Table of Contents

Appendix A: Meta-Data Schema

A.1. Overview

The Spring Batch Metadata tables closely match the Domain objects that represent them in Java. For example, `JobInstance`, `JobExecution`, `JobParameters`, and `StepExecution` map to `BATCH_JOB_INSTANCE`, `BATCH_JOB_EXECUTION`, `BATCH_JOB_EXECUTION_PARAMS`, and `BATCH_STEP_EXECUTION`, respectively. `ExecutionContext` maps to both `BATCH_JOB_EXECUTION_CONTEXT` and `BATCH_STEP_EXECUTION_CONTEXT`. The `JobRepository` is responsible for saving and storing each Java object into its correct table. This appendix describes the metadata tables in detail, along with many of the design decisions that were made when creating them. When viewing the various table creation statements below, it is important to realize that the data types used are as generic as possible. Spring Batch provides many schemas as examples, all of which have varying data types, due to variations in how individual database vendors handle data types. The following image shows an ERD model of all 6 tables and their relationships to one another:

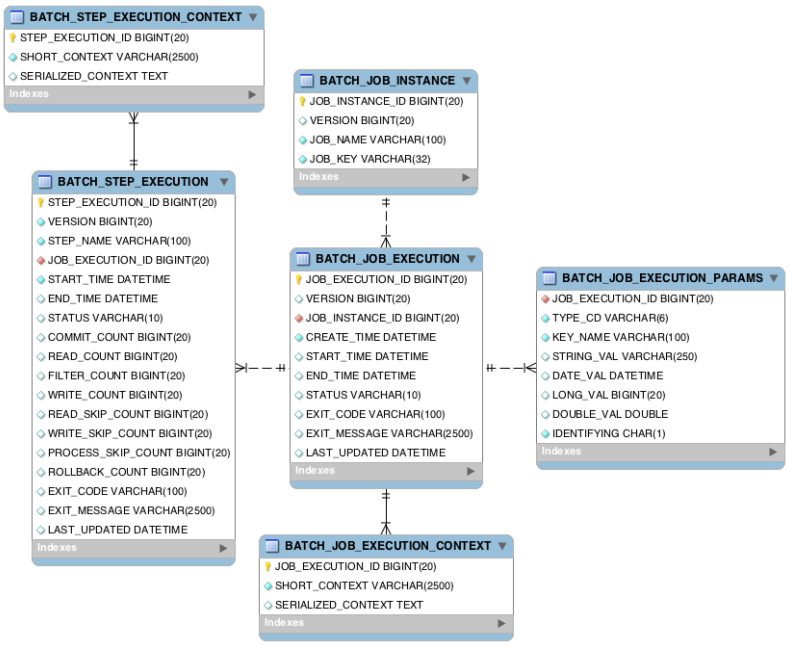

Figure 1. Spring Batch Meta-Data ERD

A.1.1. Example DDL Scripts

The Spring Batch Core JAR file contains example scripts to create the relational tables for a number of database platforms (which are, in turn, auto-detected by the job repository factory bean or namespace equivalent). These scripts can be used as is or modified with additional indexes and constraints as desired. The file names are in the form `schema-*.sql`, where "*" is the short name of the target database platform. The scripts are in the package `org.springframework.batch.core`.

A.1.2. Migration DDL Scripts

Spring Batch provides migration DDL scripts that you need to execute when you upgrade versions. These scripts can be found in the Core Jar file under `org/springframework/batch/core/migration`. Migration scripts are organized into folders corresponding to version numbers in which they were

introduced:

- 2.2: contains scripts needed if you are migrating from a version before 2.2 to version 2.2
- 4.1: contains scripts needed if you are migrating from a version before 4.1 to version 4.1

A.1.3. Version

Many of the database tables discussed in this appendix contain a version column. This column is important because Spring Batch employs an optimistic locking strategy when dealing with updates to the database. This means that each time a record is 'touched' (updated) the value in the version column is incremented by one. When the repository goes back to save the value, if the version number has changed it throws an OptimisticLockingFailureException, indicating there has been an error with concurrent access. This check is necessary, since, even though different batch jobs may be running in different machines, they all use the same database tables.

