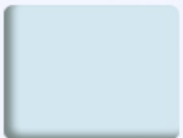


CloverETL overview part I.

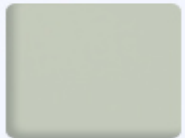


■ Agenda

- CloverETL Concept
- CloverETL Basics
- Running Transformations
- CloverETL Logging
- Extending CloverETL
- Fine-tuning CloverETL
- Writing Transformation Code

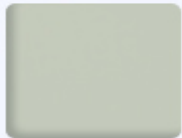


CloverETL Concept



- CloverETL is based on concept of transformation graph
- Graph consists of
 - Nodes – perform “atomic” data manipulation
 - Edges – connect nodes, carry data among Nodes
- Graph has to be:
 - Acyclic (one piece of data can hit particular node only once)
 - Must have at least one node

- Graph is divided into units – phases
- Phase
 - Has unique number used for ordering Phases
 - Every node/edge has to be assigned to phase
 - Graph must have at least one phase
 - Phases are executed sequentially
 - Nodes within phase are executed in parallel



■ Phase

- Data crossing phase boundary (through edge connecting two nodes in two different phases) need special treatment
- Phase is considered to be execution step
 - if processing of one phase fails the whole transformation is stopped



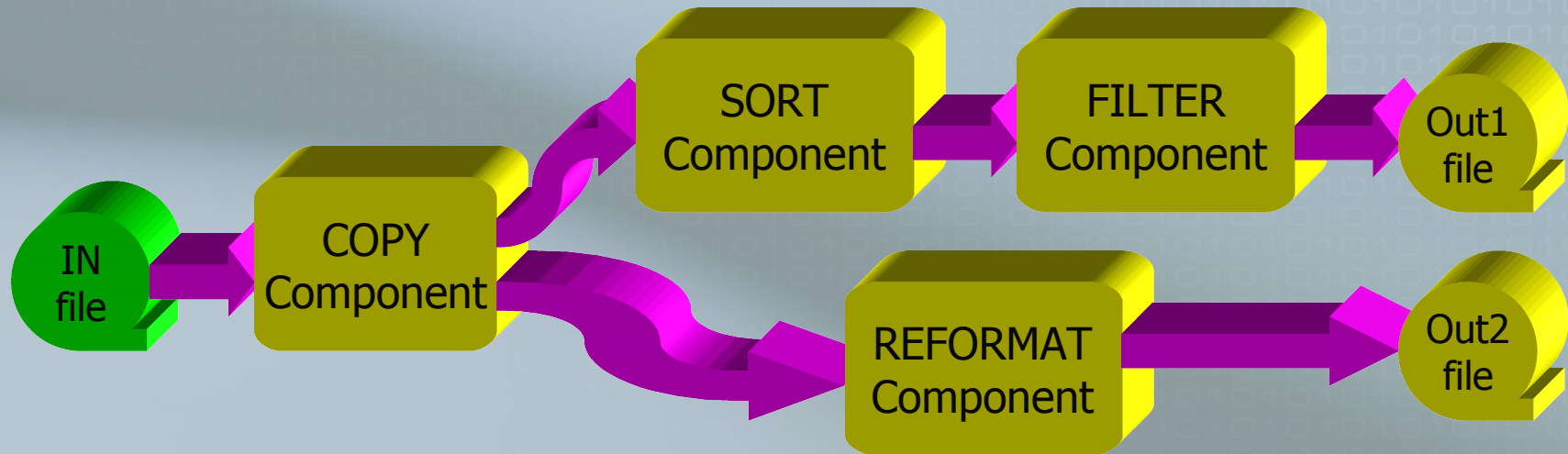
■ Data

- CloverETL can transform only structured data
- Data must fit Record->Field structure
- Data flow consists of set of Records flowing between Nodes (Transformation Components)
- There is currently only one Record type
- There are 7 data types (String, Date, Byte and numeric types)

- CloverETL implements pipeline-parallelism
 - When one Node is done with processing data record the record is immediately sent to following Node
 - At one point in time there are as many records processed as there are nodes – thus level of parallelism corresponds to graph's topology
 - Memory consumption is determined by graph's topology/configuration, not by size of data processed
 - Scalability is approaching linear function

Note: most of the time the transformation is I/O bound.

Transformation graph



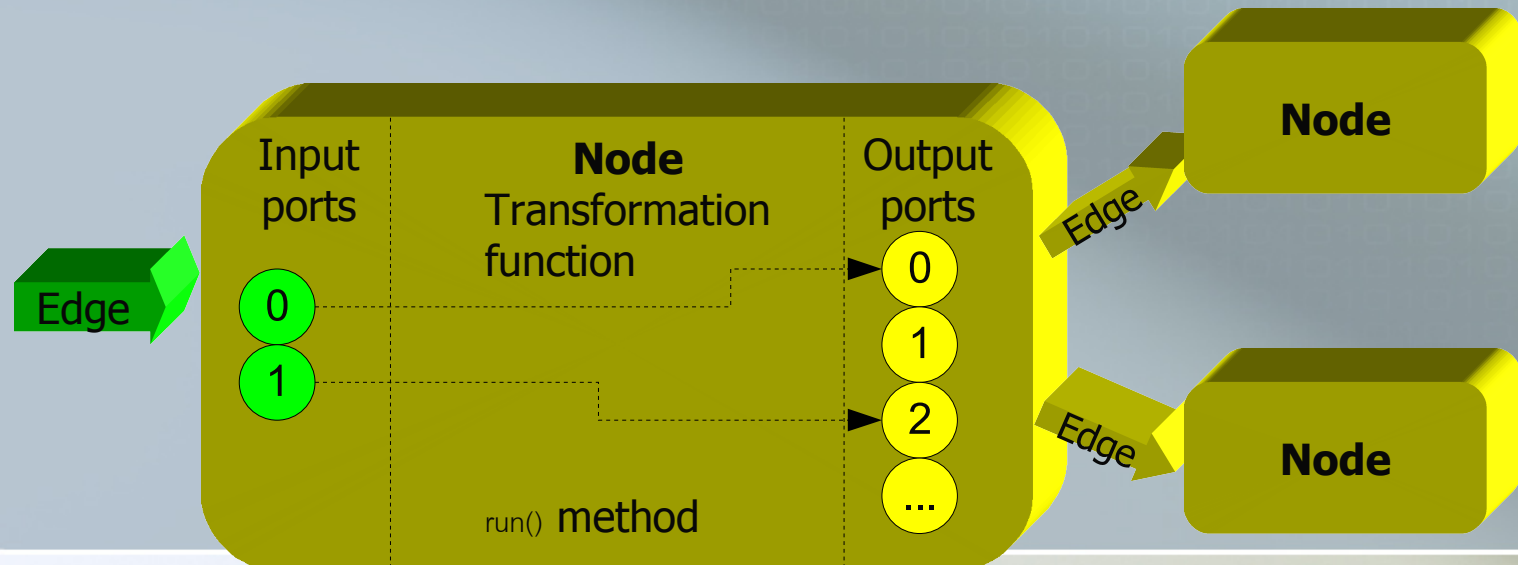
Edge

Component

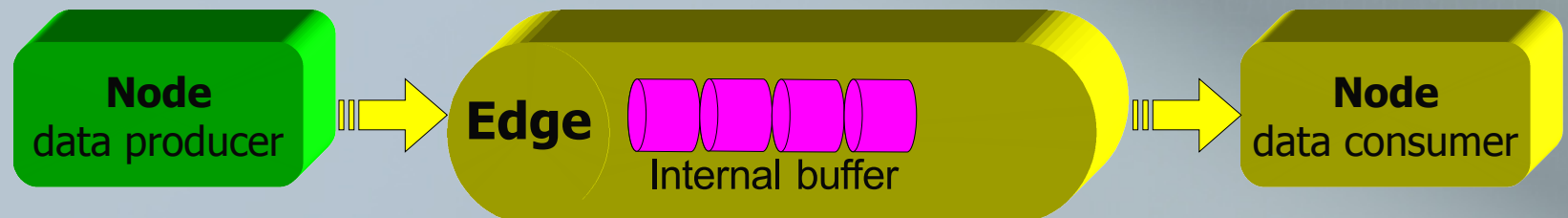
Component

■ Node

- has 0..n input ports
- has 0..n output ports
- Performs “atomic” data transformation between set of input ports & output ports



- **Edge**
 - Connects two Nodes in one direction
 - Connection is between specific port on “sending” Node and port on “receiving” Node
 - There are different Edge variants for different purposes



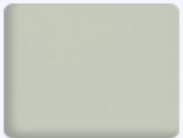
- Port
 - Node communicates through ports (Input/Output)
 - Node operates in vacuum – it has no knowledge about other components connected to it
 - Each port has associated metadata which describe structure of data coming from that port (if Input) or data which can be sent through port (Output)



- Metadata
 - Object with assigned “name”
 - Describes structure of data (record)
 - Record
 - list of fields
 - other properties (some user defined)
 - Field
 - Name
 - Type
 - Formatting parameters
 - Other specific properties (scale, length, etc.)

