NETVIBES | Exalead CloudView

CloudView CV23

Consolidation

Table of Contents

Consolidation Server

This guide explains how to deploy and configure consolidation for source connectors.

The Consolidation Server supports all kinds of connectors.

Audience

This guide is mainly destined to software programmers or users with a few programming skills in Java or Groovy.

Further Reading

You might need to refer to the following guides:

Guide	for more details on
Connectors	standard connector's configuration.
Configuration	indexing and search concepts, as well as advanced functionality.

What's New?

There are no enhancements in this release.

About the Consolidation Server

This chapter describes the Consolidation Server components and the processing pipeline workflow.

Why Use Consolidation

Like most search engines, Exalead CloudView has a simple data model to provide good performance at query time. Unlike relational databases, it has only one table. This allows Exalead CloudView to have minimal query latency even on a very large corpus, but things get more difficult in the indexing phase when the original data model is more complex than what Exalead CloudView can support.

Object trees, often based on several relational database tables, have to be flattened to be used efficiently in Exalead CloudView.

Consolidation is very helpful when indexing relational data and handling this flattening during an incremental index build. In other words, it takes updates as they come instead of rebuilding the entire index when an object changes. The incremental update is a complex task as it requires calculating the impact of any changes and building complete documents according to projection rules. To do so, the Consolidation Server keeps track of object relationships and stores data to rebuild Exalead CloudView documents.

Note: The Consolidation Server is not limited to one data source. It can work across several data sources, which allows building documents based on objects coming from different sources. This avoids having an ETL or an equivalent tool to perform cross-source joins and aggregations.

Consolidation Server Terminology

This section describes the most important terms and concepts of the Consolidation Server.

- CDIH (Consolidated Document Identifier Holder) – is similar to the Indexing Server DIH. It assigns unique IDs to the documents processed by the Consolidation Server.

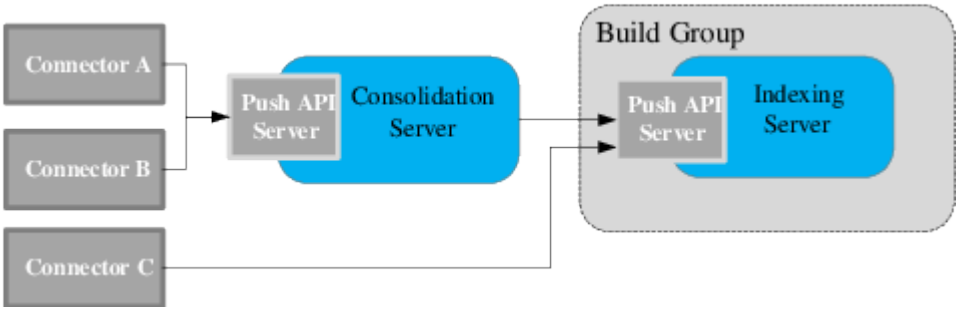
- Consolidation config – A consolidation config specifies consolidation settings, some applying to transformation and aggregations processors. You can also specify rules to forward consolidated documents to another build group or Consolidation Server.
- Transformation processors – Use transformation processors to specify the relationships between the objects pushed to the Consolidation Server. The documents and their relationships are stored using an object graph, where documents are nodes and relationships are arcs.
- Aggregation processors – An Aggregation processor is a set of rules describing how to build the documents sent to Exalead CloudView. They allow the Consolidation Server to enrich object graph documents with the metas of their related nodes. You can write these rules in Groovy or Java.
- Documents - All the objects to index, regardless of file or entity type in the data source. For example, HTML, JPG or CSV files, database records are all considered documents within Exalead CloudView, since they are all converted into a Exalead CloudView-specific document format (also known as a PAPI document) after being scanned by a connector.

