WebSphere Application Server

WebSphere Liberty Batch: Configuring the Job Repository for DB2 on z/OS

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102716** under the category of "White Papers"

Version Date: November 1, 2017
See "Document change history" on page 13 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software
Written by:

David Follis
IBM Poughkeepsie
845-435-5462
follis@us.ibm.com

Robert Catterall
IBM Atlanta
404-348-5279
rfcatter@us.ibm.com

Don Bagwell
IBM Advanced Technical Sales
301-240-3016
dbagwell@us.ibm.com

Many thanks go to the whole batch team

Contents

Introduction

The open standard for Java Batch (JSR-352) requires something called a Job Repository. Here is what the specification says about the repository (Section 7.4):

> A job repository holds information about jobs currently running and jobs that have run in the past. The JobOperator interface provides access to this repository. The repository contains job instances, job executions, and step executions. For further information on this content, see sections 10.9.8, 10.9.9, 10.9.10, respectively.
>
> Note the implementation of the job repository is outside the scope of this specification.

That last sentence is important. The specification says nothing about how the repository is implemented. This means that any implementation has to explain how it is implemented and how it is managed.

WebSphere Liberty provides two different implementations. The first, default, implementation is the in-memory repository. This is handy for a development environment, in which one generally does not care about the results of previous jobs or otherwise need to remember associated information. When the objective is to just get an application to run, having the batch environment forget everything when the server is restarted is probably a good thing.

The other implementation is more appropriate for a production environment. It uses a database to keep track of all the information the batch runtime needs to properly manage batch jobs. That being the case, the database will need tables that the runtime will use. The intent of this paper is to provide information on creating those tables and their purpose, and to describe some customization one might (or might not) want to consider.

Configuring the server

In order for the batch runtime to keep things in the Job Repository, it has to know where it is. To try to make things easy for developers, the default behavior is to use an in-memory Job Repository. That means that if one does nothing at all then the in-memory Repository will be used. That simplifies things for developers – the Repository works in a way that offers simplicity in use and management.

As work progresses into testing and pre-production, it might be desirable to have an actual database without the need to set things up in DB2 for z/OS. For that, Liberty Batch supports Derby, which is relatively easy to set up and use.

For a production environment the default behavior will not be what one wants. In production, it is important to configure exactly what is wanted, as opposed to simply trying to get something working quickly.

Configuring a server so that it knows where the Job Repository resides begins with the batchPersistence element in server.xml. Inside that element, one specifies the jobStoreRef attribute. The attribute is set to the ID value of a databaseStore element that has information about the database to use. This includes the schema and tablePrefix values. It also has a dataSourceRef that is set to the ID of a dataSource element with information about how to access the actual datasource (the database itself). Here is an example that has been used to connect to DB2 for z/OS.

```
<batchPersistence jobStoreRef="BatchDatabaseStore" />

<databaseStore id="BatchDatabaseStore"
               createTables="false"
               dataSourceRef="batchDB"
               schema="JBATCH"
               tablePrefix="" />


<dataSource    id="batchDB"
               containerAuthDataRef="batchAlias"
               type="javax.sql.XADataSource"
               jdbcDriverRef="DB2T4">
     <properties.db2.jcc
                    serverName="wg31.washington.ibm.com"
                    portNumber="2446"
                    databaseName="DSN2LOC"
                    driverType="4"
     />
</dataSource>

<jdbcDriver id="DB2T4" libraryRef="DB2T4LibRef" />

<library id="DB2T4LibRef">
     <fileset  dir="/shared/db21210/jdbc/classes/"
               includes="db2jcc4.jar db2jcc_license_cisuz.jar
                         sqlj4.zip" />
</library>

<authData id="batchAlias" user="XXXXXX" password="XXXXXX" />
```

As one can see, the batchPersistence element points to the databaseStore named BatchDatabaseStore. That, in turn, points to the dataSource called batchDB. The rest of the information supports setting up a Type-4 connection to the DB2 for z/OS instance.

The batch runtime issues a message indicating whether the in-memory or the database option is being used for persistence. When using a database, batch uses Java Persistence API (or JPA) to interact with the database. The batch runtime does not know the specifics of the database actually being used - only that it is being accessed through JPA; thus, the message appears as follows when a database is used:

```
    CWWKY0008I: The batch feature is using persistence type JPA.
```

And when the in-memory Repository is used, the message below is issued:

```
    CWWKY0008I: The batch feature is using persistence type In-
    Memory.
```

In a production environment, ensure that the in-memory repository option is not being used.