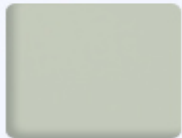


CloverETL Basics



- CloverETL is a framework
 - Can be used as
 - a standalone application (through `runGraph` utility)
 - a library of Java classes (embedded in other application)
 - Contains many reusable parts
 - Various data types
 - Data transformation algorithms
 - Scripting language
 - ...

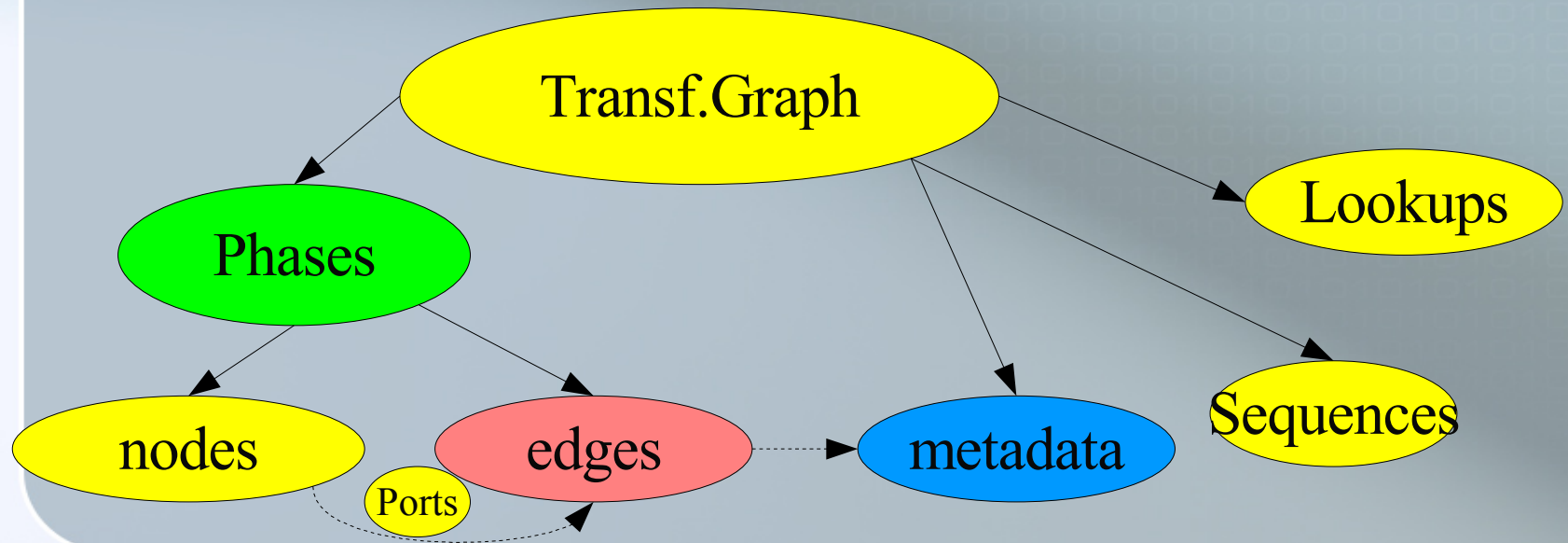
- Transformation is defined in form of graph, which contains
 - Metadata descriptions
 - Sequences
 - Lookup Tables
 - Database connections
 - Phases
 - Nodes/Transformation Components
 - Edges



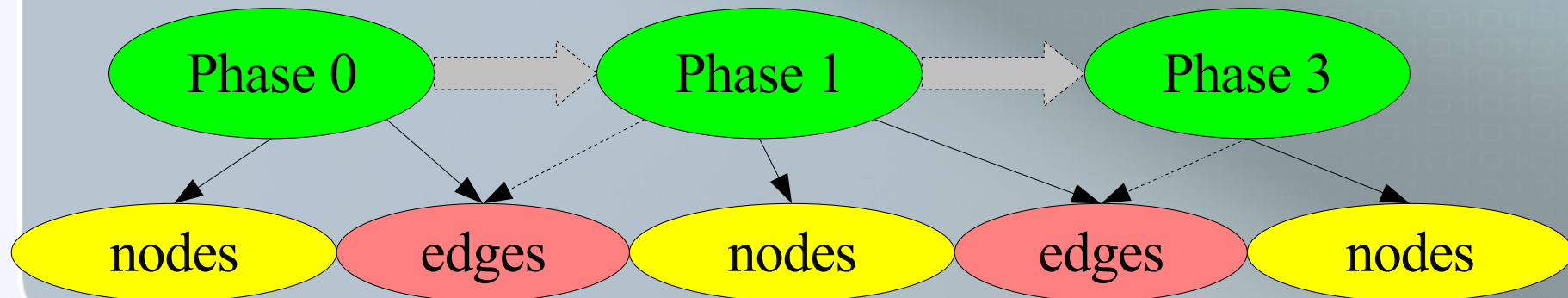
- Graph can be assembled by
 - reading/deserializing from XML
 - instantiating Java classes
 - TransformationGraph
 - DataRecordMetadata, DataFieldMetadata
 - DirectEdge, BufferedEdge, PhaseEdge,
 - Nodes -
DataReader, DataWriter, Filter, Sort, SimpleCopy.....



- Graph is internally represented by instance of TransformationGraph class
- This class keeps list of all other “objects” composing graph

