Detector Conditions

Overview

HPS uses a conditions database which is accessible through the DatabaseConditionsManager. The conditions framework provides classes that read information from the database and return them as lists of typed Java objects with methods for accessing each data field. Users can access these objects to configure their jobs, typically in the `detectorChanged()` method of their Driver classes. The conditions system is setup by specifying the unique name of a detector description and the run number. Typically, this is done by reading this information automatically from the event headers in an LCIO file.

Database Connection

Command Line Options

The parameters for connecting to the database can be specified from the command line as global Java properties.

The default connection information for reading from the database corresponds to the following command line options:

Conditions Database Command Line Options

```
java -Dorg.hps.conditions.url=jdbc:mysql://hpsdb.jlab.org:3306/hps_conditions \
    -Dorg.hps.conditions.user=hpsuser \
    -Dorg.hps.conditions.password=darkphoton [...]
```

These options should always be provided immediately after the `java` command as they are global Java properties rather than command line options provided to a specific application or program within hps-java.

The default database connection uses a read-only replica of the primary MySQL conditions database at Jefferson Lab. Therefore, when connecting from a computer which is outside of the jlab.org domain, you will ***not*** be able to make any changes to this database. If you need to insert records into the database, i.e. for new calibrations, then it must be done behind the JLab firewall, and you must provide valid credentials that allow writing to the database (they are not provided here!).

Using the SLAC Database

The following connection properties can be used to connect to the SLAC conditions database:

SLAC Conditions Database

```
java -Dorg.hps.conditions.url=jdbc:mysql://mysql-node03.slac.stanford.edu:3306/rd_hps_cond \
    -Dorg.hps.conditions.user=rd_hps_cond_ro \
    -Dorg.hps.conditions.password=2jumpinphotons. [...]
```

This database is updated only periodically, not automatically, so you may need to check if it is up-to-date before using it for your jobs.

Using a Local Conditions Database

Local jobs can be run without an internet connection using the built-in support for SQLite.

A db file compatible with sqlite3 may be obtained by using the following commands:

Downloading SQLite Database

```
https://github.com/JeffersonLab/hps-conditions-backup/raw/master/hps_conditions.db.tar.gz
tar -zxvf hps_conditions.db.tar.gz
```

This file may not be up to date with the current master in the JLab conditions database!

The local db file can be used by using this option when running Java:

Connecting to Local Conditions Database

```
java -Dorg.hps.conditions.url=jdbc:sqlite:hps_conditions.db [...]
```

No username or password is required when connecting locally in this way.

> The `hps_conditions.db` file has to be on a disk that is local to the machine. If not then an error may occur: "Caused by: org.sqlite.SQLiteException: [SQLITE_IOERR_LOCK] I/O error in the advisory file locking logic (disk I/O error)"
>
> Alternatively, if you have systems privileges (i.e. root) you can mount your NFS drive with the `local_lock=all` option.

Creating a Local SQLite Database

In order to create a local SQLite database, you will need to create a snapshot of the MySQL database and then convert it to a SQLite db file.

This converter script can be used to produce the db file.

It can be downloaded using these commands:

MySQL Conversion Script

```
  wget https://raw.githubusercontent.com/dumblob/mysql2sqlite/master/mysql2sqlite
  chmod +x mysql2sqlite
```

You can create a dump of the current conditions database using this command:

Creating a Database Dump

```
mysqldump --skip-extended-insert --compact -u hpsuser --password=darkphoton -h hpsdb.jlab.org --extended-
insert=false --lock-tables=false hps_conditions > hps_conditions.mysql
```

Now, you can load the database dump into sqlite3 as follows:

Loading Dump into SQLite

```
  mysql2sqlite hps_conditions.mysql | sqlite3 hps_conditions.db
```

You should now have an up to date copy of the master conditions database locally that can be specified on the command line.

Administration

Creating a Conditions Database Backup

The conditions database can be backed up using a command similar to the following:

Creating a Backup

```
mysqldump -h hpsdb.jlab.org -u$USER -p$PASSWORD hps_conditions &> hps_conditions.sql
```

... where $USER is replaced by your account name with the proper permissions and $PASSWORD with your password.

Loading a Conditions Database Backup

To load the database from a backup, the following command would be used.

Loading a Backup

```
mysql -h hpsdb.jlab.org -u $USER -p$PASSWORD hps_conditions < hps_conditions.sql
```

The above command is for informational purposes only. Fully restoring the database from a backup would need to go through a JLAB CCPR, as the accounts we have access to do not have all the proper permissions for doing this.

Replicating the Database to SLAC

This is a general outline of dumping the database and replicating it to the SLAC MySQL database, which is used for releases, as connecting from SLAC to the JLab database is not reliable enough.