How the Consolidation Server Fits into Exalead CloudView

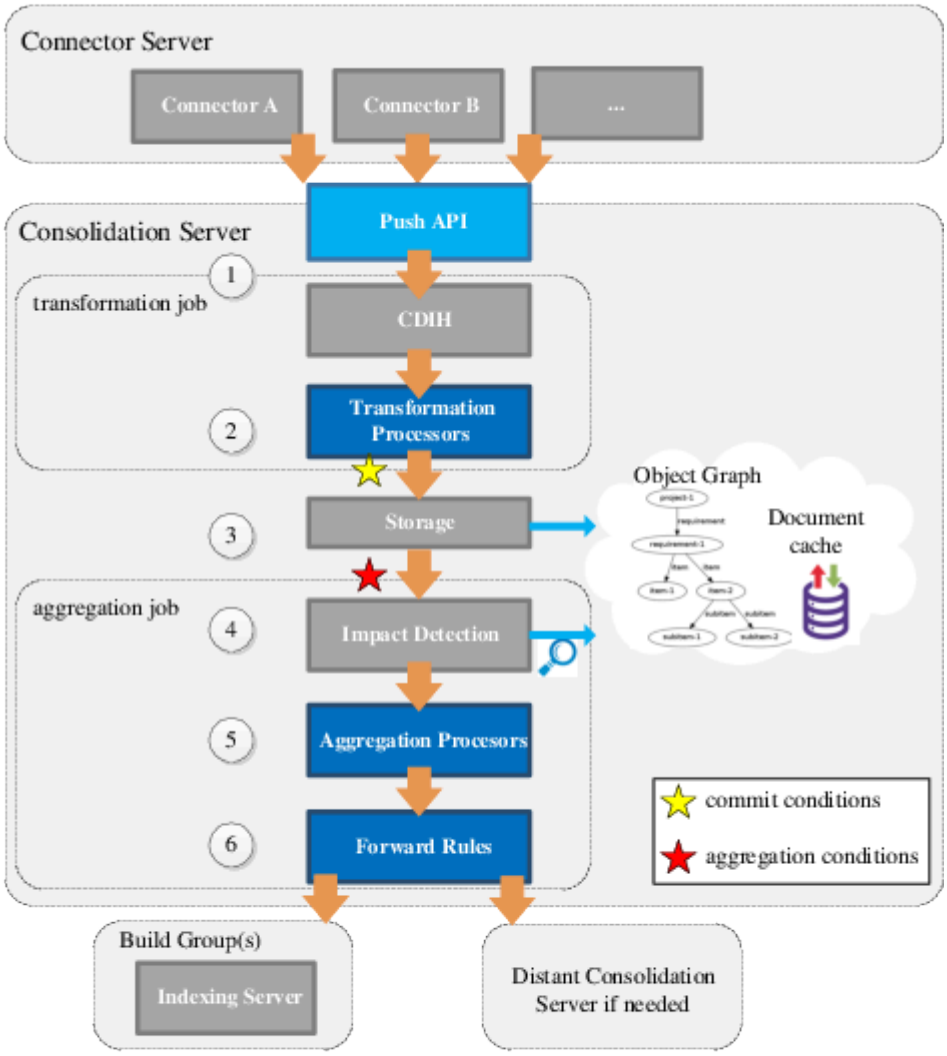
A Exalead CloudView installation is made up of one or several build groups, with connectors feeding the build groups with documents. The Consolidation Server allows you to define consolidation rules for documents before pushing them into the build group Indexing Server.

The Consolidation Server therefore fits before the build group Indexing Server or before another Consolidation Server if a specific forward rule indicates to do so. You can view it as a transformation phase between source connectors and the Indexing Server.

For each connector, you can choose to enable consolidation. You can therefore use the Consolidation Server for a set of connectors and not for other connectors as shown in the following diagram.



The following diagram illustrates the consolidation workflow within the Consolidation Server, when connectors push documents.



Step	Description
1	Connectors push an `addDocuments` bulked order through HTTP to the Consolidation Server: • Documents arrive in the Consolidation Server • Source URIs are added to the CDIH
2	Documents go through the transformation processing.
3	As soon as commit conditions are met (see yellow star on the diagram) OR when one of the source connectors sends a synchronization order, all changes are persisted to disk in the Consolidation storage. Its purpose is to store transformed documents and the updated object graph.

Step	Description
4	As soon as aggregation conditions are met (see red star on the diagram) the Impact detection is launched on new object graphs. It detects the nodes of the existing graph that have to be aggregated again.
5	Once the impact detection is complete, aggregation processing can be launched for all detected nodes. The processing depend on their type and the action context.
6	As soon as a document is aggregated, it is pushed to the target defined in the existing forward rules. The target can be an Indexing Server or another Consolidation Server.

Note: The reception order of ADD/DELETE operations for a given document is respected all along the processing chain. For example, if a connector sent an ADD order for document and then a DELETE order, the Consolidation Server will also send an ADD order and a DELETE order to the Indexing Server.

About the Consolidation Object Graph

Consolidating documents requires a means of defining the relationships between these documents. To do so, the Consolidation Server uses an object graph, in which each node corresponds to a document.

The node identifier is the document URI. Arcs represent document relationships by linking the nodes with one another.

Note: In this section, document refers to a Exalead CloudView document.