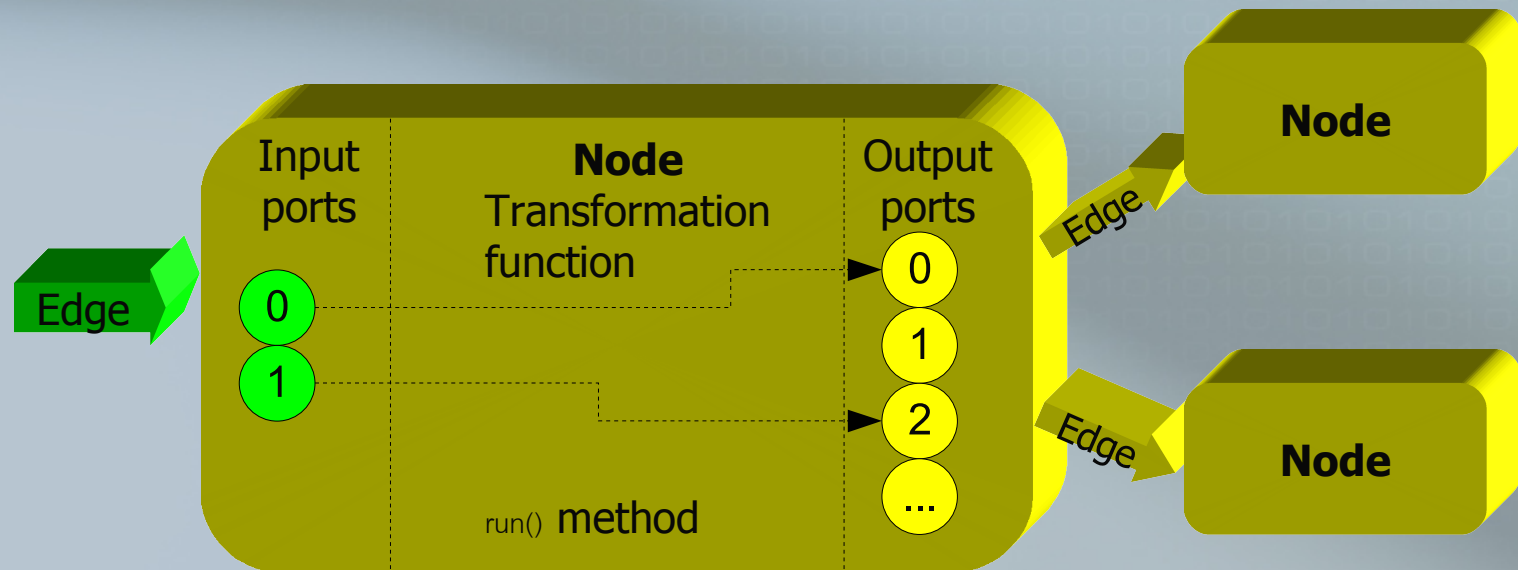


- Phase is merely organizational unit
 - Groups nodes and sets boundaries between nodes
 - Allows ordering/sequencing transformation execution
 - What should be run first, second, etc
 - It is planned to use phases as checkpoints for graph restartability



- Node (Component) is unit performing “atomic” transformation of data
 - Node is logical object
 - Component is concrete implementation of data transformation (e.g filter, sort, ...)
 - Complicated transformations are created by chaining Nodes/Components (using Edges)
- Node is also Java class – `org.jetel.graph.Node`
- All components must subclass Node class
- Each node/component in graph is executed as independent thread

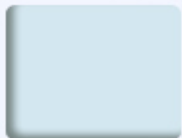
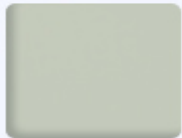
- Node's Java class has one "execution" method `run()`
 - This method is responsible for transforming data from inputs onto outputs



- `run()` method
 - Is executed upon node's initialization
 - Its responsibility is to read data from all attached input ports and send data to all attached output ports
 - When node is done with processing data, it closes all output edges to indicate following components that they shouldn't expect any more data



- Nodes/Components are divided into 4 groups
 - Nodes with only output ports
 - data producers (various data readers, generators)
 - Nodes with only input ports
 - data sinks (various writers)
 - Nodes with both in&out ports
 - data transformers (sorters, filters, joiners, etc.)
 - Nodes with no ports
 - special purpose nodes (system execute, DB execute, ...)



- Every Component has
 - common set of parameters
 - Unique ID (composed of alphanumeric chars)
 - Name (free text)
 - Type (alphanumeric chars)
 - own parameters (for example:)
 - Key
 - FileURL
 - SqlStatement
 - ...



- Component's parameters can be set

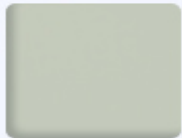
- By calling Java class methods

```
public void setSortOrderAscending(boolean ascending)
```

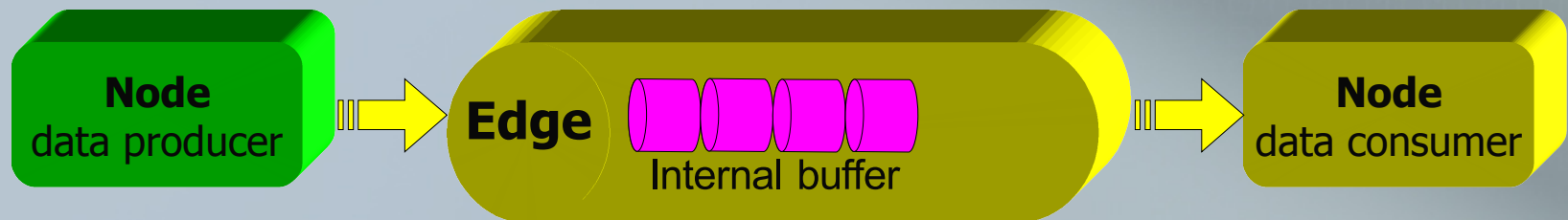
- By reading parameters from XML serialization of component

```
<Node enabled="enabled" id="SORT" sortKey="ShipName;ShipVia"  
type="SORT" sortOrder="Ascending">
```

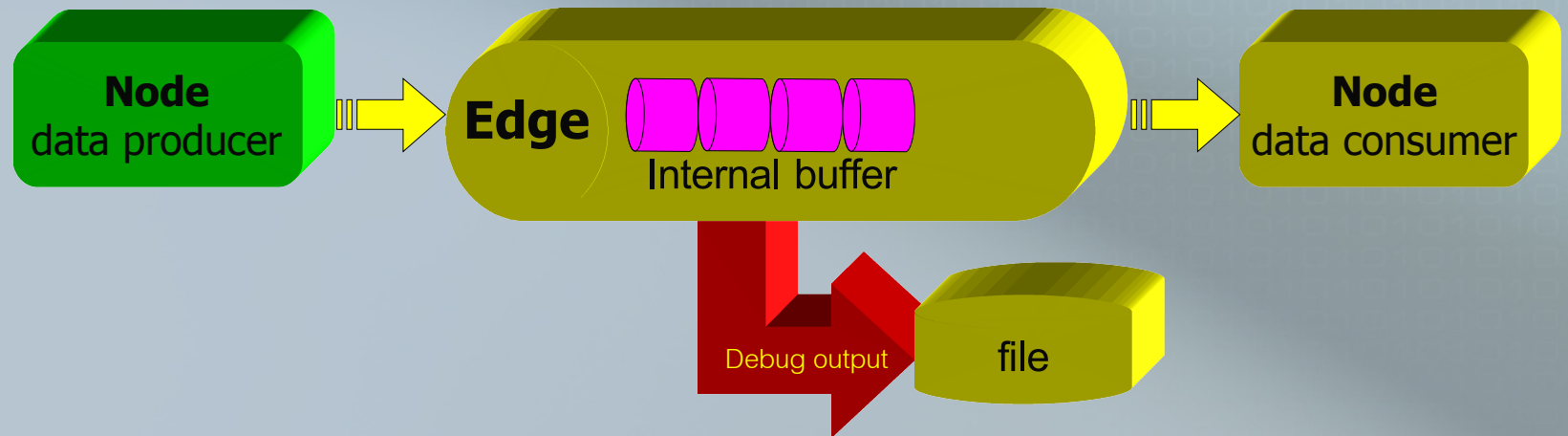
- Edge connect two Nodes/Components in one direction
 - there are currently 3 types of edges
 - Direct, Buffered, Phase
 - when assembling transformation graph (programmatically) only “basic” edge type is used.
 - other types are used by Transformation graph after analysis of current graph topology is performed - “basic” edge is then replaced by appropriated implementation



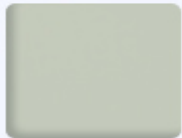
- Every Edge implementation performs internal buffering to speed-up processing of data
- Edge implements two basic functions
 - writeRecord()
 - readRecord()



- Edge can be switched into mode when it saves data flowing through it for future analysis - debugging



- Metadata describes structure of data records
 - Name
 - Type of record (fix-len, delimited)
 - Additional text-formatting information
 - Individual data fields
 - Name Text Format
 - Type Scale
 - Delimiter / Length Locale

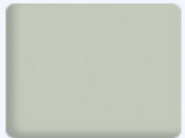


- Metadata is represented as Java objects
 - `org.jetel.metadata.DataRecord`
 - `org.jetel.metadata.DataField`
- Metadata can be defined/created
 - By creating instances of appropriate Java classes
 - Deserialized from XML
 - Create automatically based on structure of existing DB table or SQL query

