



# Best Practices for Pentaho's AEL Spark Engine

# HITACHI

Inspire the Next

Change log (if you want to use it):

Date	Version	Author	Changes

# Contents

- Overview..... 4
  - Before You Begin..... 4
    - Use Case: Performing a Word Count with AEL Spark ..... 4
- Best Practices with AEL Spark..... 5
  - Parallelism with AEL Spark..... 5
    - Steps Using Original PDI Implementations ..... 5
    - Steps Using Native Spark Implementations ..... 6
- Using AEL Spark..... 7
- Related Information..... 10
- Finalization Checklist..... 11

## Overview

This document covers some best practices on the AEL Spark engine. You will learn about the [Adaptive Execution Layer \(AEL\)](#), which provides an abstraction layer of the engine that runs Pentaho Data Integration (PDI) transformations.

The AEL feature, which is packaged with Pentaho, allows transformations to run on new and emerging technologies that support distributed job processing, in addition to the traditional Kettle engine. The Apache Spark AEL implementation is the first of these technologies. Apache Spark is designed to process massive amounts of data by distributing work across clusters.

Our intended audience is PDI developers, with some basic knowledge of Spark, YARN and HDFS.

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

Software	Version(s)
Pentaho	8.x

The [Components Reference](#) has a complete list of hardware.

This page intentionally left blank.

in Pentaho Documentation supported software and

## Before You Begin

This document assumes that you have knowledge of Pentaho and that you have already installed Pentaho, a supported Hadoop distribution, and Spark, are just interested in learning more about AEL Spark. More information about related topics outside of this document can be found in the AEL setup documentation.

### *Use Case: Performing a Word Count with AEL Spark*

*Janice needs to find out how many times the word “leverage” is used in a long whitepaper. She has decided to use a Word Count. With a simple transformation, AEL Spark can perform this summary on data within a Hadoop cluster, and scale out as the number of input files increases.*



Figure 1: A Word Count Implemented for Use with AEL Spark

# Best Practices with AEL Spark

The AEL Spark engine allows PDI transformations to be executed on the Apache Spark distributed processing system. Traditionally, each step was assigned a thread and executed in a single JVM, but with AEL Spark, the transformation's steps are distributed to Spark executors where they operate on partitions of data.

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

- [Parallelism with AEL Spark](#)
- [Using AEL Spark](#)

## Parallelism with AEL Spark

To take full advantage of an underlying execution engine, the AEL allows PDI transformation steps to have more than one implementation. With the Pentaho engine, there was a single way for steps to be executed, through the implementation of the `processRow` method in the PDI API.

With AEL Spark, some steps have Spark-specific implementations that use native Spark capabilities. The capabilities are designed to be highly distributed, yet produce the same result sets.

### *Steps Using Original PDI Implementations*

Most PDI steps can work on rows from a stream in parallel. With the Pentaho local run configuration, parallelism is controlled by right-clicking a step and changing the number of copies/threads dedicated to it. However, the number of copies setting is not used by AEL Spark, as parallelism is dictated by Spark partitioning.

Steps that perform operations on single rows from a stream are great candidates for running in parallel within AEL Spark. Some example steps that behave like this are **Calculator**, **Select Values**, **Split Fields**, and **Add a checksum**. When executed in AEL Spark, the input data sets are split into partitions that are distributed across Spark executors. The original Pentaho engine `processRow` method is distributed to the executors, where it is invoked on the partitions of data in Spark Tasks.