Object Graph and Index Incremental Updates

Using an object graph, a set of document modifications, and aggregation rules, the Consolidation Server determines which documents have been impacted by changes to update the Exalead CloudView index incrementally. By doing so, the Consolidation Server is able to limit the graph traversal and only parse relevant relationships.

The Consolidation Server is able to:

- Enrich a node with "related" data coming from related nodes.
- Aggregate many related nodes' attributes into a single node.
- Correctly handle incremental updates and recompute nodes whose "related" data have been modified (that is to say, added, deleted, updated).

*Object Graph as Displayed in the **Consolidation > Introspect** Tab*

Object Graph Node

An object graph node is made of the following string properties:

- A unique identifier – the Exalead CloudView document URI.
- A set of types, ordered from the most specific to the most generic type. For example, Cat > Animal > Living Form. The node type is used to determine which rules are going to be applied (transform, aggregate or forward).

Finally, when pushing a document (node) to the index, you can define the type explicitly. If not, the default data model class associated to the connector will be used.

Inside transformation processors, it is also possible to create nodes explicitly. See Manage Documents Explicitly.

Object Graph Arcs

An object graph arc requires three String properties:

- The source node URI, which is the key of this object
- The destination node URI
- The relationship name of the arc indicating the arc direction

Object graph arcs represent document relationships specified by transformation processors. However, several connectors with specific schemes (ENOVIA, SalesForce, etc.) specify both

nodes and arcs using custom directives. Therefore, the Consolidation Server supports custom directives directly sent by source connectors.

Recommendation: To use custom directives, use the new `com.exalead.cloudview.consolidationapi.PUSHAPITransformationHelpers.java` documented in the javadoc.

Object Graph Matching Expressions

In the aggregation phase, processors can benefit from the object graph arcs to access objects linked to the processed document, using any path connecting the objects together.

The Consolidation Server provides a dedicated grammar to build complex path expressions.

Once a matching rule is used inside an aggregation processor, it can be also used for the impact detection step. It ensures that when updating document A used by document B inside an aggregation processor, document B is processed again to ensure that the change is correctly reflected. For more information, see Impact Detection.

Configuring the Consolidation Server

This chapter describes the Consolidation Server deployment and configuration in Exalead CloudView.

The configuration procedures focus on the actions to follow but do not contain examples. For detailed common examples, see Use Cases.

Deploying the Consolidation Server

This section describes how to add and deploy a Consolidation Server in Exalead CloudView.

Add Consolidation Support at Exalead CloudView Installation Time

1. When finishing Exalead CloudView installation with the setup wizard, in the **Processing** screen, select **Set up a consolidation server role with standard configuration**.

   Note: You can also enable the support of consolidation directly after Exalead CloudView installation by launching the post-installation script `<DATADIR>/bin/postinstall` with the `--consolidation true` option.

   It creates a Consolidation Server instance (`cs0`) that sends its documents to the default build group (`bg0`).