■ Clover's data types

Data type	Based on	Size	Range of values
string	java.lang.String	depends on actual data length	
date	java.util.Date	64bit - sizeof(long)	starts: January 1, 1970, 00:00:00 GMT increment: 1ms
integer	java.lang.Integer	32bit - sizeof(int)	min: -2^{31} max: $2^{31}-1$.
numeric	java.lang.Double	64bit - sizeof(double)	min: 2^{-1074} max: $(2-2^{-52}) \cdot 2^{1023}$
long	java.lang.Long	64bit – size of (long)	min: $2^{63}-1$ max: -2^{63}
decimal	Clover's own implementation - similar to java.math.BigDecimal	depends on precision parameter - maximum digits this datatype can contain	depends on scale and precision parameters eg. decimal(6,2) can represent min: -9999.99 max: 9999.99
byte	java.lang.Byte	depends on actual data length	min: 0 max: 255

Running transformation



- In order to run CloverETL transformation, its definition must be ready first
 - In form of XML description
 - As a Java code which assembles transformation graph



■ XML description

```
<?xml version="1.0" encoding="UTF-8"?>
<Graph name="Testing Simple Copy">
  <Global>
    <Metadata fileURL="metadata/employees.fmt" id="InMetadata"/>
    <Property fileURL="workspace.prm" id="GraphParameter0"/>
  </Global>
  <Phase number="0">
    <Node id="BROADCAST" type="SIMPLE_COPY" enabled="enabled"/>
    <Node id="INPUT1" type="DELIMITED_DATA_READER"
    enabled="enabled" fileURL="{WORKSPACE}/data/employees.dat" />
    <Node id="TRASH1" type="TRASH" enabled="enabled" />
    <Edge fromNode="INPUT1:0" id="INEDGE1" metadata="InMetadata"
    toNode="BROADCAST:0"/>
    <Edge fromNode="BROADCAST:0" id="INNEREDGE1"
    metadata="InMetadata" toNode="TRASH1:0"/>
  </Phase>
</Graph>
```

■ Java code

```
TransformationGraph graph = new TransformationGraph();
Edge inEdge=new Edge("InEdge",metadataIn);
Edge outEdge=new Edge("OutEdge",metadataIn);

Node nodeRead=new DelimitedDataReader("DataParser",args[0]);
String[] sortKeys=args[3].split(",");
Node nodeSort=new Sort("Sorter",sortKeys, true);
Node nodeWrite=new DelimitedDataWriter("DataWriter",args[1],false);

// add Edges & Nodes & Phases to graph
graph.addEdge(inEdge);
graph.addEdge(outEdge);
graph.addPhase(_PHASE_1);
    _PHASE_1.addNode(nodeRead);
    _PHASE_1.addNode(nodeSort);
graph.addPhase(_PHASE_2);
    _PHASE_2.addNode(nodeWrite);

// assign ports (input & output)
nodeRead.addOutputPort(0,inEdge);
nodeSort.addInputPort(0,inEdge);
nodeSort.addOutputPort(0,outEdge);
nodeWrite.addInputPort(0,outEdge);

graph.init();
graph.run();
```

- Before run – dependencies
 - CloverETL depends on following libraries
 - Apache Commons Logging
 - Apache Log4J (in case “extended” logging is desired)
 - Apache POI (in case Excel reader/writer is used)
 - JDK 1.5 & tools.jar (part of JDK) must be installed on your system in order for Clover to run
- Starting with CloverETL version 2.0.0 all above mentioned libraries (with exception of JDK) are distributed together with Clover engine library

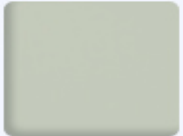
- Executing transformation graph from XML
 - Clover engine offers `runGraph` utility – located in `org.jetel.main` package
 - `runGraph` can be executed from command line:

```
java -cp "/clover/cloveretl.engine.jar;/clover/lib/commons-logging.jar;/clover/lib/log4j-1.2.12.zip;./"  
org.jetel.main.runGraph -plugins /clover/plugins <...name of your transformation graph>
```

Example assumes that CloverETL engine distribution package has been extracted to `/clover` directory.

- `runGraph` usage:

```
runGraph [- (v|cfg|P: |tracking|info|plugins|pass) ]  
  <graph definition file>
```



- `runGraph` command line parameters
 - `-v` (verbose)
 - Print more information about graph run
 - `-P:<key>=<value>`
 - Define parameter/property of graph
 - `-cfg <filename>`
 - Load definition of properties from file
 - `-tracking <seconds>`
 - How frequently execution status should be reported
 - `-plugins <directory>`
 - Where the engine should look for its plugins
 - `-pass <passphrase>`
 - Pass phrase to be used for decrypting encrypted passwords stored with graph

■ Graph execution order

1.graph.init()

a) Initialize graph objects

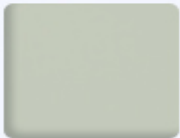
- connections (create connections to DBs)
- sequences (open sequences)

b) Distribute nodes to appropriate phases

c) Disable nodes marked as “disabled”

d) Replace disabled nodes with edges bridging that nodes (if configured so)

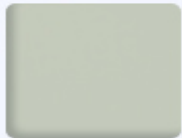
e) Analyze graph topology – graph must be acyclic



■ Graph execution order

1. graph.init() .. cont...

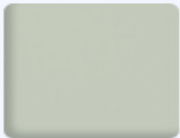
- f) Analyze edges – change implementing object to PhaseEdge for those crossing phase boundary – these use temp files for buffering data flows
- g) Analyze multiple feeds – change edges which are part of multiple feeds to BufferedEdge object
 - ◆ Multiple-feed » when one component feeds through multiple output ports other components, dead-lock could occur



■ Graph execution order

2.graph.run()

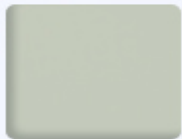
- a. Timestamp start of graph execution
- b. Create & execute WatchDog() thread » WatchDog takes over the execution from this part «
- c. Wait till WatchDog() thread finishes
- d. Free all resources (Nodes, Edges, Connections, Sequences, LookupTables, etc..)
- e. Exit to system with status indicating success or failure



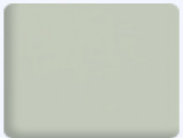
■ Graph execution order

2a. WatchDog.run()

- For each Phase defined in graph, do following:
 - a Initialize Phase – in turn initialize all Nodes in Phase (init() method on each Node is called) and also all Edges in Phase (init() method called too)
 - b Star-up all Nodes in Phase (create Thread for each Node)
 - c Watch running nodes
 - i Print tracking report
 - ii Look for failed Nodes (in such case stop execution)
 - iii Look for last Node in Phase finishing, then return
 - d Perform Phase cleanup



- Example of transformation execution



*** CloverETL framework/transformation graph runner ver 1.8, (c) 2002-06 D.Pavlis, released under GNU Lesser General Public License ***

Running with framework version: 2.0 build#119 compiled 27/10/2006 15:22:46

Graph definition file: graphAggregateUnsorted.grf

INFO [main] - Starting WatchDog thread ...

INFO [WatchDog] - Thread started.

INFO [WatchDog] - Running on 1 CPU(s) max available memory for JVM 1188 KB

INFO [WatchDog] - [Clover] Initializing phase: 0

DEBUG [WatchDog] - initializing edges:

DEBUG [WatchDog] - all edges initialized successfully...

DEBUG [WatchDog] - initializing nodes:

DEBUG [WatchDog] - AGGREGATE ...OK

DEBUG [WatchDog] - INPUT1 ...OK

DEBUG [WatchDog] - OUTPUT ...OK

INFO [WatchDog] - [Clover] phase: 0 initialized successfully.

INFO [WatchDog] - Starting up all nodes in phase [0]

DEBUG [WatchDog] - AGGREGATE ... started

DEBUG [WatchDog] - INPUT1 ... started

DEBUG [WatchDog] - OUTPUT ... started

INFO [WatchDog] - Successfully started all nodes in phase!

