:years: 2015-2022

Batch execution and jobs

Just like any financial system, Fineract also has batch jobs to achieve some processing on the data that's stored in the system.

The batch jobs in Fineract are implemented using Spring Batch. In addition to the Spring Batch ecosystem, the automatic scheduling is done by the Quartz Scheduler but it's also possible to trigger batch jobs via regular APIs.

Glossary

Job	A Job is an object that encapsulates an entire batch process.
Step	A Step is an object that encapsulates an independent phase of a Job.
Chunk oriented processing	Chunk oriented processing refers to reading the data one at a time and creating 'chunks' that are written out within a transaction boundary.
Partitioning	Partitioning refers to the high-level idea of dividing your data into so called partitions and distributing the individual partitions among Workers. The splitting of data and pushing work to Workers is done by a Manager.
Remote partitioning	Remote partitioning is a specialized partitioning concept. It refers to the idea of distributing the partitions among multiple JVMs mainly by using a messaging middleware.
Manager node	The Manager node is one of the objects taking a huge part when using partitioning. The Manager node is responsible for dividing the dataset into partitions and keeping track of all the divided partitions' Worker execution. When all Workers nodes are done with their partitions, the Manager will mark the corresponding Job as completed.
Worker node	A Worker node is the other important party in the context of partitioning. The Worker node is the one executing the work needed for a single partition.

Batch jobs in Fineract

Types of jobs

The jobs in Fineract can be divided into 2 categories:

- Normal batch jobs
- Partitionable batch jobs

Most of the jobs are normal batch jobs with limited scalability because Fineract is still passing through the evolution on making most of them capable to process a high-volume of data.