A.1.4. Identity

BATCH_JOB_INSTANCE, BATCH_JOB_EXECUTION, and BATCH_STEP_EXECUTION each contain columns ending in _ID. These fields act as primary keys for their respective tables. However, they are not database generated keys. Rather, they are generated by separate sequences. This is necessary because, after inserting one of the domain objects into the database, the key it is given needs to be set on the actual object so that they can be uniquely identified in Java. Newer database drivers (JDBC 3.0 and up) support this feature with database-generated keys. However, rather than require that feature, sequences are used. Each variation of the schema contains some form of the following statements:

```
CREATE SEQUENCE BATCH_STEP_EXECUTION_SEQ;
CREATE SEQUENCE BATCH_JOB_EXECUTION_SEQ;
CREATE SEQUENCE BATCH_JOB_SEQ;
```

Many database vendors do not support sequences. In these cases, work-arounds are used, such as the following statements for MySQL:

```
CREATE TABLE BATCH_STEP_EXECUTION_SEQ (ID BIGINT NOT NULL) type=InnoDB;
INSERT INTO BATCH_STEP_EXECUTION_SEQ values(0);
CREATE TABLE BATCH_JOB_EXECUTION_SEQ (ID BIGINT NOT NULL) type=InnoDB;
INSERT INTO BATCH_JOB_EXECUTION_SEQ values(0);
CREATE TABLE BATCH_JOB_SEQ (ID BIGINT NOT NULL) type=InnoDB;
INSERT INTO BATCH_JOB_SEQ values(0);
```

In the preceding case, a table is used in place of each sequence. The Spring core class, MySQLMaxValueIncrementer, then increments the one column in this sequence in order to give similar functionality.

A.2. BATCH_JOB_INSTANCE

The BATCH_JOB_INSTANCE table holds all information relevant to a JobInstance, and serves as the top

of the overall hierarchy. The following generic DDL statement is used to create it:

```
CREATE TABLE BATCH_JOB_INSTANCE  (
  JOB_INSTANCE_ID BIGINT  PRIMARY KEY ,
  VERSION BIGINT,
  JOB_NAME VARCHAR(100) NOT NULL ,
  JOB_KEY VARCHAR(2500)
);
```

The following list describes each column in the table:

- JOB_INSTANCE_ID: The unique ID that identifies the instance. It is also the primary key. The value of this column should be obtainable by calling the getId method on JobInstance.
- VERSION: See Version.
- JOB_NAME: Name of the job obtained from the Job object. Because it is required to identify the instance, it must not be null.
- JOB_KEY: A serialization of the JobParameters that uniquely identifies separate instances of the same job from one another. (JobInstances with the same job name must have different JobParameters and, thus, different JOB_KEY values).

A.3. BATCH_JOB_EXECUTION_PARAMS

The BATCH_JOB_EXECUTION_PARAMS table holds all information relevant to the JobParameters object. It contains 0 or more key/value pairs passed to a Job and serves as a record of the parameters with which a job was run. For each parameter that contributes to the generation of a job's identity, the IDENTIFYING flag is set to true. Note that the table has been denormalized. Rather than creating a separate table for each type, there is one table with a column indicating the type, as shown in the following listing:

```
CREATE TABLE BATCH_JOB_EXECUTION_PARAMS  (
    JOB_EXECUTION_ID BIGINT NOT NULL ,
    TYPE_CD VARCHAR(6) NOT NULL ,
    KEY_NAME VARCHAR(100) NOT NULL ,
    STRING_VAL VARCHAR(250) ,
    DATE_VAL DATETIME DEFAULT NULL ,
    LONG_VAL BIGINT ,
    DOUBLE_VAL DOUBLE PRECISION ,
    IDENTIFYING CHAR(1) NOT NULL ,
    constraint JOB_EXEC_PARAMS_FK foreign key (JOB_EXECUTION_ID)
    references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
);
```

The following list describes each column:

- JOB_EXECUTION_ID: Foreign key from the BATCH_JOB_EXECUTION table that indicates the job execution to which the parameter entry belongs. Note that multiple rows (that is, key/value

pairs) may exist for each execution.