INFO [INPUT1] - Start parsing file D:/ClarityBlue/demo_workspace/Training/data/orders.dat

INFO [WatchDog] - Execution of phase [0] successfully finished - elapsed time(sec): 1

<...continues on next page..>

```

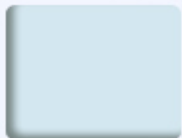
INFO [WatchDog] - -----** Start of tracking Log for phase [0] **-----
INFO [WatchDog] - Time: 31/10/06 08:28:36
INFO [WatchDog] - Node                Status                Port                #Records
INFO [WatchDog] - -----
INFO [WatchDog] - AGGREGATE                OK
INFO [WatchDog] -                               In:0                49
INFO [WatchDog] -                               Out:0               34
INFO [WatchDog] - INPUT1                   OK
INFO [WatchDog] -                               Out:0               49
INFO [WatchDog] - OUTPUT                   OK
INFO [WatchDog] -                               In:0                34
INFO [WatchDog] - -----** End of Log **-----
INFO [WatchDog] - Forcing garbage collection ...
INFO [WatchDog] - -----** Summary of Phases execution **-----
INFO [WatchDog] - Phase#           Finished Status           RunTime(sec)       MemoryAllocation(KB)
INFO [WatchDog] - 0                 0                       1                   196
INFO [WatchDog] - -----** End of Summary **-----
INFO [main] - WatchDog thread finished - total execution time: 1 (sec)
INFO [main] - Graph execution finished successfully
Execution of graph successful !

```

CloverETL Logging



- CloverETL uses Apache Commons Logging as its logging layer
 - Commons Logging is a thin wrapper around one of three possible logging subsystems:
 - Java's logging package (present from JVM 1.4)
 - Apache Log4J (<http://logging.apache.org/log4j/>)
 - Very simple logging subsystem which is part of Commons Logging



- It is recommended to use Log4J package together with CloverETL as it gives greatest flexibility
- Log4J is activated simply by adding its .jar into Clover's CLASSPATH when executed
- CloverETL already contains configuration file for Log4J which ensures proper display of messages
- CloverETL from version 2.0 contains Log4J bundled with it.

- Configuring log4J
 - Configuration is performed by `log4j.properties` file
 - It is part of CloverETL library in root directory of `cloveretl.engine.jar`
 - As it is a text file it can be easily changed to modify Log4J behavior

- Configuring log4J
 - Following is a complete log4J configuration

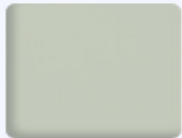
```
#this options sets the finest level of messages being output
#you may set it to INFO in order to eliminate debugging messages
log4j.rootLogger=DEBUG, A1
```

```
# A1 is set to be a ConsoleAppender which outputs to System.out.
log4j.appender.A1=org.apache.log4j.ConsoleAppender
```

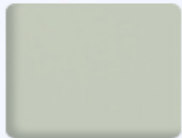
```
# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
```

```
# The conversion pattern uses format specifiers. You might want to
# change the pattern and watch the output format change.
#log4j.appender.A1.layout.ConversionPattern=%-4r %-5p [%t] %37c
%3x - %m%n
log4j.appender.A1.layout.ConversionPattern=%-5p [%t] - %m%n
```

- Configuring log4J
 - Setting level of messages
 - By setting `rootLogger` to either DEBUG, INFO or WARN
 - It is not recommended to set it into different mode as important log information could be lost



- **Configuring log4J**
 - **Setting where to log**
 - **By setting `log4j.appender.A1` to either:**
 - `org.apache.log4j.ConsoleAppender`
 - **Writes to STDOUT**
 - `org.apache.log4j.FileAppender`
 - **Writes to file**
 - `org.apache.log4j.net.SocketAppender`
 - **Sends log output through network to other application**



- **Configuring log4J**
 - **FileAppender**

```
#this options sets the finest level of messages being output
#you may set it to INFO in order to eliminate debugging messages
log4j.rootLogger=DEBUG, A1
```

```
# A1 is set to be a FileAppender which outputs to network socket.
log4j.appender.A1=org.apache.log4j.FileAppender
```

```
# A1 sends traffic to following host & port
log4j.appender.A1.File=log.out
log4j.appender.A1.Append=true
```

```
# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-5p [%t] - %m%n
```

- Configuring log4J
 - SocketAppender

```
#this options sets the finest level of messages being output  
#you may set it to INFO in order to eliminate debugging messages  
log4j.rootLogger=DEBUG, A1
```

```
# A1 is set to be a SocketAppender which outputs to network socket.  
log4j.appender.A1=org.apache.log4j.net.SocketAppender
```

```
# A1 sends traffic to following host & port  
log4j.appender.A1.RemoteHost=localhost  
log4j.appender.A1.Port=4560
```

- Configuring log4J
 - Console & FileAppender at the same time

```
#this options sets the finest level of messages being output
#you may set it to INFO in order to eliminate debugging messages
log4j.rootLogger=DEBUG, A1 , A2
```

```
# A1 is set to be a ConsoleAppender which outputs to System.out.
log4j.appender.A1=org.apache.log4j.ConsoleAppender
```

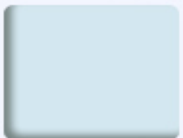
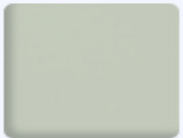
```
# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-5p [%t] - %m%n
```

```
# A2 is set to be a FileAppender which outputs to network socket.
log4j.appender.A2=org.apache.log4j.FileAppender
```

```
# A2 sends traffic to following host & port
log4j.appender.A2.File=log.out
log4j.appender.A2.Append=true
```

```
# A2 uses PatternLayout.
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%-5p [%t] - %m%n
```

Extending CloverETL



- Writing own transformation component
 - Component must be Java class declared as subclass of `org.jetel.graph.Node` base class which in turn subclasses `org.jetel.graph.GraphElement`
 - Component must be accompanied by XML descriptor to be understood by plugging subsystem of Clover
 - To be made available in GUI, GUI XML descriptor has to be created for component

- Writing own transformation component
 - Must provide own implementation of following methods
 - `init()` - *initialize itself*
 - `getType()` - *return component type/name*
 - `run()` - *start performing data transformation*
 - `fromXML()` - *load own set-up from XML data*
 - `toXML()` - *save own set-up as XML data*

More details about methods' signatures can be found in Clover's JavaDoc

- Writing own transformation component
 - method `public void init()`
 - called when phase is being prepared for execution
 - its task is to verify component's configuration and allocate all necessary resources, so component can execute successfully
 - Required to throw `ComponentNotReadyException` in case it can't successfully finish



```

public void init() throws ComponentNotReadyException {
    // test that we have at least one input port and one output
    if (inPorts.size() < 1) {
        throw new ComponentNotReadyException("At least one input port
        has to be defined!");
    } else if (outPorts.size() < 1) {
        throw new ComponentNotReadyException("At least one output port
        has to be defined!");
    }
    recordBuffer = ByteBuffer.allocateDirect(MAX_RECORD_SIZE);
    if (recordBuffer == null) {
        throw new ComponentNotReadyException("Can NOT allocate internal
        record buffer ! Required size:" +
        Defaults.Record.MAX_RECORD_SIZE);
    }
    // create sorter
    newSorter = new SortDataRecordInternal(
        getInputPort(READ_FROM_PORT).getMetadata(), sortKeys,
        sortOrderAscending);
    if (useI18N) {
        newSorter.setUseCollator(true);
    }
    if (localeStr != null) {
        newSorter.setCollatorLocale(localeStr);
    }
}
}

```

- Writing own transformation component
 - method `public String getType ()`
 - called when component's type needs to be determined
 - Must return alphanumeric string which identifies the type of transformation this component performs

```
public final static String COMPONENT_TYPE = "SORT";  
  
public String getType() {  
    return COMPONENT_TYPE;  
}
```

- Writing own transformation component
 - method `public void run ()`
 - executed as independent Thread
 - its task is to perform desired data transformation from/to input&output ports
 - Must update instance variables `resultCode` and `resultMsg` upon finish



```
public void run () {
    InputPortDirect inPort = (InputPortDirect) getInputPort(READ_FROM_PORT);
    boolean isData = true;

    while (isData && runIt) {
        try {
            isData = inPort.readRecordDirect(recordBuffer);
            if (isData) {
                writeRecordBroadcastDirect(recordBuffer);
            }
        } catch (IOException ex) {
            resultMsg = ex.getMessage();
            resultCode = Node.RESULT_ERROR;
            closeAllOutputPorts();
            return;
        } catch (Exception ex) {
            resultMsg = ex.getClass().getName()+" : "+ ex.getMessage();
            resultCode = Node.RESULT_FATAL_ERROR;
            return;
        }
    }
    broadcastEOF();
    if (runIt) resultMsg = "OK"; else resultMsg = "STOPPED";
    resultCode = Node.RESULT_OK;
}
```