Creating the DDL

In examining the sample configuration above, one might notice an attribute called createTables that is part of the databaseStore element. When set to true (the default) JPA will automatically create the needed tables for batch if it does not find them the first time it tries to access them. For a developer who wants to use a true database instead of the in-memory Repository, this is very helpful – it eliminates the need to set up tables. All that is needed is to set up server.xml properly. That is probably the preferred approach if Derby is used for Job Repository persistence in a test environment.

For production, on the other hand, one might want to be a bit more careful and actually perform the table set-up work. Doing that requires the right DDL to define the tables. JPA can generate the appropriate DDL for the database. The JPA code knows the table structure and, by examining the server.xml contents, it knows the database to which programs will connect. Using that information, it can generate the right DDL syntax for the database.

To do this, first enable the JMX local connector feature (`localConnector-1.0`) in the server. Then, run the `ddlGen` utility found in the Liberty `bin` directory. Specify ‘generate’ and the server name as a parameter,as shown below:

```
    ./ddlGen generate JSRDISP
```

It is assumed here that Java is on the path and WLP_USER_DIR is set appropriately.

A message (`CWWKD0107I`) should be issued, providing information on where the generated DDL has been placed (in a directory called ‘`ddl’` under the server configuration).

A quick look at the tables

As of this writing the ddlGen will create five tables and assorted indexes. In this section we will take a quick look at what is in those tables and how they are used by the batch runtime. This is not intended to be complete documentation, just an overview. New tables might be added in the future which might not be covered here.

The key to everything is the Job Instance table. There is a row in this table for every job that has been submitted. Key information about the job is retained, along with the state of the instance.

Every execution of a job instance is represented by a row in the Job Execution table. Remember that if a job is submitted, fails, and is restarted, it gets one instance and two executions (one for each attempt to run the job). Starting a job creates one instance and one execution. Restarting a failed job creates another execution for the already-existing job instance. The Job Execution table contains information about each execution.

The Step Instance table contains a row for each step in each job executed and contains information that may be used if the job is restarted. This includes checkpoint information used by a chunk step to help it pick up where it left off if the job failed part way through processing a chunk.

The Step Execution table also contains a row for each step in each job executed. It contains detailed information about the step's execution including metrics about records read, checkpoints taken, and records skipped. It also holds the exit status value for the step.

The Job Parameters table holds the parameters provided when jobs were submitted. Each parameter provided on any job gets a separate row. This makes it easy to find jobs submitted with particular parameters (e.g., an application name).

Other tables may be added in the future to enable additional functionality. New tables (or new columns in existing tables) are always optional, but new functions will not work if they require new tables and/or columns that are not present in the database.

Customizing the generated DDL for DB2 on z/OS

In looking over the generated DDL and preparing to run it to create the Job Repository tables, one might think of some possible customizations that could be effected. Some things, like table and column names, could not be changed without breaking the batch runtime code that expects to use them; there are, however, some other modifications that could be performed.

The following sections describe some considerations regarding a few things one might (or might not) want to change with respect to the Job Repository DDL.

Creating DB2 databases explicitly

The generated DDL does not explicitly create databases, therefore databases will be dynamically created by Db2, with Db2-generated database names. Historically, some Db2 for z/OS-using organizations have explicitly created databases with names that follow a certain convention, but this approach is often not needed in a modern Db2 environment. "Housekeeping" utilities, such as REORG and COPY, will ideally be executed on a needs-based, per-table and per-index basis, with no dependency on database names. Some organizations manage the Db2 DBADM database administrator privilege by database name, but system DBADM authority, introduced with Db2 10, makes it unnecessary to grant DBADM authority on a database-by-database basis.

Creating DB2 table spaces explicitly

The generated DDL creates the required tables but does not explicitly create associated table spaces. One could manually alter the DDL to explicitly create table spaces for the tables, if desired (this might be done, for example, to follow a certain table space naming convention).

Table spaces implicitly created by DB2 will be universal partition-by-growth table spaces, with LOCKSIZE(ROW).

Note that when DB32 implicitly creates table spaces it will also create any required UNIQUE indexes. Choosing to explicitly define table spaces will require manual creation of UNIQUE indexes for some of the Job Repository tables. Failure to do so will result in -540 SQL codes when attempting to define the tables.

