HITACHI
Inspire the Next

PDI Techniques - Design Guidelines

HITACHI
Inspire the Next

Change log (if you want to use it):

Date	Version	Author	Changes

Contents

Overview

This document covers some best practices on designing and building your Pentaho Data Integration (PDI) transformations and jobs. You will learn how to create transformations and jobs for maximum speed, reuse, portability, maintainability, debugging, and knowledge transfer.

The intention of this document is to speak about topics generally; however, these are the specific versions covered here:

Software	Version(s)
Pentaho	6.x, 7.x, 8.0

The Components Reference in Pentaho Documentation has a complete list of supported software and hardware.

Before You Begin

Before beginning, use the following information to prepare for the procedures described in the main section of the document.

This document is arranged in a series of topic groups, with individual best practices for the topic explained. It is not intended to demonstrate how to implement each best practice or to provide templates on the practices defined within.

Server Configuration

This section contains steps for enabling and naming Spoon connections, tips for avoiding the Java Naming and Directory Interface (JNDI), and guides for repository and database usage.

You can find details on these topics in the following sections:

- Enable Spoon Connection Option
- Avoid Using JNDI
- Naming the Connections
- Follow the Lifecycle Management Guidelines
- Use SQL for the Database
- Avoid Stored Procedures and Database Views

Enable Spoon Connection Option

Because Spoon defaults to writing all the shared connection information into every transformation or job file, we recommend you enable Spoon's **Only save used connections to XML?** option.

This setting will keep only those connections used by that specific transformation or job in the XML file.

Avoid Using JNDI

Since PDI has its own algorithms for allocating and using unique connections per step, you do not need to rely on JNDI and connection pooling.

Therefore, we recommend that you:

- Avoid JNDI or similar settings used in enterprise applications for PDI, unless the transformation will run inside the Pentaho Server.
- Use variables to parameterize /hide the connection credentials.
- Avoid connection pooling.

Naming the Connections

Variables make changes to databases over time. To prevent having to change all instances of MySQL when you migrate to Oracle, make sure you name your connections without using the words `Production`, `Development`, `Oracle`, or `MySQL`.

When you migrate from Production to Development, change the underlying `KETTLE_HOME`. This will swap out your connections and the changes will apply in that unique environment.

Follow the Lifecycle Management Guidelines

Although the Pentaho Repository does offer a basic version control system (VCS), that system is not intended to replace any common market VCS tools, such as Git, that you may already be using. Instead, the repository's solution is intended only as a simpler possibility for customers who do not have a company standard VCS in place.

If you already have a VCS, integrate Pentaho with it in development to allow for a more controlled development and release process. While in development, use a file-based workflow. Then, promote your contact to the Pentaho Repository for User Acceptance Testing (UAT) and production.

Use SQL for the Database

We recommend that you do not use PDI steps for features that a database could perform better. A database is often more efficient at sorting and joining than PDI can be.

Remember, this is not an absolute statement, and using your database management system (DBMS)'s SQL can lead to overly complex SQL input.

Avoid Stored Procedures and Database Views

We recommend that you avoid using stored procedures for lookups and database views for inputs.

Views are typically slow to return data. It will be more efficient for you to replace the `view` SQL in the PDI input step than to call that view with an additional `WHERE` clause.

Stored procedures can function well for inputs, but do not use them for lookups, since they are much slower than PDI is. PDI can process several thousand records per second. If each of those rows must go out to a database and run a stored procedure, processing will be slower.

Development for Transformations

Here you will find information on transformation development. Details on these topics can be found in the following sections:

Create Pre-Job Entry Transformations

Create a transformation before creating the job entry that calls that transformation. Create subjobs before creating the jobs that call them.

This allows you to design, code, debug, and test each individual piece of your job without requiring you to run it as part of a larger process. Designing and building from the inside out in this way is key to a modular design.

Add Descriptive Notes

Assign at least one descriptive note to each transformation and job. These notes allow anyone reviewing the code, or taking over support of it, to understand decisions behind the logic or changes. Any major changes to the logic or flow should also specify who made the change.

Avoid Overloading Transformations

If you put more than one data process into a transformation, the processes can get mixed up and become confusing. In addition, such a combined transformation or job can make running one process difficult if the source system for another process in the same transformation or job is down.