- Writing own transformation component
 - method `public Node fromXML ()`
 - Java static method called when component is being deserialized /instantiated from graph's XML representation
 - input parameters are
 - reference to transformation graph
 - its XML's element*
 - return value is object/instance of class implementing component's functionality

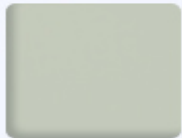
```

public static Node fromXML(TransformationGraph graph, Element xmlElement)
    throws XMLConfigurationException {
    ComponentXMLAttributes xattrs = new ComponentXMLAttributes(xmlElement,
        graph);

    Sort sort;
    try {
        sort = new Sort(xattrs.getString(XML_ID_ATTRIBUTE),
            xattrs.getString(XML_SORTKEY_ATTRIBUTE)
                .split(Defaults.Component.KEY_FIELDS_DELIMITER_REGEX));
        if (xattrs.exists(XML_SORTORDER_ATTRIBUTE)) {
            sort.setSortOrderAscending(xattrs.
                getString(XML_SORTORDER_ATTRIBUTE).matches("^ [Aa].*"));
        }
        if (xattrs.exists(XML_USE_I18N_ATTRIBUTE)) {
            sort.setUseI18N(xattrs.getBoolean(XML_USE_I18N_ATTRIBUTE));
        }
        if (xattrs.exists(XML_LOCALE_ATTRIBUTE)) {
            sort.setLocaleStr(xattrs.getString(XML_LOCALE_ATTRIBUTE));
        }
    } catch (Exception ex) {
        throw new XMLConfigurationException(ComponentType + ":" +
            xattrs.getString(XML_ID_ATTRIBUTE, " unknown ID ") + ":" +
            ex.getMessage(), ex);
    }
    return sort;
}

```

- Writing own transformation component
 - method `public Node toXML ()`
 - called when component is to be serialized /saved to XML representation
 - input parameters are
 - XML's element to which save component's parameters



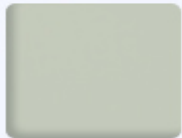
```
public void toXML(Element xmlElement) {
    super.toXML(xmlElement);
    if (sortKeys != null) {
        StringBuffer buf = new StringBuffer(sortKeys[0]);
        for (int i=1; i< sortKeys.length; i++) {
            buf.append(Defaults.Component.KEY_FIELDS_DELIMITER +
                sortKeys[i]);
        }

        xmlElement.setAttribute(XML_SORTKEY_ATTRIBUTE,
            buf.toString());
    }
    if (sortOrderAscending == false) {
        xmlElement.setAttribute(XML_SORTORDER_ATTRIBUTE, "Descending");
    }

    if (useI18N) {
        xmlElement.setAttribute(XML_USE_I18N_ATTRIBUTE,
            String.valueOf(useI18N));
    }

    if (localeStr!=null) {
        xmlElement.setAttribute(XML_LOCALE_ATTRIBUTE, localeStr);
    }
}
```

- Writing engine plugin descriptor
 - all CloverETL's components are treated as plugins – i.e. loaded upon engine start-up from pre-defined location (directory)
 - components are not part of engine's library, are compiled separately
 - descriptor file (XML) is required in order to make component visible and understandable to engine



Extending CloverETL

```
<plugin id="com.XYZ.owncomponent" version="1.9.0" provider-name="XYZ">

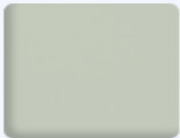
    <runtime>
        <library path="../../../owncomponent/bin"/>
    </runtime>

    <requires>
        <import plugin-id="org.jetel.connection"/>
    </requires>

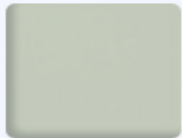
    <extension point-id="component">
        <parameter id="className" value="com.XYZ.owncomponent.Sort"/>
        <parameter id="type" value="OWN_SORT"/>
    </extension>

</plugin>
```

- Writing CloverGUI plugin descriptor
 - GUI treats all Clover components equally – it's the same object internally with different attributes:
 - Icon
 - Number of Input & Output ports
 - Set of parameters/properties of different types
 - String
 - Number
 - Boolean
 - URL
 - ...

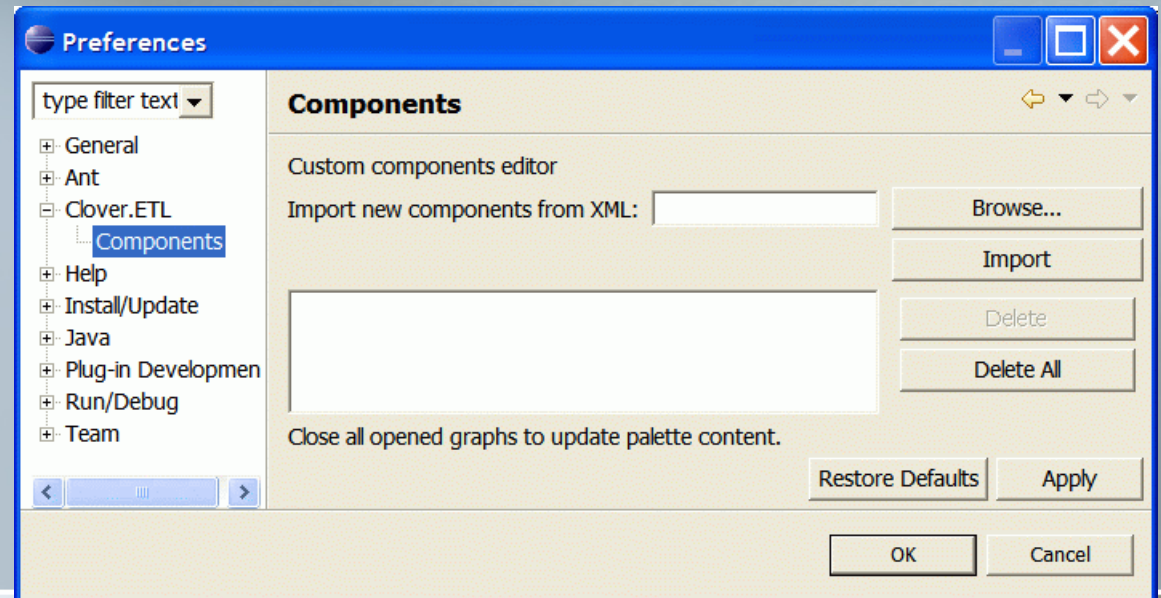


- Writing CloverGUI plugin descriptor
 - In order for GUI to understand particular component, GUI XML component descriptor has to be provided
 - It tells GUI how to handle particular component visually

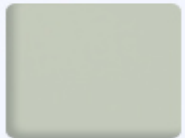


```
<ETLComponentList>
<ETLComponent category="transformers" className="org.jetel.component.Sort"
iconPath="icons/sort32.png" name="Sort" smallIconPath="icons/sort16.png"
type="SORT" passThrough="true">
<shortDescription/>
<description>Sorts the incoming records based on specified key.. </description>
<inputPorts>
    <singlePort name="0"/>
</inputPorts>
<outputPorts>
    <multiplePort/>
</outputPorts>
<properties>
<property category="basic" displayName="Sort order" modifiable="true"
name="sortOrder" nullable="true">
    <enumType>
        <item displayValue="Ascending" value="Ascending"/>
        <item displayValue="Descending" value="Descending"/>
    </enumType>
</property>
<property category="basic" displayName="Sort key" modifiable="true"
name="sortKey" nullable="true">
    <singleType name="key" inputPortName="0"/>
</property>
</properties>
</ETLComponent>
</ETLComponentList>
```

- Writing CloverGUI plugin descriptor
 - The last step is to load component descriptor into GUI
 - In Eclipse, go to `Window->Preferences->Clover.ETL->Components` and import your XML descriptor



Fine-tuning CloverETL



- Clover engine can be adjusted/tuned in several ways
 - By tuning JVM on which engine is executed
 - Changing Default settings of Clover
 - in `defaultProperties` file under `org/jetel/data/` directory of Clover's .jar file
 - Changing some pre-set compile-time constants at individual Java classes

■ Tuning JVM

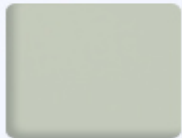
- Except for very small data sets, JVM should be run in server mode – command line option `"-server"`
- Maximum heap size (`-Xmx` option) should be set to appropriately high number (but still leaving some memory for operating system)
- On multi-CPU machines, the parallel garbage collector should be enabled `"-XX:+UseParallelGC"`

Please, refer to your JVM's documentation for optimal parameters !!