Add Consolidation Support Manually

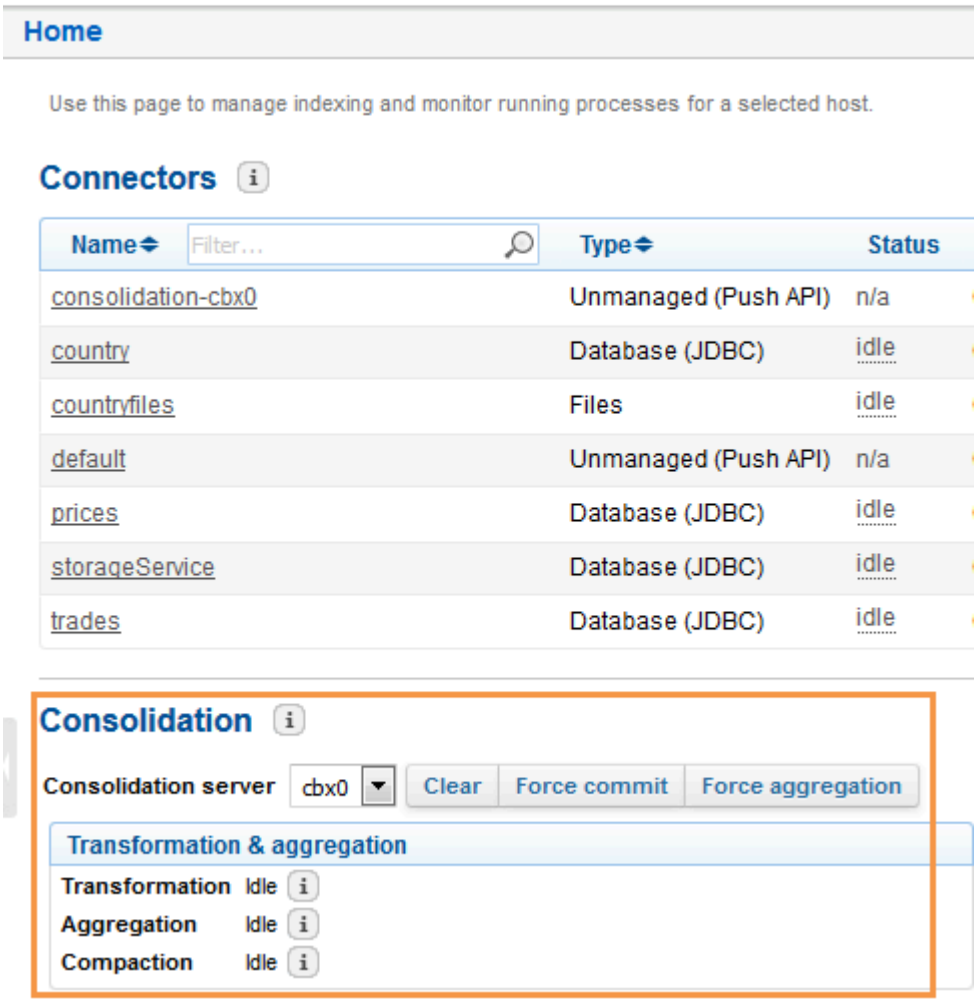
1. In the Administration Console, go to **Deployment > Roles**.
2. Add a new **Consolidation server** role.
3. Expand the Consolidation server role and define an **Instance name** for this Consolidation Server.

   You cannot change the Consolidation Server instance name once created.
4. Apply the configuration.

   Now that the Consolidation Server is deployed, connectors can target its Push API. For more information, see the Exalead CloudView Connectors Guide.

5. Go to the **Home** page.

   You can now see a **Consolidation** section below the **Connectors** section.



Enable Consolidation on Source Connectors

1. For each source connector on which consolidation must be applied, go to the **Deployment** tab.
2. For **Push to PAPI server**, select the Consolidation Server instance on which the connector must push its documents.
3. Apply the configuration.

Configuring the Consolidation

This section describes the overall Consolidation Server configuration. Details and examples are given further in this guide.

Configuring the Processors

Trigger and Synchronize Consolidation

Forwarding Documents to Other Build Groups

Configuring the Processors

By default, the consolidation configuration pushes the documents they receive without transformation.

For more information, see Writing Transformation and Aggregation Processors and the examples provided in Use Cases.

Define a Consolidation Configuration

1. Go to **Index > Consolidation**.
2. The documents received by the Consolidation Server first go through **transformation processor(s)**. The purpose of this transformation step is mainly to add arcs between documents to create the object graph.

   Note: If you are using a custom connector, you can configure it to handle the generation of arcs directly. For more information, see "Consolidation Server directives" in the Exalead CloudView Connector Programmer's Guide.

3. The second step is the definition of the **aggregation processor(s)**, which creates a consolidation view on top of the object graph.
4. Once processors are defined, click **Apply**.
5. Go to the **Home** page and under the connectors list, click **Scan** for the connectors managed by the Consolidation Server.

In the **Connectors** list, a **consolidation-<instance name>** row displays status information about consolidation.

Take into Account New Transformation Processors

To take into account changes made on your transformation processors, you need to rescan the impacted sources. You can clear the sources and scan them again or clear the Consolidation Server as described in Clearing the Consolidation Server.

1. Go to the **Home** page.
2. Clear the source connectors' documents.
3. Re-**scan** your connectors.

Take into Account Aggregation Processors

1. In the Administration Console, go to the **Home** page
2. Clear the index.

3. Under **Consolidation**, start a **Force aggregation** operation.

   Note:

   You can also start **Force aggregation** from the API Console.

   A force aggregation behaves as a commit operation. The consolidation storage is fully synchronized at the end of the operation.

   If you specify a type, the force aggregation operation is not managed as a commit operation. If the consolidation storage has not yet been synchronized (either by triggering an aggregation or with a force commit operation), it stays in the same state after the operation. It only aggregates the targeted content of the consolidation storage.

   Important: Dynamically computed impact rules are based on old aggregation jobs. If you change your aggregation processors, these rules may no longer be consistent. To get back to a correct behavior, you either have to start a full **Force aggregation** operation or clear the Consolidation Server and rescan all its source connectors.

Trigger and Synchronize Consolidation

This section describes how consolidated data is sent to the index.