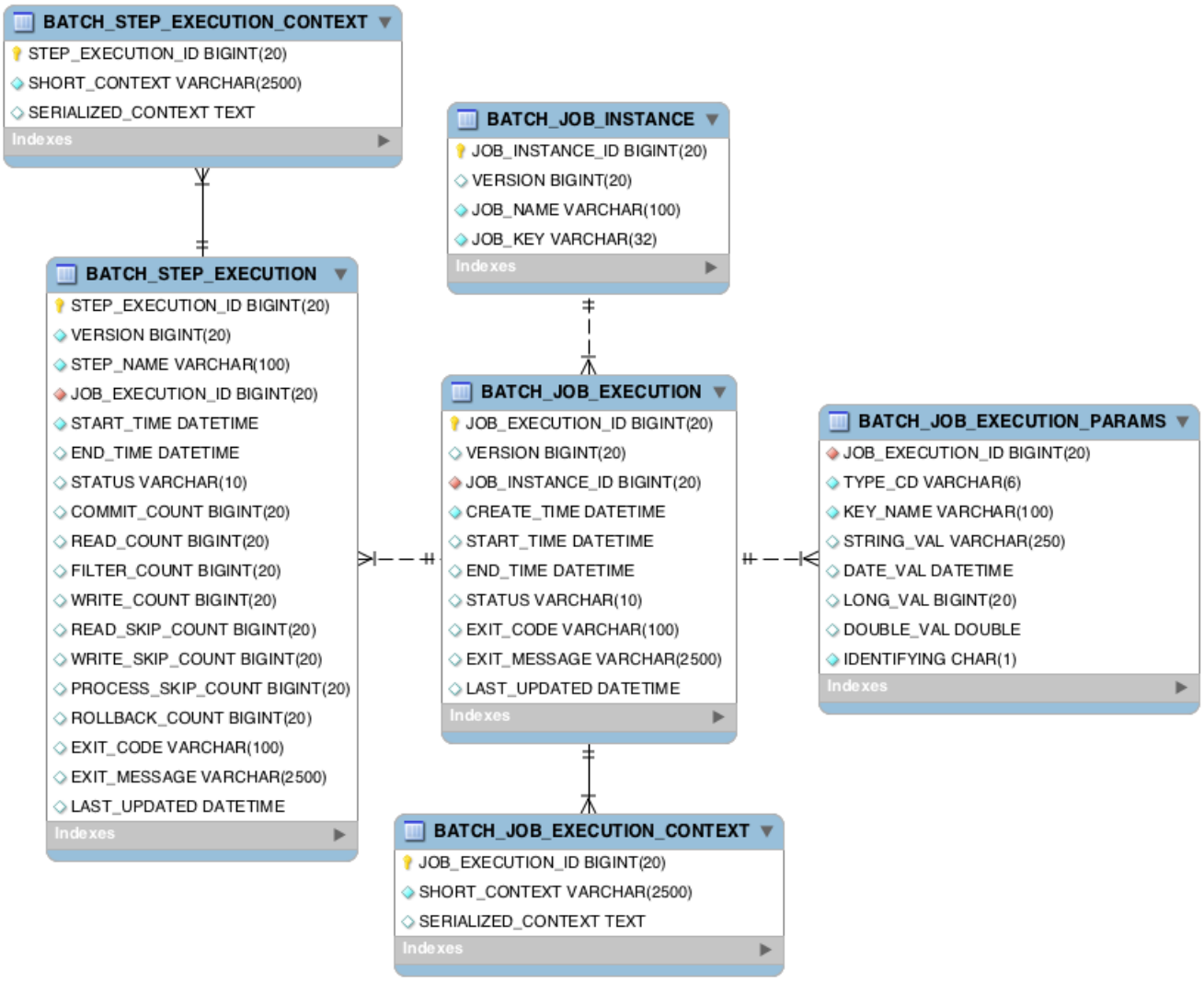
List of jobs

Job name	Active by default	Partitionable	Description
LOAN_CLOSE_OF_BUSINESS	No	Yes	TBD

Batch job execution

State management

State management for the batch jobs is done by the Spring Batch provided state management. The data model consists of the following database structure:

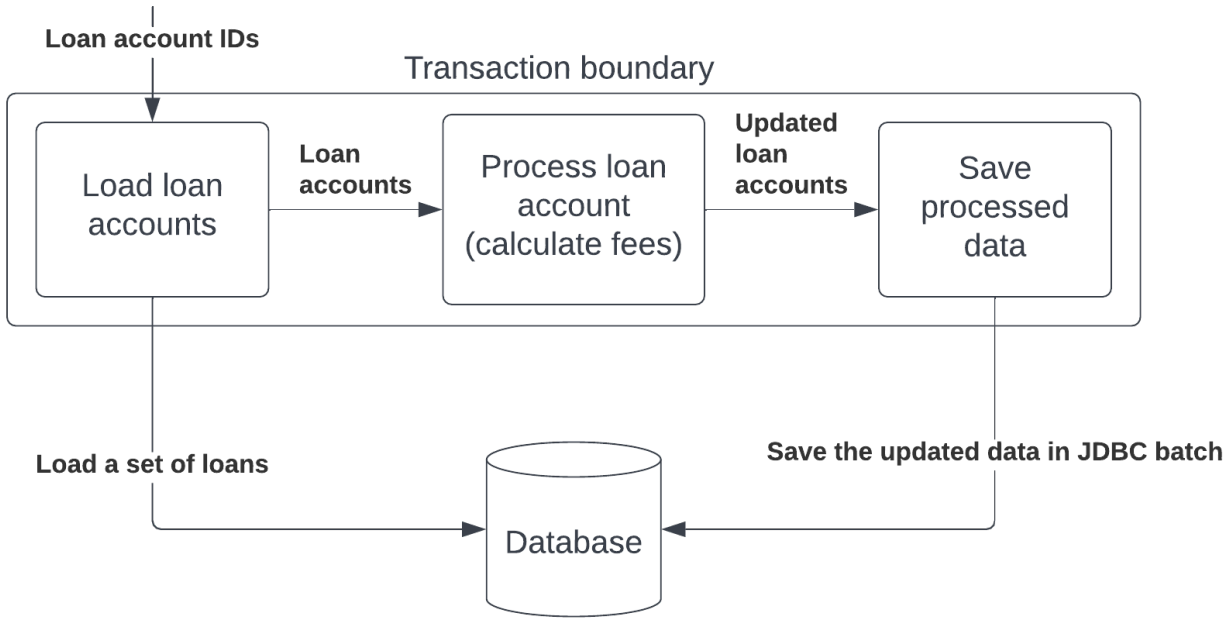


The corresponding database migration scripts are shipped with the Spring Batch core module under the `org.springframework.batch.core` package. They are only available as native scripts and are named as `schema-.sql` where `` is the short name of the database platform. For MySQL it's called `schema-mysql.sql` and for PostgreSQL it's called `schema-postgresql.sql`. When Fineract is started, the database dependent schema SQL script will be picked up according to the datasource configurations.