First, from a JLab machine such as `ifarm`, a SQL dump should be created that includes all required statements for dropping tables (the default configuration of `mysqldump` should be fine).

```
mysqldump -h hpsdb.jlab.org -u$USER -p$PASSWORD --disable-lock-tables hps_conditions &> hps_conditions_for_slac.
sql
```

This file should then be copied over to SLAC.

```
scp hps_conditions_for_slac.sql $USER@rhel6-64.slac.stanford.edu:/nfs/slac/g/hps/someDir
```

Finally, the SQL dump should be loaded into the SLAC database using a command similar to the following.

```
mysql -D rd_hps_cond -h mysql-node03.slac.stanford.edu -P 3306 -u $USER -p $PASSWORD < hps_conditions_for_slac.
sql
```

The command may take awhile to execute. If it is successful, the SLAC database should contain an exact replica of the primary conditions database from JLab.

Configuring Detector Conditions for a Job

Command Line Setup

The conditions configuration is typically performed using arguments to command line programs, or the system is setup automatically from information in the LCIO or EVIO events.

The detector name and run number can be provided explicitly to the job manager to override these settings.

```
java -jar hps-distribution-bin.jar -d detector_name -R 5772 [args]
```

Configuration of the EvioToLcio utility is done similarly.

```
java -cp hps-distribution-bin.jar org.hps.evio.EvioToLcio -d detector_name -R 5772 [args]
```

Providing conditions in this way will cause the manager to automatically "freeze" itself after initialization so that subsequent run numbers and detector header information from the input files will be ignored.

Additionally, tags can be specified to filter out the available conditions records in the job, which is described in the Detector Conditions Tags documentation.

Java Initialization

The global instance of the conditions manager can be accessed using the following command:

```
final DatabaseConditionsManager mgr = DatabaseConditionsManager.getInstance();
```

If an instance has not already been instantiated, one will be created.

The conditions system is initialized using the ConditionsManager's setDetector method which takes the name of a detector and a run number.

```
DatabaseConditionsManager.getInstance().setDetector("detector_name", 5772);
```

In some special cases, the conditions system may need to be completely reset by creating and installing a new instance of the manager.

This can be done by calling a special static method on the manager.

Resetting the Conditions Manager

```
DatabaseConditionsManager.reset();
```

You should absolutely ***not*** do this under normal circumstances such as within your Driver code. The method is public only so that this can be done if necessary.

Additional HPS Features

HPS adds several features to the lcsim conditions system.

You can add one or more tags for filtering the conditions records. Only those records belonging to the tag will be accessible.

```
DatabaseConditionsManager.getInstance().addTag("pass0");
```

The conditions system can be "frozen" after it is initialized, meaning that subsequent calls to set a new detector and run number will be completely ignored.

```
DatabaseConditionsManager.getInstance().freeze();
```

This is useful to force the system to load a specific configuration by run number if the actual event data does not have the same run number (or for run 0 events from simulation).

Accessing Conditions from a Driver

Conditions information is accessed in the beginning of the job through the Driver class's detectorChanged method.

```
public void detectorChanged(Detector detector) {
    DatabaseConditionsManager conditionsManager = DatabaseConditionsManager.getInstance();
    EcalChannelCollection channels =
            conditionsManager.getCachedConditions(EcalChannelCollection.class, "ecal_channels").getCachedData();
    System.out.println("got " + channels.size() + " ECal channels");
}
```

All conditions collections required by a Driver should be loaded in this method to avoid incurring a performance overhead by reading the conditions on every event.

You can also access collections not associated to the current run by providing the collection ID.

```
DatabaseConditionsManager conditionsManager = DatabaseConditionsManager.getInstance();
EcalGainCollection gains = new EcalGainCollection();
gains.setConnection(conditionsManager.getConnection());
gains.setTableMetaData(conditionsManager.findTableMetaData("ecal_gains");
gains.select(1234); /* where number is a valid collection ID in the database */
```

This can be used to retrieve reference data that is not accessible in the conditions for the run.