Commit triggers define when to write documents to the index. You can link commit conditions to inactivity, number, or size of documents, or elapsed time.

Aggregation triggers define when transformed documents and documents stored or synchronized in the Consolidation Server storage are aggregated. You can also link these conditions to inactivity, number, or size of documents, or elapsed time. Once complete, the result of the aggregation job is sent to the target Indexing Server specified in the **Forward rules** section.

When launching a **Force commit** operation, you commit the transformation job and then start an aggregation.

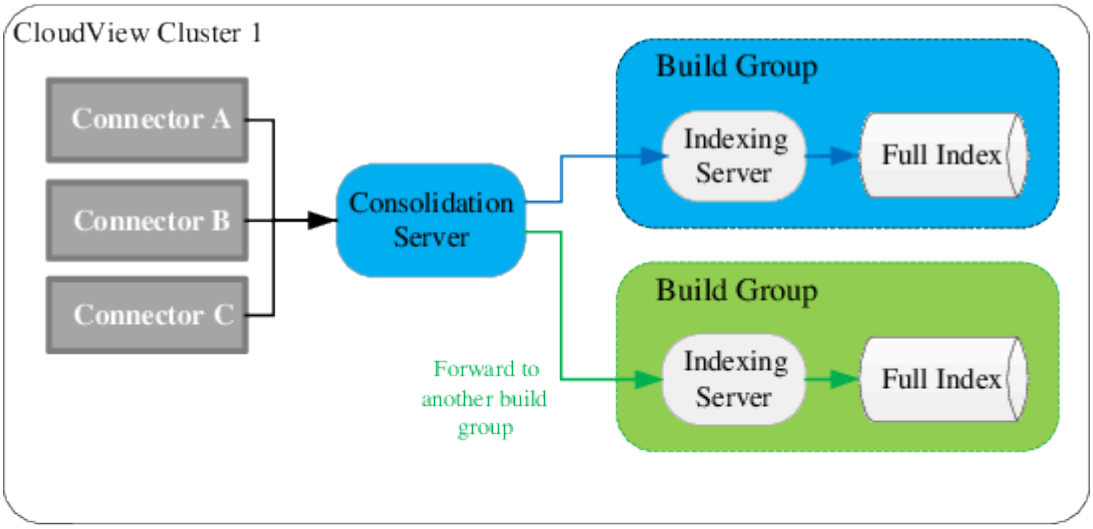
Important: By default, a connector does not send a synchronization order to the Consolidation Server when its scan is finished. To enable this behavior, go to **Connectors > Deployment > Push API** and select the **Force indexing after scan** option.

Forwarding Documents to Other Build Groups

By default, consolidated documents are forwarded to the Indexing Server of a specific build group, for example, bg0. However, for advanced cases, consolidated documents can also be useful for several build groups or other Consolidation Servers. To fulfill these needs, you can define forward

rules in your Consolidation configuration to forward consolidated documents to the target of your choice.

The following diagram shows the forward of consolidated documents to another build group on the same Exalead CloudView instance. Documents are sent to another Indexing Server and stored in another Index.



Important: Delete orders are pushed to all build groups without checking forward rules.

Write Forward Rules in the Administration Console

1. Go to **Index > Consolidation > Forward rules**.
2. In **Forward to**, select the target build group or Consolidation Server on which you want to forward consolidated documents.
3. In the **Document types** field, enter the comma-separated list of document types that to forward, that is to say the document types specified in the transformation and aggregation processors. Leave this field empty to forward all document types.
4. Select **Trigger indexing** if you want to trigger an indexing job on the target build group or Consolidation Server automatically. This bypasses the commit conditions defined on the target build group or Consolidation Server.
5. Click **Apply**.
6. Clear and reindex your documents with the main Consolidation Server.

Write Forward Rules in the API Console

1. Open the Exalead CloudView API Console, `<HOSTNAME>:<BASEPORT+1>/api-ui/`
2. Click **Manage**.
3. Search for the **setConsolidationConfigList** method.
4. Edit the `AggregationForwardProcessorConfigList` node to write your forward rules.

```
<conso:ConsolidationConfig ...>
  <conso:AggregationForwardProcessorConfigList>
    <conso:AggregationForwardProcessorConfig triggerIndexing="true" pushAPIServer=
  </conso:AggregationForwardProcessorConfigList>
</conso:ConsolidationConfig>
```

5. Click **Save**.
6. Click **Apply**.
7. Clear and reindex your documents with the main Consolidation Server.

Clearing the Consolidation Server

To clear the Consolidation Server content, you have the choice between the following options in the Administration Console **> Home** page.