Adjusting lock granularity

DB2 for z/OS supports several levels of lock granularity for table spaces: row, page, etc. As noted above, table spaces implicitly created by DB2 will utilize row-level locking. In some cases, row level locking can increase CPU overhead as compared to page-level locking, but switching to page-level locking could lead to lock contention and a reduction in throughput.

Generally speaking, in a non-data sharing environment (i.e., for a stand-alone DB2 subsystem), the CPU cost of row-level locking will be very close to that of page-level locking. The CPU cost of row-level locking is higher in a DB2 data sharing environment (i.e., when DB2 is operating in data sharing mode on a Parallel Sysplex), but the impact on the overall CPU consumption of a DB2 workload will likely be quite small if row-level locking is used only for a few tables in the system.

Inlining LOB values

There are four BLOB (binary large object) columns in the Job Repository tables. One is used to contain the JSL (the XML that specifies the job flow, much like JCL for traditional z/OS batch). Another is used to contain checkpoint data provided by application code at checkpoints in chunk step processing. The third is used to contain job parameters specified when the job was submitted (these are kept in a single JSON string, stored as a BLOB in DB2). The fourth BLOB column contains persistent user data utilized by the application.

These BLOB columns are all defined with the same maximum size: 64,000 bytes. Could that maximum-length specification be made smaller? Perhaps. That would require knowing the maximum size for application use in a given installation. If the wrong value is chosen, it might be necessary to alter the table in the future to accommodate a new application that needs more space (after determining that the problem was the result of a too-small BLOB length limit). In fact, one might need to make the maximum length of one or more of the BLOB columns even large than the default size, if an application uses data that needs to occupy more space (user data is most likely to get rather large). This is a discussion to have with your application developers.

DB2 for z/OS offers a feature called LOB inlining. Normally the values in a LOB column are stored in a table space that is physically separate from the table space of the associated base table (though the LOB values appear to be in the base table from a logical perspective). If it is known that most of the values in a LOB column are relatively small – not more than a few thousand bytes in length,for example – then the definition of the column can be altered so that DB2 will "inline" a user-specified portion of each LOB value in the base table (if the length of a LOB value exceeds the user-specified inline limit, the portion of the LOB value beyond that limit will be stored in the LOB table space associated with the LOB column) . When a majority of a LOB column's values can be completely inlined in a base table, LOB inlining can improve performance for INSERT operations and for queries that return LOB data. On the other hand, LOB inlining can have a detrimental impact on the performance of queries that do not retrieve LOB values, because the inlined LOB data increases the length of base table rows, and that can reduce the effectiveness of the base table's buffer pool with regard to reducing disk read I/O activity. More information on LOB inlining, including guidance on determining whether or not inlining would be a good idea for a LOB column, can be found in a blog entry written by one of the co-authors of this document: http://robertsdb2blog.blogspot.com/2013/07/db2-for-zos-clearing-up-some-matters.html.

Allocating space for tables

If table spaces for the Job Repository tables are explicitly defined, primary and secondary disk space allocation quantities for the table spaces can be defined. That said, the recommendation is to let DB2 manage disk space allocation requests. In that case, the primary quantity will default to the value specified in the ZPARM parameter TSQTY, and secondary space requests will be managed by DB2 using what is known as the "sliding scale" algorithm (assuming that the ZPARM parameter MGEXTSZ is set to its default value of YES). Information about DB2-managed disk space allocation, including the sliding scale algorithm, can be found in the DB2 for z/OS Knowledge Center on the Web (see https://www.ibm.com/support/knowledgecenter/SSEPEK_11.0.0/admin/src/tpc/db2z_allocationsecondaryspace.html).

Assigning buffer pools

Table spaces implicitly created by DB2 for the Job Repository tables can be assigned to buffer pools by adding a BUFFERPOOL *bpname* specification to the CREATE TABLE statements.

Additional indexes

The generated DDL defines a few indexes that are needed or that have been found to be helpful in processing the workload in a well-performing manner. Additional indexes can be defined if it is determined that they would positively impact performance. The most likely case in which additional indexes could be helpful from a performance perspective would be in support of queries associated with batch REST API requests (or the Liberty Administration

Center Batch tool). For example, if the Admin Center will be used to view all jobs submitted by a particular submitter userid, an index on SUBMITTER in the Job Instance table could have a positive impact on performance.