### *Steps That Are Not Distributed*

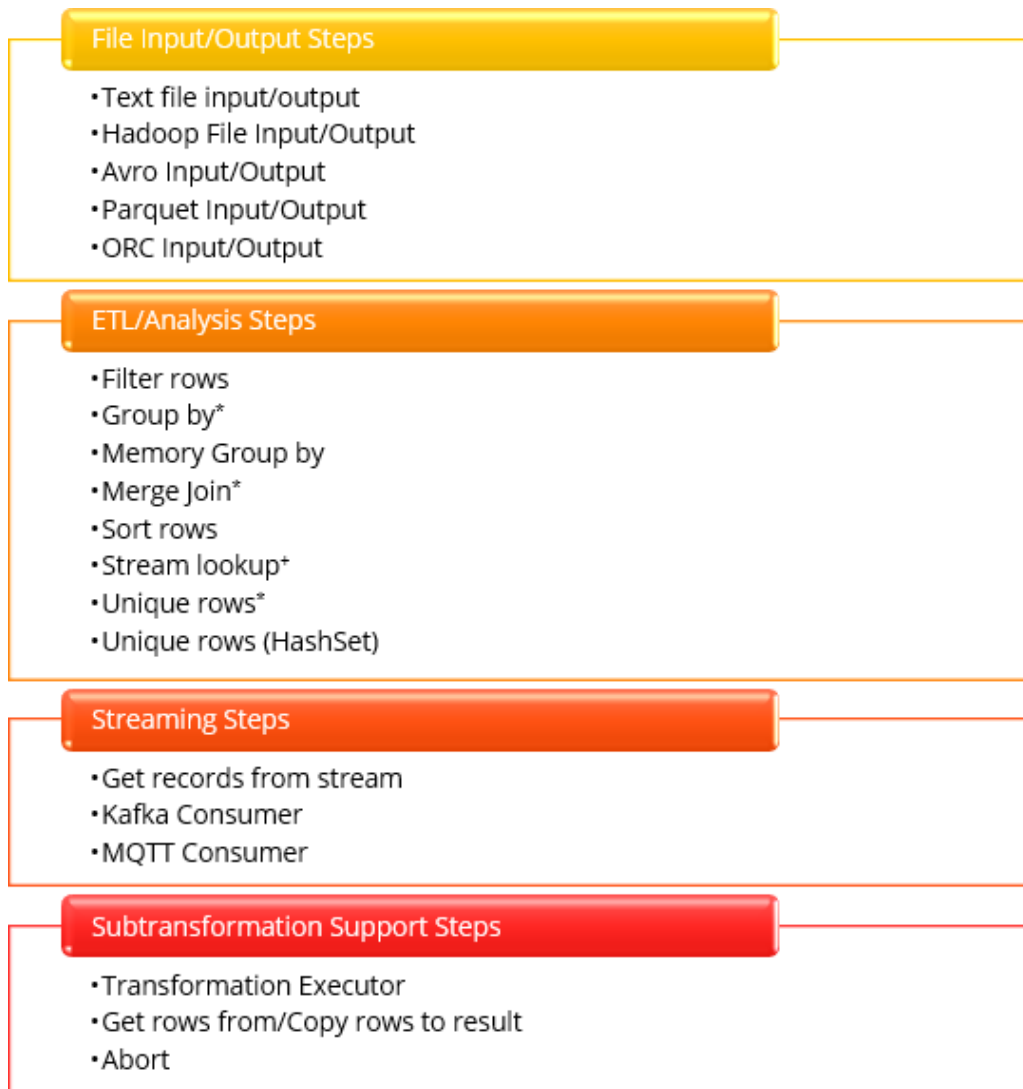
Some PDI steps require multiple input rows of data before they can produce output rows, or they may not naturally lend themselves to concurrent execution. If these steps have not been overridden by a native Spark implementation, the AEL Spark engine automatically adds a `coalesce` operation to process all input data by a single thread on a single Spark executor using the `processRow` method. While this may reduce performance, it ensures correct results are produced by the transformation.

You can find the list of steps that have the `coalesce` operation added under an installed AEL daemon in the `data-integration/system/karaf/etc/org.pentaho.pdi.engine.spark.cfg` file, in the `forceCoalesceSteps` property. Some of the steps on this list include **Row denormaliser**, **Add**

**sequence**, and other steps that a user would not change the number of threads in when using the classic PDI engine.

## Steps Using Native Spark Implementations

Spark is designed to process big data that is distributed on a cluster. Unlike MapReduce, which was limited to at-most one reduce operation, Spark can have as many reduces (data shuffling operations) as are required to process a directed acyclic graph (DAG). Some of the PDI steps have implementations that override the Pentaho engine's `processRow` method to use Spark's native distributed processing capabilities. As of 8.1, the list of steps with native Spark implementations is as follows:



\*Does not require sorted input rows in Spark AEL

\*The info stream is broadcast to all Spark Executors

Figure 2: PDI Steps with Native Spark Implementation

## Using AEL Spark

In the following section, you will find a collection of our recommendations for using AEL Spark, including how to use certain PDI steps to keep your jobs and transformations running efficiently. Pentaho's 8.1 release introduced several new features, including the ability to reuse Spark sessions, which is a great tool for AEL Spark ETL development.



Figure 3: Best Practices for AEL Spark and Pentaho

Stream  
lookup for  
Simple Joins

When you are performing simple joins with small info streams, use **Stream lookup** to allow data from an info stream to augment rows from a main stream. In cases where a cartesian product is not required (the info stream only has single entries for the key lookup fields), **Stream lookups** can be used to implement left (main left join info) or inner joins (inner joins require an additional filtering step).

We recommend using **Stream lookups** only when the info stream is considerably smaller than the main input stream. With the AEL Spark engine, the **Stream lookup** step is overridden with a Spark implementation that utilizes Spark's broadcast variable feature. The data from the lookup step (or info stream) is prepared and distributed to all Spark executors. When the step executes, a mapping function retrieves and augments the main input row stream with field data from the broadcasted info stream to produce output rows.

If the data to be used for the lookup step/info stream can fit in the memory of every Spark executor, a **Stream lookup** should perform significantly better than a **Merge Join**. The **Merge Join** implementation will shuffle and move virtually all row data across the network of executor nodes, while the **Stream lookup** can perform the joins without moving the rows of the larger input stream around the network.

In MapReduce terminology, the **Stream lookup** executes an *N-1* map-side join, where a smaller input is temporarily placed in the distributed cache. The **Merge Join** behaves more like a reduce-side join.