Chunk oriented processing

Chunking data has not been easier. Spring Batch does a really good job at providing this capability.

In order to save resources when starting/committing/rollbacking transactions for every single processed item, chunking shall be used. That way, it's possible to mark the transaction boundaries for a single processed chunk instead of a single item processing. The image below describes the flow with a very simplistic example.



In addition to not opening a lot of transactions, the processing could also benefit from JDBC batching. The last step - writing the result into the database - collects all the processed items and then writes it to the database; both for MySQL and PostgreSQL (the databases supported by Fineract) are capable of grouping multiple DML (INSERT/UPDATE/DELETE) statements and sending them in one round-trip, optimizing the data being sent over the network and granting the possibility to the underlying database engine to enhance the processing.

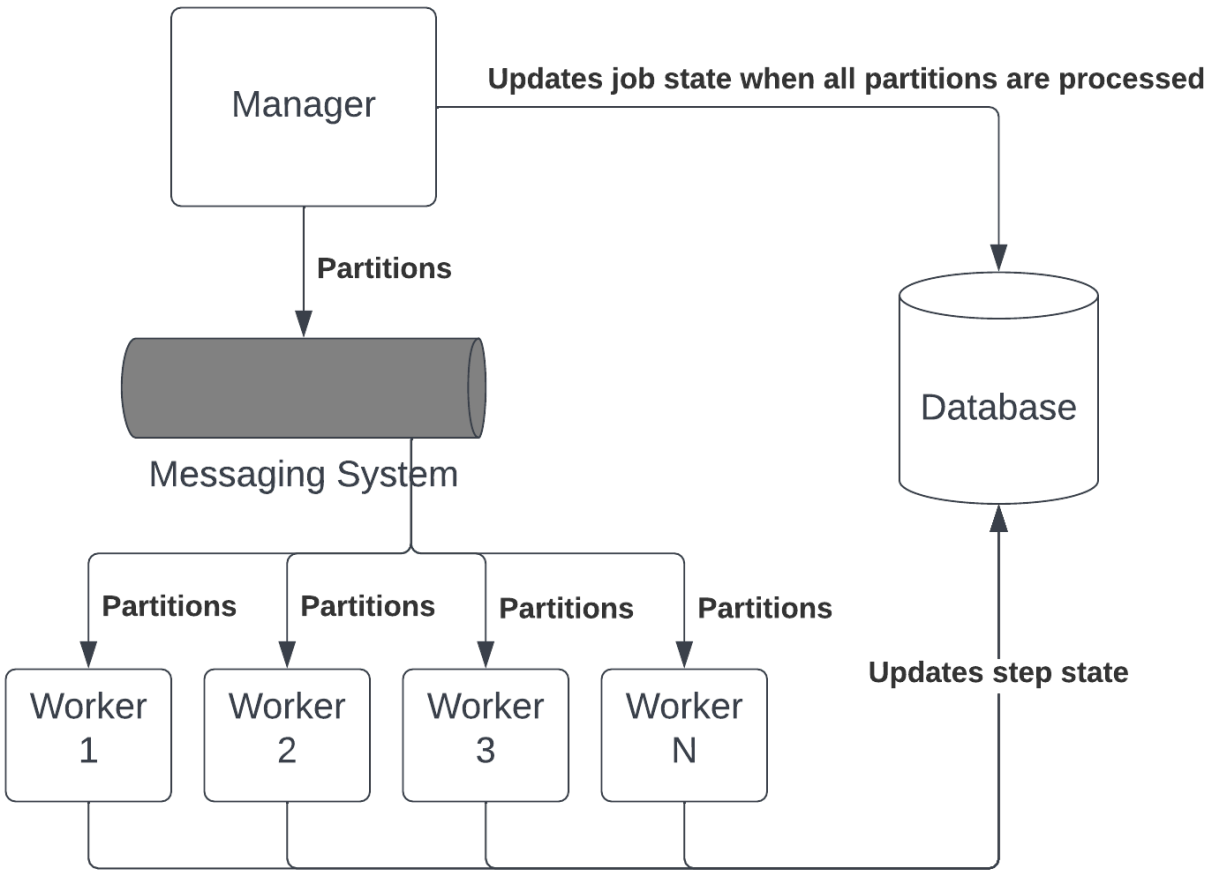
Remote partitioning

Spring Batch provides a really nice way to do remote partitioning. The 2 type of objects in this setup is a manager node - who splits and distributes the work - and a number of worker nodes - who picks up the work.