You can perform one of the following actions:

Action	To ...
Home > Connectors > Clear documents for specific connectors pushing to the Consolidation Server.	Notify the object graph and the document storage to take this change into account and send the proper deletion orders to the Indexing Server. **Important:** Impact analysis is performed on any deleted document so it might take some time for large sources.
Consolidation > Clear	Clear consolidated data from the Consolidation Server. The Consolidation Server then sends delete orders to its aggregation targets, that is to say, the target Indexing Server or another Consolidation Server (if you specified forward rules). If you want to accelerate the process and if possible, it is better to clear the index first and then clear the Consolidation Server.
Clear documents action for **consolidation-<instance name>**	Clear from the index all documents previously pushed by all the Consolidation Server connectors. This action deletes all consolidated documents from the Indexing Server but not from the Consolidation Server. Therefore, we do not recommend this option

Action	To ...
	as it may lead to inconsistent states between the Consolidation Server and the Indexing Server. Yet it can be useful, if you then launch a force aggregation action to make sure that the Indexing Server does not contain results of previous aggregations.

Tuning and Sizing the Consolidation Server

Though the object graph is serialized on disk, it is also fully sent to memory for performance reasons.

Tuning

Basic Tuning

In the Administration Console, you can adjust:

- The number of aggregation threads in **Consolidation > Advanced Settings**. For example, if you set it to `4`, you get 4 transformation workers, 4 aggregation workers, and 4 forwarders (* by number of forward rules), all potentially running in parallel for an incremental batch.
- Your commit conditions to fit your current scenarios

Get the Initial Scan Recommended Settings

1. In **Consolidation > Commit triggers**, specify a commit trigger based on **No. of tasks** to 500,000 tasks.
2. Specify a commit trigger based on **Inactivity** set to 1 task and 60s of inactivity.
3. Add an Aggregation trigger based **Inactivity** set to 1 task and 60s of inactivity (so your aggregation starts at the end of your initial push).

Get the Incremental Scan Recommended Settings

1. In **Consolidation > Commit triggers**, specify a commit trigger based on **No. of tasks** to 50,000 tasks.
2. Specify a commit trigger based on **Inactivity** set to 1 task and 60s of inactivity.
3. Adjust your aggregation triggers to fit your required freshness.