- TYPE_CD: String representation of the type of value stored, which can be a string, a date, a long, or a double. Because the type must be known, it cannot be null.
- KEY_NAME: The parameter key.
- STRING_VAL: Parameter value, if the type is string.
- DATE_VAL: Parameter value, if the type is date.
- LONG_VAL: Parameter value, if the type is long.
- DOUBLE_VAL: Parameter value, if the type is double.
- IDENTIFYING: Flag indicating whether the parameter contributed to the identity of the related `JobInstance`.

Note that there is no primary key for this table. This is because the framework has no use for one and, thus, does not require it. If need be, you can add a primary key may be added with a database generated key without causing any issues to the framework itself.

A.4. BATCH_JOB_EXECUTION

The `BATCH_JOB_EXECUTION` table holds all information relevant to the `JobExecution` object. Every time a `Job` is run, there is always a new `JobExecution`, and a new row in this table. The following listing shows the definition of the `BATCH_JOB_EXECUTION` table:

```
CREATE TABLE BATCH_JOB_EXECUTION  (
  JOB_EXECUTION_ID BIGINT  PRIMARY KEY ,
  VERSION BIGINT,
  JOB_INSTANCE_ID BIGINT NOT NULL,
  CREATE_TIME TIMESTAMP NOT NULL,
  START_TIME TIMESTAMP DEFAULT NULL,
  END_TIME TIMESTAMP DEFAULT NULL,
  STATUS VARCHAR(10),
  EXIT_CODE VARCHAR(20),
  EXIT_MESSAGE VARCHAR(2500),
  LAST_UPDATED TIMESTAMP,
  JOB_CONFIGURATION_LOCATION VARCHAR(2500) NULL,
  constraint JOB_INSTANCE_EXECUTION_FK foreign key (JOB_INSTANCE_ID)
  references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
) ;
```

The following list describes each column:

- `JOB_EXECUTION_ID`: Primary key that uniquely identifies this execution. The value of this column is obtainable by calling the `getId` method of the `JobExecution` object.
- `VERSION`: See Version.
- `JOB_INSTANCE_ID`: Foreign key from the `BATCH_JOB_INSTANCE` table. It indicates the instance to which this execution belongs. There may be more than one execution per instance.

- CREATE_TIME: Timestamp representing the time when the execution was created.
- START_TIME: Timestamp representing the time when the execution was started.
- END_TIME: Timestamp representing the time when the execution finished, regardless of success or failure. An empty value in this column when the job is not currently running indicates that there has been some type of error and the framework was unable to perform a last save before failing.
- STATUS: Character string representing the status of the execution. This may be COMPLETED, STARTED, and others. The object representation of this column is the BatchStatus enumeration.
- EXIT_CODE: Character string representing the exit code of the execution. In the case of a command-line job, this may be converted into a number.
- EXIT_MESSAGE: Character string representing a more detailed description of how the job exited. In the case of failure, this might include as much of the stack trace as is possible.
- LAST_UPDATED: Timestamp representing the last time this execution was persisted.