In remote partitioning, the worker instances are receiving the work via a messaging system as soon as the manager splits up the work into smaller pieces.

Remote partitioning could be done 2 ways in terms of keeping the job state up-to-date. The main difference between the two is how the manager is notified about partition completions.

One way is that they share the same database. When the worker does something to a partition - for example picks it up for processing - it updates the state of that partition in the database. In the meantime, the manager regularly polls the database until all partitions are processed. This is visualized in the below diagram.



An alternative approach to this - when the database is not intended to be shared between manager and workers - is to use a messaging system (could be the same as for distributing the work) and the workers could send back a message to the manager instance, therefore notifying it about failure/completion. Then the manager can simply keep the database state up-to-date.

Even though the alternative solution decouples the workers even better, we thought it's not necessary to add the complexity of handling reply message channel to the manager.

Also, please note that the partitioned job execution is multitenant meaning that the workers will receive which tenant it should do the processing for.

Supported message channels

For remote partitioning, the following message channels are supported by Fineract:

- Any JMS compatible message channels (ActiveMQ, Amazon MQ, etc)

Fault-tolerance scenarios

There are multiple fault tolerance use-cases that this solution must and will support:

1. If the manager fails during partitioning
2. If the manager completes the partitioning and the partition messages are sent to the broker but while the manager is waiting for the workers to finish, the manager fails

3. If the manager runs properly and during a partition processing a worker instance fails

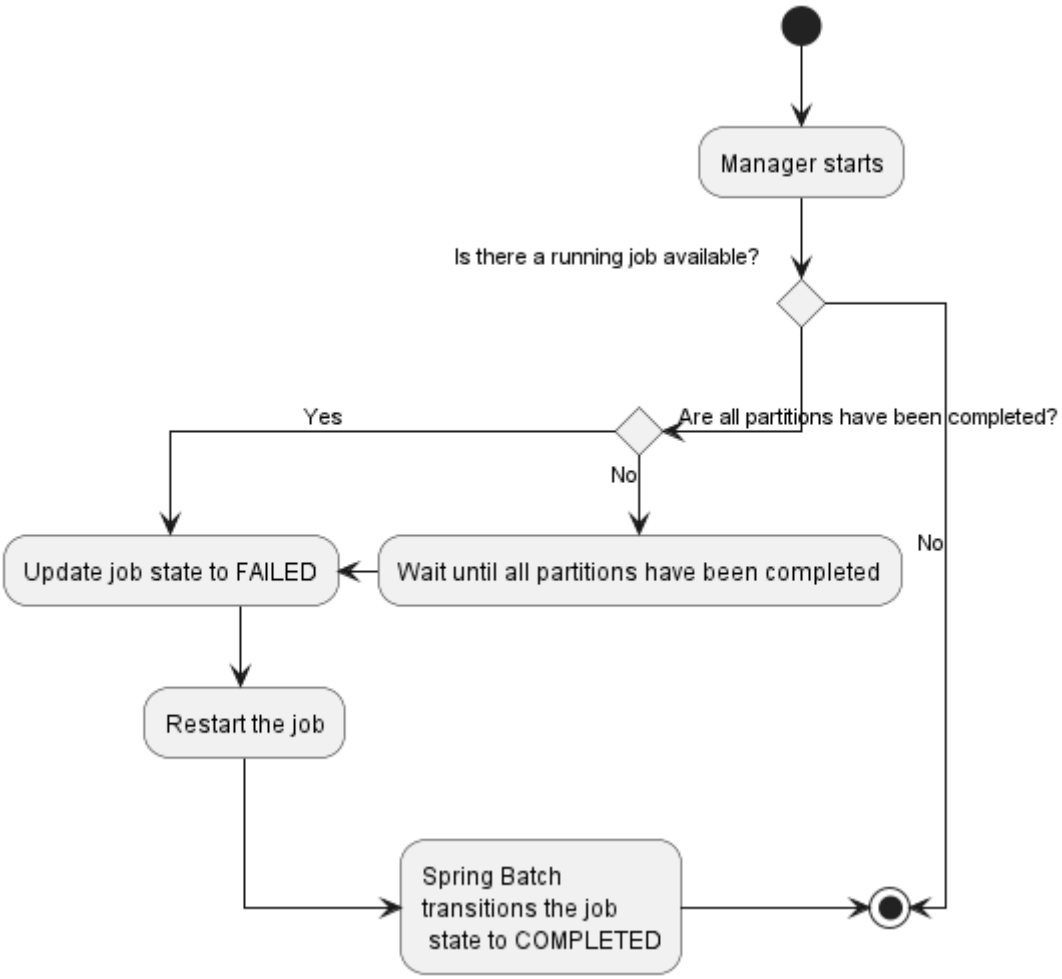
In case of scenario 1), the simple solution is to re-trigger the job via API or via the Quartz scheduler.