Advanced Tuning

In your `<DATADIR>/config/Consolidation.xml` file, you can add an `AdvancedConfig` node to `ConsolidationConfig` to tweak internal queues used during aggregation. It might increase throughput with more buffering, but you must take it into account in your sizing.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<conso:ConsolidationConfigList version="0" xmlns:bee="exa:exa.bee"
xmlns:edit="exa:com.exalead.editor.v10" xmlns:index="exa:com.exalead.mercury.mami.ind
xmlns:conso="exa:com.exalead.mercury.mami.consolidation.v10" xmlns:config="exa:exa.be
  <conso:ConsolidationConfig name="cc0_standard" nbThreads="4"
maxNativeMemoryConsumptionThreshold="enabled" maxNativeMemoryConsumptionInMB="2048">
 ...
    <conso:AdvancedConfig>
      <conso:AdvancedAggregationConfig impactQueueSize="8" aggregationQueueSize="8"
forwardQueueSize="800" />
    </conso:AdvancedConfig>
  </conso:ConsolidationConfig>
</conso:ConsolidationConfigList>
```

Default values for `AdvancedAggregationConfig` are:

- Impact Queue size / Aggregation Queue size = number of threads * 2
- Forward Queue size = number of threads * 200

Sizing

Heap Sizing (Estimation)

Process	Sizing formula
Transformation	`MAX_PAYLOAD_SIZE * NB_THREADS * 8192`
Impact Detection	`((VERTEX_SIZE + ((MAX_PATH_LENGTH * MAX_PATH_COUNT) * (VERTEX_SIZE + (MAX_ARC_COUNT_PER_VERTEX * ARC_SIZE)))) * NB_THREADS) + (IMPACT_QUEUE_SIZE * VERTEX_SIZE)`
Aggregation	`((MAX_PAYLOAD_SIZE + ((MAX_PATH_LENGTH * MAX_PATH_COUNT) * (VERTEX_SIZE + (MAX_ARC_COUNT_PER_VERTEX * ARC_SIZE)))) * NB_THREADS) + (AGGREGATION_QUEUE_SIZE * VERTEX_SIZE)`
Forward	`MAX_PAYLOAD_SIZE * (FORWARD_QUEUE_SIZE + (FORWARD_RULES_COUNT * 100))`

Process	Sizing formula
Caching	`10 MB * NB_THREADS`
VERTEX_SIZE (in bytes)	`URI_SIZE + (TYPE_COUNT * TYPE_SIZE)`
ARC_SIZE (in bytes)	`TARGET_URI_SIZE + ARC_TYPE_SIZE`
PAYLOAD_SIZE (in bytes)	`(META_COUNT * (META_KEY_SIZE + META_VALUE_SIZE)) + (DIRECTIVE_COUNT * (DIRECTIVE_KEY_SIZE + DIRECTIVE_VALUE_SIZE)) + (PART_COUNT * (PART_KEY_SIZE + PART_VALUE_SIZE))`

Hardware Sizing

Your graph structure on disk MUST fit in your system memory. Check the size of your `<DATADIR>/build/consolidation-INSTANCE/sdc-storage/objectgraph`.

Writing Transformation and Aggregation Processors

This chapter describes the elementary bricks to write consolidation and aggregation rules.

About Document Processing

You can write transformation and aggregation processors in several languages.

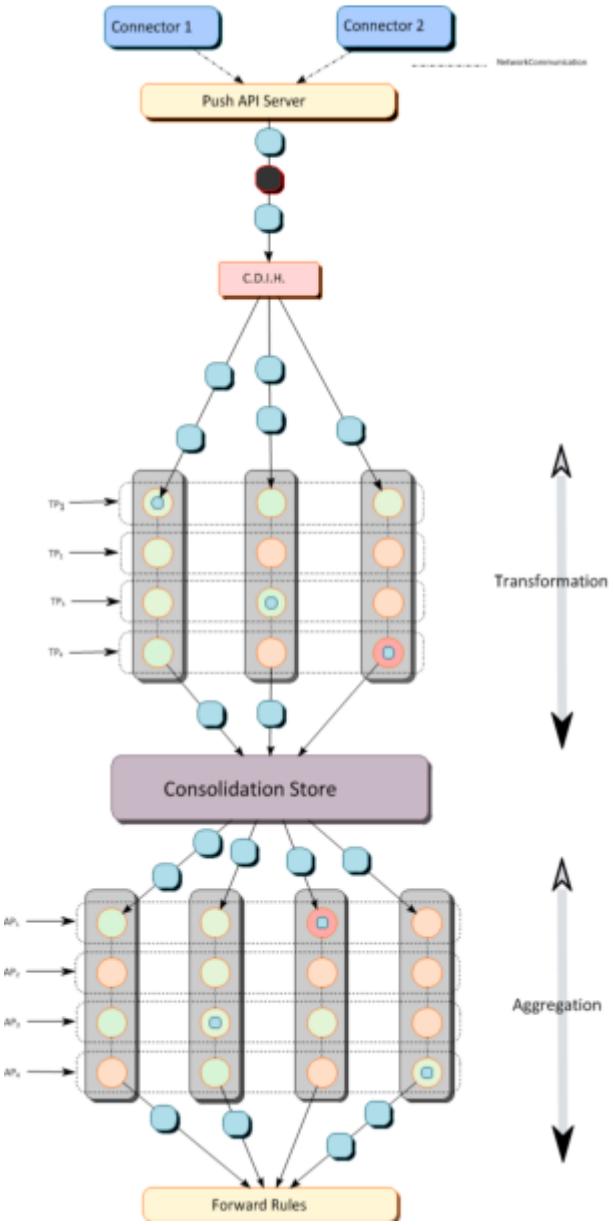
The Consolidation Server supports:

- Groovy – The optimal programming language for writing short rules.
- Java – The language developers are most accustomed to. It is more suitable for production than Groovy.
- DSL – The Domain Specific Language used in Exalead CloudView V6R2014x, which is still supported in legacy mode.

  Important: This chapter focuses on the use of Java. For information on the use of Groovy or the legacy DSL, see Appendix - Groovy Processors and Appendix - Old DSL Functions.

Document Processing in the Consolidation Server

The following diagram gives a detailed view of document processing in the Consolidation Server.



At the top level, connectors send documents to the Consolidation Server. The PushAPI Server receives them and first pass them to the Consolidated Document Identifier Holder (CDIH), which assigns them unique IDs.

Note: If we send a delete order for a particular document that the CDIH does not know, the order does not even proceed to the transformation processors. This is the case for the document depicted in black in the picture.

For each transformation thread, the PushAPI Server then dispatches them to a list of transformation threads. In the processing chain of one transformation thread, a document tries to be applied on all defined processors (here 4 in the diagram, 1 <= TPi <= 4). We say "try" since, as we will see later, you can associate a processor code to a particular document type hierarchy. As a result, some processors are skipped (colored in orange) and others are selected (colored in

green) depending on the document type. For more information, see Processor Type Inheritance and Runtime Selection.

At execution time, once the document is transmitted to a transformation processor, it is then automatically passed to the next available and valid processor... unless told otherwise (using a `discard` call). This the case, for example, for the processor highlighted in red where the document is not transmitted to the next phase (either next processor or here the Consolidation Store). Clearly when making such decision, this document does not participate to the Aggregation Phase.

The Consolidation Store stores all the documents pushed to it as well as the potential relationships created at the transformation phase.