Instead, split processing into one transformation per source system for the type of destination data you are working with.

Create Transformations One Step at a Time

Create your transformations one step at a time. This makes testing and debugging easier, as you are adding just one new variable, process, or step to the transformation at a time, so you can isolate problems.

Start with the input when you create a transformation. Test existing steps before adding another step. Add further steps only after you receive the expected result. Do not add or try to connect up multiple steps at the same time.

Working this way allows you to start from a working process each time you add new steps to your transformation.

Name Items Consistently

Name transformations, jobs, and steps consistently, using the same convention each time. This will allow you to see what type of task is being performed when you review logs and files.

Consider a naming convention such as choosing a prefix of the task type, followed by descriptive text, for a job entry. For a transformation step, consider a prefix of the type of step, followed by descriptive text.

Parameterize Jobs and Transformations

Create your transformations starting with fixed inputs, outputs, and configurations. Ensure the basic transformation or job is working without variables.

At this point, you should introduce variables before deploying your jobs or transformations into the server. Parameterized jobs and transformations only work once with fixed values.

This allows you to start from a working process with less complexity. It also makes testing and debugging easier, as you are parameterizing what was once already working. Trying to debug parameters, and the logic of the transformation, is difficult.

Label with Dummy Steps

Use **Dummy (Do nothing)** PDI steps to label your data flows. This will allow you to see the number of rows flowing through this branch in the logs. It also buffers you from downstream changes, because only the steps after the dummy step will need to change, and that will not affect the branching.

Rename each dummy step to better describe the data that flows through that part of the transformation. Use this technique after each filter, case, error, or any other type of branching in the data flow.

Control Transformation Environments

Prior to deployment, develop in a controlled desktop environment, maximizing your time spent in Spoon. This makes transformations easier to build, test, and debug. Migrate from there to Kitchen or the server only after things are working and fully parameterized.

A sample workflow could be:

1. Create and unit-test your transformations in Spoon, while disconnected from a server.
2. Run the job via Kitchen, once the process is tested and working properly.
3. If that is successful, then migrate the code to a server environment running Pentaho Server or Carte.

Using Variables

This section provides information on variables and their use. You can find details on these topics:

Use Parameters for Variables

Use parameters to pass variables into jobs and transformations. Then, at runtime, you can set a parameter to different values to influence the behavior.

This approach allows you to test each transformation and job without relying on global variables. Use variables themselves only to set global variables for an entire job flow.

Separate *`KETTLE_HOME`* Variables

When `KETTLE_HOME` files are shared between projects, values can be mixed and shared improperly. This can cause security and data issues.

Instead, try a different `KETTLE_HOME` variable for each environment (development, QA, production), customer, and project. Pentaho scripts can use the `KETTLE_HOME` variable at startup to alter the files used for that runtime environment.

Use Variables for External References

Use variables and/or parameters for any external reference outside of Pentaho. This applies to host, user, password, directory, filename, and so on.

This will make it easier to migrate from development to QA and production. It will also externalize security sensitive connection information so that developers do not need to know the associated passwords to use the connection.

Validate Job Variables

Before you start the main job, validate all variables and settings required for proper operation, to avoid later connection errors. You can use data validation steps inside transformations, in job-level connections, and in table testing.

Especially critical during transitions between development, QA, and production environments, this approach avoids the problem of getting deep into a job's execution only to find that the environment was not properly set up at the start.

Logging

In this section, you will find ways to navigate through logging operations and optimize them.

More information on these topics is available in:

- Use Logging Tables
- Redirect Output to Kettle Logging
- Subjob for One Log File
- Track Audits on Target Tables
- Row-Level Logging
- Error Handling for Root Job Failures

Use Logging Tables

Use job, transformation, and channel logging tables so you can track performance over time. Use the `kettle.properties` variables instead of creating your own variables or selectively defining which transformations and jobs get logged.

Redirect Output to Kettle Logging

Change the `kettle.properties` variables of `KETTLE_REDIRECT_STDERR` and `STDOUT` to `Y` to redirect all output to `KETTLE` logging destinations.

These variables are set to `N` by default. Turning them to `Y` gives `STDERR` and `STDOUT` more information useful for logging and debugging errors.