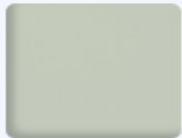
■ Changing Default settings

- `defaultProperties` file located in engine's `.jar` under `org/jetel/data/` directory is place where several default parameters can be found
- These parameters are widely used by Clover when instantiating components and have impact on
 - Engine's performance
 - Engine's memory footprint !!
- Any change to this file affects engine when next executed (through `runGraph` utility)

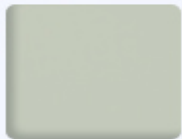
- **Data objects sizes - MAX_RECORD_SIZE (= 8192)**
 - This limits the maximum size of data record in binary form. The binary form is the form used by Clover when manipulating data. Parser are here to convert text representation or database representation of data records to Clover's internal. Some data can have larger representation in text form - dates, numbers and some shorter - strings, for example (java stores strings in unicode - 16bits per character)
 - Note: If you start getting buffer overflow or similar errors, increase this value. The limit is theoretically 2^{31} , but you should keep it under 64K.
 - Clover at runtime allocates several internal buffers of `MAX_RECORD_SIZE`, thus increasing this value increases also memory utilization.



- **Data objects sizes** - `FIELD_BUFFER_LENGTH` (= 512)
 - This is constant for some textual data parsers (and also formatters). It determines what is the maximum size of one particular data field in text format. If you have data containing long text strings, increase this value. Could be 1024,2048 or even higher.
 - The impact on memory utilization is low as each parser your graph uses allocates only one such buffer.



- Data objects sizes - `LOOKUP_INITIAL_CAPACITY` (= 512)
 - The initial capacity (#records) of a lookup table (or internal hash-table for HashJoin) when created without specifying the size at object creation
 - This is only the starting point as the size can grow unrestricted. However when this pre-set size is reached, hash table has to be rehashed which is time&memory consuming task. By setting high-enough size here, you prevent frequent rehashing of lookup tables used within transformation graphs



- Data objects sizes -
DEFAULT_INTERNAL_IO_BUFFER_SIZE (= 32768)
 - This constant determines the internal buffer clover components allocate for I/O operations. Again, increasing this value does not have big impact on overall memory utilization as only few such buffers are used at runtime. There is no sense in increasing this value to speed up something. It has been tested that the performance improvement is negligible. However, if you increase the size of `MAX_RECORD_SIZE` , make sure this value is minimally $2 * \text{MAX_RECORD_SIZE}$.



- Graph objects sizes -
DIRECT_EDGE_INTERNAL_BUFFER_SIZE (= 32768)
 - Size of internal buffer of DirectEdge (the most common edge implementation) for storing data records when transmitted between two components.
The size should be at least `MAX_RECORD_SIZE + 8`, better several times bigger.
 - Higher value (larger buffer) allows particular component to process more data record before it is blocked waiting for previous component to provide more data – thus improving performance as less synchronization of threads is required.



- Graph object sizes -

`DIRECT_EDGE_FAST_PROPAGATE_NUM_INTERNAL_BUFFERS`
(= 4)

- Number of internal buffers for storing/buffering records transmitted through Fast Propagate Edge. One buffer can store one data record. Minimum size is 1. Default is 4. Higher number can help slightly increasing processing speed.



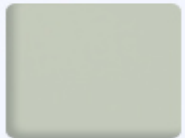
- Graph object sizes -

BUFFERED_EDGE_INTERNAL_BUFFER_SIZE (= 81920)

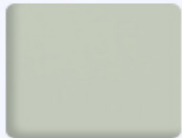
- Size of internal buffer of Buffered Edge for storing/caching data records. Buffered Edge is used when engine needs to compensate fact that component reads data from two different ports and there might be some interdependencies between the source data flows. The size should be at least $MAX_RECORD_SIZE * 10$, better several times bigger - 128kB or more.



Writing transformation code



- CloverETL assumes that structure of data passed between two components is fixed
- If there is a need to change data's structure, it must be done within component which then serves as a bridge between two different data structures
- As of version 2.0 Clover has five components deemed to change structure of data
 - Reformat
 - HashJoin
 - MergeJoin
 - Intersection
 - ApproximativeJoin



- All structure changing components require transformation developer to provide necessary code which changes structure of data from source S to target T
- There are currently three ways transformation code can be assembled
 - Writing Java class implementing RecordTransform interface
 - Using code "pre-processor" for automatically generation Java code from transformation template
 - Using internal scripting language for writing transformation



- Java class performing transformation must implement `RecordTransform` interface from `org.jetel.component` package
 - This interface defines 4 main methods
 - `init()`
 - Called once at the start of component performing transformation
 - `transform()`
 - Called for every pair of input&output which should be transformed
 - `finished()`
 - Called once at the end of component run to indicate that no more data is available
 - `setGraph()`
 - Called once together with `init()` to pass in reference to transformation graph

■ Java class transformation

■ `boolean init()`

■ Passed parameters

- `Properties parameters`

- `DataRecordMetadata sourceMetadata[]`

- `DataRecordMetadata targetMetadata[]`

- It is responsibility of `init()` method implementation to perform whatever initialization is needed – based on provided parameters

- `init()` should return `true` if initialization was successful otherwise `false`

■ Java class transformation

■ `boolean transform()`

■ Passed parameters

- `DataRecord source[]`

- `DataRecord target[]`

- Transform method is responsible for converting data from source records to target records.

- For each input port component has, there is one item in `source[]` array of `DataRecords`

- For each output port, there is also one item in `target[]` array

- `transform()` should return `true` if one step in transformation of source to target was successful, otherwise `false`.

The exact behavior of component if `false` is returned varies

- Reformat skips such pair of records

- HashJoin&MergeJoin cease execution

■ Java class transformation

■ `boolean transform()`

- The exact number of items in `source[]` and `target[]` arrays depend on component's configuration – for example:
 - HashJoin has 2 inputs and 1 output, therefore `source[]` array will always contain 2 items/DataRecords and `target[]` array always 1
 - Reformat has always 1 input port and 1 or more output ports – therefore `source[]` will always contain 1 item
 - If HashJoin is running in *left-outer-join* mode, then there can be master data as item 0 in `source[]` array and no slave data – item 1 of `source[]` (i.e. `source[]` array contains just one data record).