A.5. BATCH_STEP_EXECUTION

The BATCH_STEP_EXECUTION table holds all information relevant to the StepExecution object. This table is similar in many ways to the BATCH_JOB_EXECUTION table, and there is always at least one entry per Step for each JobExecution created. The following listing shows the definition of the BATCH_STEP_EXECUTION table:

```
CREATE TABLE BATCH_STEP_EXECUTION  (
  STEP_EXECUTION_ID BIGINT  PRIMARY KEY ,
  VERSION BIGINT NOT NULL,
  STEP_NAME VARCHAR(100) NOT NULL,
  JOB_EXECUTION_ID BIGINT NOT NULL,
  START_TIME TIMESTAMP NOT NULL ,
  END_TIME TIMESTAMP DEFAULT NULL,
  STATUS VARCHAR(10),
  COMMIT_COUNT BIGINT ,
  READ_COUNT BIGINT ,
  FILTER_COUNT BIGINT ,
  WRITE_COUNT BIGINT ,
  READ_SKIP_COUNT BIGINT ,
  WRITE_SKIP_COUNT BIGINT ,
  PROCESS_SKIP_COUNT BIGINT ,
  ROLLBACK_COUNT BIGINT ,
  EXIT_CODE VARCHAR(20) ,
  EXIT_MESSAGE VARCHAR(2500) ,
  LAST_UPDATED TIMESTAMP,
  constraint JOB_EXECUTION_STEP_FK foreign key (JOB_EXECUTION_ID)
  references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;
```

The following list describes for each column:

- STEP_EXECUTION_ID: Primary key that uniquely identifies this execution. The value of this column should be obtainable by calling the getId method of the StepExecution object.
- VERSION: See Version.
- STEP_NAME: The name of the step to which this execution belongs.
- JOB_EXECUTION_ID: Foreign key from the BATCH_JOB_EXECUTION table. It indicates the JobExecution to which this StepExecution belongs. There may be only one StepExecution for a given JobExecution for a given Step name.
- START_TIME: Timestamp representing the time when the execution was started.
- END_TIME: Timestamp representing the time the when execution was finished, regardless of success or failure. An empty value in this column, even though the job is not currently running, indicates that there has been some type of error and the framework was unable to perform a last save before failing.
- STATUS: Character string representing the status of the execution. This may be COMPLETED, STARTED, and others. The object representation of this column is the BatchStatus enumeration.
- COMMIT_COUNT: The number of times in which the step has committed a transaction during this execution.
- READ_COUNT: The number of items read during this execution.
- FILTER_COUNT: The number of items filtered out of this execution.
- WRITE_COUNT: The number of items written and committed during this execution.
- READ_SKIP_COUNT: The number of items skipped on read during this execution.
- WRITE_SKIP_COUNT: The number of items skipped on write during this execution.
- PROCESS_SKIP_COUNT: The number of items skipped during processing during this execution.
- ROLLBACK_COUNT: The number of rollbacks during this execution. Note that this count includes each time rollback occurs, including rollbacks for retry and those in the skip recovery procedure.
- EXIT_CODE: Character string representing the exit code of the execution. In the case of a command-line job, this may be converted into a number.
- EXIT_MESSAGE: Character string representing a more detailed description of how the job exited. In the case of failure, this might include as much of the stack trace as is possible.
- LAST_UPDATED: Timestamp representing the last time this execution was persisted.

A.6. BATCH_JOB_EXECUTION_CONTEXT

The BATCH_JOB_EXECUTION_CONTEXT table holds all information relevant to the ExecutionContext of a Job. There is exactly one Job ExecutionContext per JobExecution, and it contains all of the job-level data that is needed for a particular job execution. This data typically represents the state that must be retrieved after a failure, so that a JobInstance can "start from where it left off". The following listing shows the definition of the BATCH_JOB_EXECUTION_CONTEXT table:

```
CREATE TABLE BATCH_JOB_EXECUTION_CONTEXT  (
  JOB_EXECUTION_ID BIGINT PRIMARY KEY,
  SHORT_CONTEXT VARCHAR(2500) NOT NULL,
  SERIALIZED_CONTEXT CLOB,
  constraint JOB_EXEC_CTX_FK foreign key (JOB_EXECUTION_ID)
  references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
) ;
```

The following list describes each column:

- JOB_EXECUTION_ID: Foreign key representing the JobExecution to which the context belongs. There may be more than one row associated with a given execution.
- SHORT_CONTEXT: A string version of the SERIALIZED_CONTEXT.
- SERIALIZED_CONTEXT: The entire context, serialized.