In case of scenario 2), there's no out-of-the-box solution by Spring Batch. Although there's a custom mechanism in place that'll resume the job upon restarting the manager. There are 2 cases in the context of this scenario:

- If all the partitions have been successfully processed by workers
- If not all the partitions have been processed by the workers

In the first case, we'll simply mark the stuck job as `FAILED` along with it's partitioning step and instruct Spring Batch to restart the job. The behavior in this case will be that Spring Batch will spawn a new job execution but will notice that the partitions have all been completed so it's not going to execute them once more.

In the latter case, the same will happen as for the first one but before marking the job execution as `FAILED`, we'll wait until all partitions have been completed.



In case of scenario 3), another worker instance will take over the partition since it hasn't been finished.

Configurable batch jobs

There's another type of distinction on the batch jobs. Some of them are configurable in terms of their behavior.

The currently supported configurable batch jobs are the following:

- LOAN_CLOSE_OF_BUSINESS

The behavior of these batch jobs are configurable. There's a new terminology we're introducing called **business steps**.

Business steps

Business steps are a smaller unit of work than regular Spring Batch Steps and the two are not meant to be mixed up because there's a large difference between them.

A Spring Batch Step's main purpose is to decompose a bigger work into smaller ones and making sure that these smaller Steps are properly handled within a single database transaction.

In case of a business step, it's a smaller unit of work. Business steps live **within** a Spring Batch Step. Fundamentally, they are simple classes that are implementing an interface with a single method that contains the business logic.

Here's a very simple example:

```
public class MyCustomBusinessStep implements BusinessStep<Loan> {
    @Override
    public Loan process(Loan loan) {
        // do something
    }
}
```

```
public class LoanCOBItemProcessor implements ItemProcessor<Loan, Loan> {
    @Override
    public Loan process(Loan loan) {
        List<BusinessStep<Loan>> bSteps = getBusinessSteps();
        Loan result = loan;
        for (BusinessStep<Loan> bStep : bSteps) {
            result = bStep.process(result);
        }
        return result;
    }
}
```

Business step configuration

The business steps are configurable for certain jobs. The reason for that is because we want to allow the possibility for Fineract users to configure their very own business logic for generic jobs, like the Loan Close Of Business job where we want to do a formal "closing" of the loans at the end of the day.

All countries are different with a different set of regulations. However in terms of behavior, there's no all size fits all for loan closing.

For example in the United States of America, you might need the following logic for a day closing:

1. Close fully repaid loan accounts
2. Apply penalties
3. Invoke IRS API for regulatory purposes

While in Germany it should be:

1. Close fully repaid loan accounts
2. Apply penalties
3. Do some fraud detection on the account using an external service
4. Invoke local tax authority API for regulatory purposes

These are just examples, but you get the idea.

The business steps are configurable through APIs:

Retrieving the configuration for a job:

```
GET /fineract-provider/api/v1/jobs/{jobName}/steps?tenantIdentifier={tenantId}
HTTP 200

{
  "jobName": "LOAN_CLOSE_OF_BUSINESS",
  "businessSteps": [
    {
      "stepName": "APPLY_PENALTY_FOR_OVERDUE_LOANS",
      "order": 1
    },
    {
      "stepName": "LOAN_TAGGING",
      "order": 2
    }
  ]
}
```

Updating the business step configuration for a job:

```
PUT /fineract-provider/api/v1/jobs/{jobName}/steps?tenantIdentifier={tenantId}

{
  "businessSteps": [
    {
      "stepName": "LOAN_TAGGING",
      "order": 1
    },
    {
      "stepName": "APPLY_PENALTY_FOR_OVERDUE_LOANS",
      "order": 2
    }
  ]
}
```

The business step configuration for jobs are tracked within the database in the `m_batch_business_steps` table.