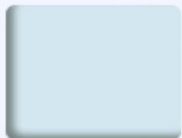
Writing transformation code



// EXAMPLE OF TRANSFORM METHOD

```
public boolean transform(DataRecord _source[], DataRecord _target[]){
    DataRecord source=_source[0];
    DataRecord target=_target[0];
    try{
        SetVal.setInt(target,"OrderID",GetVal.getInt(source,"OrderID"));
        SetVal.setString(target,"CustomerID",GetVal.getString(source,"CustomerID"));
        SetVal.setValue(target,"OrderDate", GetVal.getDate(source,"OrderDate"));
        SetVal.setString(target,"ShippedDate","02.02.1999");
        SetVal.setInt(target,"ShipVia",GetVal.getInt(source,"ShipVia"));
        SetVal.setString(target,"ShipCountry",GetVal.getString(source,"ShipCountry"));
    } catch (Exception ex){
        message=ex.getMessage()+" ->occured with record :"+counter;
        throw new RuntimeException(message);
    }
    counter++;
    return true;
}
```

- DataRecordTransform Java abstract class
 - To ease development of `RecordTransform` interface, skeleton class `DataRecordTransform` is provided which provides default implementation for most interface methods.
 - Only transform method has to be provided by user in his own transformation class which is subclassing `DataRecordTransform`



```
/* EXAMPLE OF COMPLETE IMPLEMENTATION OF JAVA TRANSFORMATION CLASS USING  
DataRecordTransform ABSTRACT CLASS */
```

```
import org.jetel.component.DataRecordTransform;  
import org.jetel.data.DataRecord;  
import org.jetel.data.GetVal;  
import org.jetel.data.SetVal;
```

```
public class reformatOrders extends DataRecordTransform{
```

```
String message;  
int counter=0;  
int field=0;
```

```
public boolean transform(DataRecord _source[], DataRecord _target[]){  
    DataRecord source=_source[0];  
    DataRecord target=_target[0];  
    try{  
        SetVal.setInt(target, "OrderID", GetVal.getInt(source, "OrderID"));  
        SetVal.setString(target, "CustomerID", GetVal.getString(source, "CustomerID"));  
        SetVal.setValue(target, "OrderDate", GetVal.getDate(source, "OrderDate"));  
        SetVal.setString(target, "ShippedDate", "02.02.1999");  
        SetVal.setInt(target, "ShipVia", GetVal.getInt(source, "ShipVia"));  
        SetVal.setString(target, "ShipCountry", GetVal.getString(source, "ShipCountry"));  
    } catch (Exception ex){  
        message=ex.getMessage()+" ->occured with record :"+counter;  
        throw new RuntimeException(message);  
    }  
    counter++;  
    return true;  
}
```

```
}
```

- RecordTransform & GetVal/SetVal “macros”
 - CloverETL library contains two classes which may help when implementing data transformation
 - `org.jetel.data.GetVal`
 - Set of final methods which help with getting values of record's fields
 - `getInt()`, `getDate()`, `getDouble()`, `getLong()`, `getString()`, `isNull()`, `getValue()`, `getValueNVL()`
 - `org.jetel.data.SetVal`
 - Set of final methods able to set values of record's fields
 - `setInt()`, `setDate()`, `setDouble()`, `setLong()`, `setString()`, `setNull()`, `setValue()`

- Other option is to use “pre-processor” which understands simple assignment constructs:

```
${in.record_ordinal_num.field_name} = ${out.record_ordinal_num.field_name}
```

Such script is pre-processed into Java code similar to this:

```
((IntegerDataField)outputRecords[0].getField(OUT0_PROJECT_ID)).setValue(  
    (((IntegerDataField)inputRecords[0].getField(IN0_PROJECT_ID)).getInt()));
```

Scripts can contain mixture of Java code and listed constructs:

```
${out.0.RESPONSIBLEPERSON} = "Person"+${in.1.RESPONSIBLEPERSON};
```

```
${out.0.NAME} = ${in.0.NAME}+${in.1.NAME}.toUpperCase();
```

Following slide shows complete transformation graph with Reformat component using transformation function coded using such script.

Note: more detailed description can be found in

http://cloveretl.berlios.de/docs/CloverETL_components.html section CodeParser

```
<?xml version="1.0" encoding="UTF-8"?>
<Graph name="Testing Reformat">
<Global>
<Metadata fileURL="metadata/orders.fmt" id="InMetadata"/>
<Metadata fileURL="metadata/ordersRef.fmt" id="OutMetadata"/>
<Property fileURL="workspace.prm" id="GraphParameter0"/>
<Property id="GraphParameter1" name="key" value="0"/>
</Global>
<Phase number="0">
<Node enabled="enabled" fileURL="${WORKSPACE}/data/orders.dat" id="INPUT"
type="DELIMITED_DATA_READER"/>
<Node append="false" enabled="enabled" fileURL="${WORKSPACE}/output/orders.dat.out"
id="OUTPUT" type="DELIMITED_DATA_WRITER"/>
<Node enabled="enabled" id="REF" type="REFORMAT">
<attr name="transform">
${out.0.CustomerID} = ${in.0.CustomerID};
${out.0.OrderKey} = 0;
${out.0.OrderID} = ${in.0.OrderID};
${out.0.OrderDate} = ${in.0.OrderDate};
        java.util.GregorianCalendar shippedDateC=new
java.util.GregorianCalendar();
        shippedDateC.setTime(${in.0.OrderDate});
        shippedDateC.add(java.util.Calendar.DAY_OF_YEAR,7);
${out.0.ShippedDate} = shippedDateC.getTime();
${out.0.ShipVia} = ${in.0.ShipVia};
${out.0.ShipTo} =
${in.0.ShipName}+${in.0.ShipAddress}+${in.0.ShipCity}+${in.0.ShipCountry};
</attr>
<attr name="key">0</attr>
</Node>
<Edge fromNode="INPUT:0" id="INEDGE" metadata="InMetadata" toNode="REF:0"/>
<Edge fromNode="REF:0" id="OUTEDGE" metadata="OutMetadata" toNode="OUTPUT:0"/>
</Phase>
</Graph>
```

- Third option to write transformation code is to use internal scripting language (with syntax similar to C or Java) – code named TL
 - TL provides all common statements
 - `if-else`
 - `case`
 - `while`
 - `for`
 - `do-while`
 - Offers “native” Clover data types –
`int, long, number, decimal, date, string, boolean`
 - Supports writing user functions
 - Has many built-in functions and procedures which can be used in transformation code

- Third option to write transformation code is to use internal scripting language (with syntax similar to C or Java) – code named TL
 - TL provides all common statements
 - `if-else`
 - `case`
 - `while`
 - `for`
 - `do-while`
 - Offers “native” Clover data types –
`int, long, number, decimal, date, string, boolean`
 - Supports writing user functions
 - Has many built-in functions and procedures which can be used in transformation code

- TL is interpreted language which brings some benefits and some drawbacks
 - + Transformation can be assembled much faster in TL than in Java, it is focused on data manipulation
 - + It allows more complicated transformation to be written than “pre-processor” allows
 - + Error reporting is more accurate&meaningful than when Java code is used
 - Being interpreted it is slower than running Java code or using “pre-processor” as it translates code into Java and then compiles it

Note: more information is present in following document:

http://cloveretl.berlios.de/docs/TL_doc/CloverETL_transformation_language.html

- Transformation in TL
 - Must implement at least `transform()` function which is called for each pair of input&output records
 - If `init()` and `finalize()` functions are present then they are called at their respective times



```
<?xml version="1.0" encoding="UTF-8"?>
<Graph name="Testing Reformat">
<Global>
<Metadata fileURL="metadata/orders.fmt" id="InMetadata"/>
<Metadata fileURL="metadata/ordersRef.fmt" id="OutMetadata"/>
</Global>
<Phase number="0">
<Node enabled="enabled" fileURL="{WORKSPACE}/data/orders.dat" id="INPUT"
type="DELIMITED_DATA_READER"/>
<Node append="false" enabled="enabled" fileURL="{WORKSPACE}/output/orders.dat.out"
id="OUTPUT" type="DELIMITED_DATA_WRITER"/>
<Node enabled="enabled" id="REF" type="REFORMAT">
<attr name="transform">
int key; // define global variable, will be used as counter for generating IDs
key=-1; // assign value to it
/* sample function, just to show how things work */
function sum(a,b){
    return a+b;
}
function transform()
{
$0.CustomerID := $0.CustomerID;
$0.OrderKey := ++key;
$0.OrderID := $0.OrderID;
$0.OrderDate := $0.OrderDate;
$0.ShippedDate := sum($0.OrderDate,7); // we ship 1 week later :-
$0.ShipVia := $0.ShipVia;
$0.ShipTo := $0.ShipName+$0.ShipAddress+$0.ShipCity+$0.ShipCountry;
}</attr>
</Node>
<Edge fromNode="INPUT:0" id="INEDGE" metadata="InMetadata" toNode="REF:0"/>
<Edge fromNode="REF:0" id="OUTEDGE" metadata="OutMetadata" toNode="OUTPUT:0"/>
</Phase>
</Graph>
```

End