A.7. BATCH_STEP_EXECUTION_CONTEXT

The BATCH_STEP_EXECUTION_CONTEXT table holds all information relevant to the ExecutionContext of a Step. There is exactly one ExecutionContext per StepExecution, and it contains all of the data that needs to be persisted for a particular step execution. This data typically represents the state that must be retrieved after a failure, so that a JobInstance can 'start from where it left off'. The following listing shows the definition of the BATCH_STEP_EXECUTION_CONTEXT table:

```
CREATE TABLE BATCH_STEP_EXECUTION_CONTEXT  (
  STEP_EXECUTION_ID BIGINT PRIMARY KEY,
  SHORT_CONTEXT VARCHAR(2500) NOT NULL,
  SERIALIZED_CONTEXT CLOB,
  constraint STEP_EXEC_CTX_FK foreign key (STEP_EXECUTION_ID)
  references BATCH_STEP_EXECUTION(STEP_EXECUTION_ID)
) ;
```

The following list describes each column:

- STEP_EXECUTION_ID: Foreign key representing the StepExecution to which the context belongs. There may be more than one row associated to a given execution.
- SHORT_CONTEXT: A string version of the SERIALIZED_CONTEXT.
- SERIALIZED_CONTEXT: The entire context, serialized.

A.8. Archiving

Because there are entries in multiple tables every time a batch job is run, it is common to create an archive strategy for the metadata tables. The tables themselves are designed to show a record of what happened in the past and generally do not affect the run of any job, with a few notable exceptions pertaining to restart:

- The framework uses the metadata tables to determine whether a particular `JobInstance` has been run before. If it has been run and if the job is not restartable, then an exception is thrown.
- If an entry for a `JobInstance` is removed without having completed successfully, the framework thinks that the job is new rather than a restart.
- If a job is restarted, the framework uses any data that has been persisted to the `ExecutionContext` to restore the `Job's` state. Therefore, removing any entries from this table for jobs that have not completed successfully prevents them from starting at the correct point if run again.

A.9. International and Multi-byte Characters

If you are using multi-byte character sets (such as Chinese or Cyrillic) in your business processing, then those characters might need to be persisted in the Spring Batch schema. Many users find that simply changing the schema to double the length of the `VARCHAR` columns is enough. Others prefer to configure the JobRepository with `max-varchar-length` half the value of the `VARCHAR` column length. Some users have also reported that they use `NVARCHAR` in place of `VARCHAR` in their schema definitions. The best result depends on the database platform and the way the database server has been configured locally.

A.10. Recommendations for Indexing Meta Data Tables

Spring Batch provides DDL samples for the metadata tables in the core jar file for several common database platforms. Index declarations are not included in that DDL, because there are too many variations in how users may want to index, depending on their precise platform, local conventions, and the business requirements of how the jobs are operated. The following below provides some indication as to which columns are going to be used in a `WHERE` clause by the DAO implementations provided by Spring Batch and how frequently they might be used, so that individual projects can make up their own minds about indexing:

Table 1. Where clauses in SQL statements (excluding primary keys) and their approximate frequency of use.

Default Table Name	Where Clause	Frequency
BATCH_JOB_INSTANCE	JOB_NAME = ? and JOB_KEY = ?	Every time a job is launched
BATCH_JOB_EXECUTION	JOB_INSTANCE_ID = ?	Every time a job is restarted
BATCH_EXECUTION_CONTEXT	EXECUTION_ID = ? and KEY_NAME = ?	On commit interval, a.k.a. chunk
BATCH_STEP_EXECUTION	VERSION = ?	On commit interval, a.k.a. chunk (and at start and end of step)
BATCH_STEP_EXECUTION	STEP_NAME = ? and JOB_EXECUTION_ID = ?	Before each step execution