Conditions Types

Conditions Type Table

Java Object Class	Java Collection Class	Default Database Table	Description
BeamEnergy	BeamEnergyCollection	beam_energies	nominal beam energies
EcalBadChannel	EcalBadChannelCollection	ecal_bad_channels	ECal bad channel list
EcalCalibration	EcalCalibrationCollection	ecal_calibrations	per channel ECal pedestals and noise
EcalChannel	EcalChannelCollection	ecal_channels	ECal channel information including map of DAQ to physical channels
EcalGain	EcalGainCollection	ecal_gains	per channel ECal gains
EcalLed	EcalLedCollection	ecal_leds	per crystal LED configuration
EcalLedCalibration	EcalLedCalibrationCollection	ecal_led_calibrations	per crystal LED calibration information (from calibration run)
EcalPulseWidth	EcalPulseWidthCollection	ecal_pulse_widths	ECal signal pulse width (currently unused in recon)
EcalTimeShift	EcalTimeShiftCollection	ecal_time_shifts	ECal signal time shift (currently unused in recon)
SvtAlignmentConstant	SvtAlignmentConstantCollection	svt_alignment_constants	SVT alignment constants in Millepede format may be disabled using *-DdisableSvtAlignmentConstants*
SvtBadChannel	SvtBadChannelCollection	svt_bad_channels	SVT bad channel list
SvtBiasConstant	SvtBiasConstantCollection	svt_bias_constants	SVT bias setting for a time range
SvtCalibration	SvtCalibrationCollection	svt_calibrations	per channel SVT noise and pedestal measurements
SvtChannel	SvtChannelCollection	svt_channels	SVT channel information
SvtDaqMapping	SvtDaqMappingCollection	svt_daq_map	SVT mapping of DAQ to physical channels
SvtGain	SvtGainCollection	svt_gains	per channel SVT gains
SvtMotorPosition	SvtMotorPositionCollection	svt_motor_positions	SVT motor position in mm
SvtShapeFitParameters	SvtShapeFitParametersCollection	svt_shape_fit_parameters	SVT parameters for the signal fit
SvtT0Shift	SvtT0ShiftCollection	svt_t0_shifts	SVT T0 (first sample) shifts
SvtTimingConstants	SvtTimingConstantsCollection	svt_timing_constants	SVT timing configuration constants including offset and phase
TestRunSvtChannel	TestRunSvtChannelCollection	test_run_svt_channels	test run SVT channel information
TestRunSvtDaqMapping	TestRunSvtDaqMappingCollection	test_run_svt_daq_map	test run SVT DAQ mapping
TestRunSvtT0Shift	TestRunSvtT0ShiftCollection	test_run_svt_t0_shifts	test run SVT T0 shift

Database Schema

Data Tables

Each type of condition has an associated database table which contains records with conditions information plus a few additional pieces of information. These tables are modeled by the ConditionsObjectCollection class.

For instance, this is the table schema for the BeamEnergy condition.

```
mysql> describe beam_energies;
+---------------+---------+------+-----+---------+----------------+
| Field         | Type    | Null | Key | Default | Extra          |
+---------------+---------+------+-----+---------+----------------+
| id            | int(11) | NO   | PRI | NULL    | auto_increment |
| collection_id | int(11) | NO   |     | NULL    |                |
| beam_energy   | double  | NO   |     | NULL    |                |
+---------------+---------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

The *id* is the row ID used to uniquely identify the record. The *collection_id* associates a set of records together into a collection. Every data table has these two fields plus additional columns with the conditions data.

Conditions Table

The conditions table associates collections with a run number range.

```
mysql> describe conditions;
+---------------+--------------+------+-----+-------------------+-----------------------------+
| Field         | Type         | Null | Key | Default           | Extra                       |
+---------------+--------------+------+-----+-------------------+-----------------------------+
| id            | int(11)      | NO   | PRI | NULL              | auto_increment              |
| run_start     | int(11)      | NO   |     | NULL              |                             |
| run_end       | int(11)      | NO   |     | NULL              |                             |
| updated       | timestamp    | NO   |     | CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
| created       | datetime     | NO   |     | NULL              |                             |
| tag           | varchar(256) | YES  |     | NULL              |                             |
| created_by    | varchar(255) | YES  |     | NULL              |                             |
| notes         | longtext     | YES  |     | NULL              |                             |
| name          | varchar(40)  | NO   |     | NULL              |                             |
| table_name    | varchar(50)  | NO   |     | NULL              |                             |
| collection_id | int(11)      | NO   |     | NULL              |                             |
+---------------+--------------+------+-----+-------------------+-----------------------------+
11 rows in set (0.01 sec)
```

The *run_start* and *run_end* give a range of run numbers for which the conditions are valid. These can be the same number to specify a single run.

The *table_name* gives the name of the table containing the conditions data.

The *collection_id* gives the collection ID to load from the table.

This table is modeled by the ConditionsRecord class which is accessible via the DatabaseConditionsManager.

When multiple collections of the same type are valid for the current run, the most recently added one will be used by default.

Defining Conditions Classes

New conditions classes should follow a basic template which provides information about its associated database tables and columns.

For example, here is the definition for the BeamEnergy condition.

```
@Table(names = {"beam_energies"})
public final class BeamEnergy extends BaseConditionsObject {

    public static final class BeamEnergyCollection extends BaseConditionsObjectCollection<BeamEnergy> {
    }

    @Field(names = {"beam_energy"})
    public Double getBeamEnergy() {
        return this.getFieldValue("beam_energy");
    }
}
```

The *@Table* annotation on the class maps the class to its possible database tables. Typically, this is a single value.

The *@Field* annotation is applied to a method which should be mapped to a column in the database. The method must be public.

An optional *@Converter* annotation can be used to override the default conversion from the database.

```
@Converter(converter = ConditionsRecordConverter.class)
```

This is only used in a few special cases.