that the presented approach is indeed capable of mitigating the potential performance penalty to batch jobs when migrating to microservices.

5 Related Work

Our chunking approach has similarities with *group prefetching* [2]. Coroutines are also used for performance improvement in database research, e.g., [4].

6 Conclusions

In this paper, we have presented and evaluated a coroutine-based approach to mitigate the potential performance penalty to batch jobs when migrating to microservices. The evaluation results suggest that this approach indeed has the potential to achieve this goal, although additional measures may be necessary to deal with the lack of a strict execution order. Future research might provide means to identify critical code more easily, and thus improve the applicability of the approach.

References

[1] M. Conway. "Design of a Separable Transition-Diagram Compiler". In: *Comm. of the ACM* 6.7 (1963).

[2] S. Chen et al. "Improving Hash Join Performance Through Prefetching". In: *ACM Trans. Database Syst.* 32.3 (2007).

[3] M. Stine. *Migrating to Cloud-Native Application Architectures*. O'Reilly, 2015.

[4] C. Jonathan et al. "Exploiting Coroutines to Attack the "Killer Nanoseconds"". In: *Proc. VLDB Endow.* 11.11 (2018).

[5] H. Knoche and W. Hasselbring. "Drivers and Barriers for Microservice Adoption – A Survey among Professionals in Germany". In: *Enterprise Modelling and Information Systems Architectures (EMISAJ)* 14 (2019).

[6] L. Ward et al. *Spring Batch – Reference Documentation*. `https://docs.spring.io/spring-batch/4.1.x/reference/html/index.html`. 2019.