Avoid Sorts  
Before  
Certain Steps

When you use Kettle, input rows must be sorted by key fields for steps like **Group by**, **Unique rows**, **Merge Join**, and **Switch/case**. However, all of these steps have overridden Spark implementations with AEL Spark, which do not require the input rows to be sorted.

If the transformation being created is *exclusively* for execution using AEL Spark, omitting extra **Sort rows** steps will improve performance on AEL Spark, since **Sort rows** requires a shuffle, or network transfer of the row data around the cluster of Spark executors.



*If a transformation is required for both AEL Spark and the PDI engine, try the **Memory Group by** or **Unique rows (Hashset)** steps instead. These steps do not require sorted inputs for the PDI engine, but take advantage of an overridden Spark implementation with AEL Spark.*





### Spark History Server

Pentaho 8.1's AEL implementation introduced the ability to configure the Spark History server, which logs job and performance metrics as Spark is processing.

Configure the History server (disabled by default, located in the `application.properties` file within `data-integration/adaptive-execution/config/`) by setting the `sparkEventLogEnabled` property to `true`, and finding the correct value for the `sparkEventLogDir` property by visiting the Spark History server's home page.

---

```
sparkEventLogEnabled=true
sparkEventLogDir=hdfs://quickstart.cloudera:8020/user/spark/spark2ApplicationHistory
```

---




### Reuse Spark Sessions During Development

Reusing Spark sessions during development lets you keep your Spark cluster of executors active instead of being shut down after the transformation completes. Any following transformation requests will be sent to the still-active Spark session, drastically reducing the wait time. However, this feature is only intended for development, and should not be used for production AEL Spark workloads. To enable the feature, simply add the following property to the `kettle.properties` file on the development client:

---

```
KETTLE_AEL_PDI_DAEMON_CONTEXT_REUSE=true
```

---



### Tune with Parameters and Variables

Custom properties that allow tuning or configuration of Spark are traditionally specified in `spark.conf` or as additional arguments in a `spark-submit` or `spark-shell` command. AEL Spark also has a way to specify per-transformation or per-daemon properties.

Any property that is prefixed with `spark.` will be passed to the Spark execution environment.

- The first method to specify a custom Spark property is to add it to the `data-integration/adaptive-execution/config/application.properties` file. This property will be applied to all transformations executed by the corresponding AEL Spark daemon.
- The second option is to specify a `spark.` property as a transformation variable or parameter at execution time. Transformation parameters have the highest precedence, followed by transformation variables, and finally followed by values specified in the `application.properties` file.

Please note that there are some properties that cannot be overridden. More information on this can be found at [Specify Additional Spark Properties](#).

### Cluster Ingest Data Treatment

Control the partitioning for your input steps through cluster ingest data treatment.

Spark processes partitions of data defined by file input splits. With Hadoop, every individual file is split by some block size defined when the file is written to HDFS.

When you are preparing the data files to be used as input to an AEL Spark transformation, tune parallelism by staging input files at a size that yields the desired number of partitions to process.

### Caution Using External Resources

Use caution when you are working with steps that call upon external resources.

Keep in mind that Spark is a distributed processing technology, and a transformation could execute tasks on data partitions with many threads, within many processes, on many worker nodes.

Steps like **REST Client** or **Database lookup** in a transformation executing in parallel could overwhelm the server being called. Simply put, keep this in mind during transformation design, so you do not self-inflict a denial of service attack.

### Caution Using Coalesced Steps

The `forceCoalesceSteps` list was described in the [Steps That Are Not Distributed](#) section of this document. Any steps on this list that do not have an overridden Spark implementation will be executed by a single thread on a single Spark executor. This is required so the transformation behaves correctly at the cost of performance.

If possible, try to reserve steps on this list for smaller streams, such as those that have been summarized by a **Group by**, or towards the end of the transformation.

## Related Information

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

- [Components Reference](#)
- [Set Up the Adaptive Execution Layer \(AEL\)](#)
- [Specify Additional Spark Properties](#)

# Finalization Checklist

This checklist is designed to be added to any implemented project that uses this collection of best practices, to verify that all items have been considered and reviews have been performed.

Name of the Project: \_\_\_\_\_

Date of the Review: \_\_\_\_\_

Name of the Reviewer: \_\_\_\_\_

Item	Response	Comments
Did you use <b>Stream lookup</b> for simple joins?	YES _____ NO _____	
Did you remember not to use sorts before steps like <b>Group by, Unique Rows, and Merge Join</b> ?	YES _____ NO _____	
Did you set up the Spark History server?	YES _____ NO _____	
Did you reuse Spark sessions during development?	YES _____ NO _____	
Did you tune using parameters and variables?	YES _____ NO _____	
Did you control the partitioning for your input steps through cluster ingest data treatment?	YES _____ NO _____	
Did you use caution when calling on external resources?	YES _____ NO _____	
Did you use caution regarding coalesced steps?	YES _____ NO _____	