Adding indexes to tables should be done after careful consideration of costs and benefits. Indexes can speed execution of certain queries (as “searched” UPDATE and DELETE statements), but they also consume additional disk space, increase the CPU cost of INSERT and DELETE statements (and UPDATEs affecting indexed columns), and increase the CPU cost of index-intensive utilities such as REORG, LOAD and RUNSTATS.

CURRENT RULES and ON DELETE rules

Because some of the Job Repository tables define referential constraints where the same table is parent and child an ON DELETE value of NO ACTION needs to be defined for proper behavior. Depending on the value of the DB2 for z/OS special register CURRENT RULES, this might be the default value or it might not. If it is not the default then a -633 SQL error will occur attempting to create the referential constraint. This can be corrected either by adding ON DELETE NO ACTION to the ALTER TABLE statement which fails or else by adding a statement to the DDL to set the CURRENT RULES register to 'STD'. Adding the ON DELETE NO ACTION is preferred.

Going forward – table maintenance

With the Job Repository tables defined, attention shifts to matters concerning table maintenance. In this section we will discuss some things to consider in this area going forward.

Purging jobs

The Job Repository keeps information about every job that has ever run in the environment from the day it was created. Eventually, there will be information about jobs that is no longer needed. Your organization might have a data retention policy that specifies how long information of the type in the Job Repository has to be kept in the database. Information of an age beyond what is needed or required would more than likely be purged. Job Repository data purge processing is described below.

The batchManager command line interface has a purge function that allows one to purge data pertaining to specified jobs. There are a variety of ways in which jobs to be purged can be specified. There is also a search capability (called `listjobs`) which supports the same options, enabling one to see the list of jobs that a particular combination of parameters would purge.

It is likely that some automation will be set up for your production environment that will run a script invoking batchManager to purge jobs on a regular basis. How often this runs and what jobs it purges will be factors affecting the size of the tables. It will also be a factor in the responsiveness of the Admin Center Batch tool. Even with performance-assisting indexes, a request for a list of jobs matching some criteria will execute more quickly if there are 1000 jobs to search versus 1 million or more jobs.

Use the utility to purge jobs. Unless IBM support has instructed you to directly remove rows from the table, do not delete data in that manner.

DB2 utilities

The primary "housekeeping" DB2 utilities are COPY, REORG, and RUNSTATS. As with any DB2 table spaces, these utilities should be run periodically for the table spaces holding the Job Repository tables. Generally speaking, a full image copy of a table space should be taken on a weekly basis, with incremental image copies executed daily (or nightly) between the weekly full image copies. REORG should be executed on a regular basis to restore tables and indexes to a well-ordered state – perhaps doing that on a monthly basis would be suitable for the Job Repository table spaces. The RUNSTATS utility can be executed, again perhaps on a monthly basis, with the TABLE(ALL) and INDEX(ALL) options specified, to keep catalog statistics for the Job Repository table spaces reasonably current (an alternative to executing RUNSTATS would be to utilize the STATISTICS option of REORG to generate statistics when the REORG utility is executed).

Overview information about the DB2 utilities can be found in the DB2 for z/OS Knowledge Center on the Web (see https://www.ibm.com/support/knowledgecenter/SSEPEK_11.0.0/intro/src/tpc/db2z_controlwithutilities.html).

Database updates due to WebSphere Liberty maintenance

When the Liberty server is upgraded to a new maintenance level, some new batch capabilities may be enabled. Those new capabilities might require new and/or changed content in the Job Repository tables. Documentation associated with a new maintenance level should provide information about any new capabilities and the table changes (if any) that are required to support them. Updating Job Repository tables should not be required in order to run at the new maintenance level, but new capabilities dependent on table changes will not work until the required changes have been implemented.

After the Job Repository tables are initially created, output of the ddlGen should be saved. It is recommended to save both the original generated DDL and a copy of the customized DDL if any modifications are made. As part of the process of putting on a new maintenance level, it is recommended to re-run the ddlGen against a server at the new level (this does not have to be a production server, just one using the same database (e.g. DB2 for z/OS) with the same `schema` and `tablePrefix` values). Then text-compare the new DDL to the saved copy to see if there were any changes. Though this check may seem unnecessary in light of the fact that documentation associated with a new maintenance level will provide information about related

Job Repository table changes, it is a quick way to help ensure that you do not miss something.

Document change history

Check the date in the footer of the document for the version of the document.

June 27, 2017	Initial Version
June 28,2017	Add document number
September 12, 2017	Add section on defining databases
November 1, 2017	Add warning about SQLCODE -540 and handling SQLCODE -633.

End of WP102716