Subjob for One Log File

When you are executing jobs in Pentaho Server or Carte, use a subjob that writes to one log file for the entire execution. This organizes all job-related logging into one file. To accomplish this, when you launch the subjob, complete the **Logging** tab on the job entry of the main job.

If you have a server running more than one job at a time, this will make it easier to research an error.

Track Audits on Target Tables

Place audit fields, such as the batch, job, or root channel ID, on each target table to keep track of which records were loaded by which job. You can then cancel certain records or an entire batch, after the fact. The root channel ID will be captured for the root job and used on the channel log table to keep track of all transformations and jobs that start under that root job.

Row-Level Logging

Use the JavaScript `writeToLog()` function for more formatting precision on row-level logging, if you have transformations where each field has its own row.

Row-level logging is useful to track the branching and flow of data during development and testing.

This function will be disabled by default, since most production environments do not use detailed logging or higher. To set the function up, set the logging level to `DEBUG`. Levels below this should not be used for row-level logging. The highest level already logs rows, so this approach is only useful at the `DEBUG` or `DETAILED` levels.

Be sure row-level logging is disabled after testing.

Error Handling for Root Job Failures

Since all job-level failures need to be logged or acted on by operators, enable error handing for all failure cases of all root job entries, as well as for transformation steps that can produce an error or exception.

Rows can be handled individually when you use row-level error handling. The transformation may continue successfully without processing the error rows, and those rows can still be processed in subsequent transformations.

Each job entry has a `TRUE` and a `FALSE` path. The `FALSE` path should always point to a failed action. All subjobs and/or subtransformations will trigger the failproof root-level handlers set in place.

For transformation steps, right-click on each and choose **Error handling…**. On the next screen, check the **Enable the error handling?** box and click **OK**. Make sure to choose the right step (likely a dummy step) and add on all the related error-handling fields for context.

Mondrian Cache

In this section, you will find information on clearing and priming the Mondrian cache, as well as JVM job execution.

Clear the Mondrian Cache

Clear the Mondrian cache after each extract/transform/load (ETL) load, either by using the Pentaho User Console (PUC) or an `HTTP` step in PDI.

Data used in Analyzer can be out of sync with the database if new data is loaded while the old data has been cached. Clearing the Mondrian cache will trigger the application to requery the database to get the latest data after the load.

Prime the Mondrian Cache

Prime the Mondrian cache after clearing it, either by running `xaction` scripts, using Community Data Access (CDA), or scheduling Analyzer reports to run on a schedule to fill the cache.

Users will experience longer load times with an empty cache, so taking this action speeds things up.

Personal JVM Job Execution

For loads that do not happen continuously, use Kitchen to execute jobs in their own Java Virtual Machine (JVM) and to log their output separately from one another.

Shorter-running JVMs are less susceptible to memory issues than longer-running JVMs.

JSON Parsing

In this section, you will find ways to use JavaScript Object Notation (JSON) for tasks, and how to optimize parsing.

More information on these topics is available in:

- Separate Tasks with JSON Parsing
- Use JavaScript for Multilevel JSON
- Expedite Parsing

Separate Tasks with JSON Parsing

When you parse a JSON file, the input step must read the entire file before it begins parsing. This limits the effective size of the JSON file.

A way around this is to make the JSON input data split at one valid object per row, instead of processing as one object per file. Use the **Text file input** PDI step to read the raw data, and the **JSON Input** step to parse each row.

Use JavaScript for Multilevel JSON

If your JSON has embedded arrays in objects, use the **Modified Java Script Value** step to parse the entire object. Using this method is more efficient than using multiple **JSON Input** steps. You can easily configure the **Modified Java Script Value** step to parse many levels in one pass.

Expedite Parsing

Use multiple copies of JSON/JavaScript steps to speed up JSON parsing.

Since the **Modified Java Script Value** step can only pass one level at a time, and JSON parsing is CPU-intensive, if you can split up the tasks across multiple cores, it will be faster.

You can do this by reading each row as an object. Enable multiple copies by right-clicking on a step and choosing **Change Number of Copies to Start…**.

Related Information

Here are some links to information that you may find helpful while using this best practices document:

- Pentaho Components Reference