Once some documents are available in the Store, the aggregation phase can start, independently of the transformation phase. So, the transformation and aggregation phases are performed in parallel. And similarly to the transformation phase, the aggregation is concurrently applied using a number of threads defined at configuration time. The logic of selection and processing is then totally similar to the one described for the transformation. The difference is that in this phase:

1. We execute aggregation processors (here 4, 1 <= APi <= 4),
2. Then documents are passed to the forward rules handler,
3. The forward rules handler ultimately route (or not) consolidated documents to the Indexing Servers or to other Consolidation Servers.

Processor Action Context

You have to define **an action context** for each processor in the Consolidation Server pipeline.

There are two different action contexts to specify the action performed on documents:

- **create/update**: to create or update documents coming from one or more connectors or the Consolidation Store.
- **delete**: to delete documents from the connectors or the Consolidation Store.

Delete Action Context

This is what occurs in the Consolidation Server when connectors push delete orders to remove documents from the Indexing Server:

- If you defined a processor with a delete action context that matches the document types, the processor code is executed and yields to the next processor or stage, unless a `discard` operation is specified.

- If you did not define a processor with a `delete` action context, or if it does not match the document types, the document proceeds as if a default processor was defined with auto-yielding. This behavior is true for both transformation and aggregation phases.

In other words, unless a delete processor has been defined and matches the document types, when connectors push delete orders, the Consolidation Server:

1. Pushes a delete order to the Consolidation Store and removes documents from it.

2. Pushes delete orders to the Indexing Servers and removes them from the Indexing Store.

In addition, default delete orders are also applied to all child documents.

Delete Orders in Create/Update Action Context

You can also perform delete orders in a create/update action context, using `deleteDocument` operations. This is mostly in the Aggregation Phase that such operations can be useful. Indeed, we recommend controlling the presence of documents in the Consolidation Store with orders coming from the connectors.

Control the Processing

In Java and Groovy, the evaluation of documents in the list of transformation and aggregation processors is:

- Ordered: They are processed in the order they are defined.
- Automatic: Processed documents are allowed to pass to the next processor or the next stage automatically without declaring a `yield` operation. However, you must yield explicitly all documents created inside a processor. Calls to `delete` operations are automatically yielded.

Since the document is automatically passed to the next processor or the next stage available in the processing pipeline, you must make a call to the `discard` method to prevent it from going further.

This method stops the pipelining. If the document was already present in the Consolidation Store, discarding it at the transformation phase does not delete it from the Consolidation Store. If you want to discard it and ask for deletion, you can add a `delete` operation in the processor where the `discard` operation occurs.

Important: As for the `yield` method, the `discard` method does not interrupt the runtime execution flow of your processor.

In the following code snippet, the code after `discard` is executed. If you want to interrupt the flow, you have to add a `return;` after the `discard` call. The documents to yield in an aggregation

processor are the current processed document and, potentially the documents created during the process code execution.

Recommendation: Do not yield other documents that could have been grabbed using a match function. Doing so would lead to undefined behavior on the receiving end (Indexing Server for example).

```
@Override
public void process(final IJavaAllUpdatesAggregationHandler handler, final IAggregati
 throws Exception {
  ...  if (someCondition) {
    ...    discard();
  }
  // Some other calls
  ...
}
```

For more code samples, see Discard Processor Code Samples.

Processor Type Inheritance and Runtime Selection

Every document, within the transformation or aggregation phase, has at least one type, but possibly more. You can define a type inheritance for each of them.

To do so, see: `IMutableTransformationDocumentParameterized.setType`, `ICreateTransformationHandler.create'*'`, `ICreateAggregateHandler.create'*'`.

For example, you could write:

```
@Override
 public void process(final IJavaAllUpdatesTransformationHandler handler, final IMutab
 document) throws Exception {
 document.setType("cat", "felid", "mamal", "vertebrate");
  ...
}
```

As a result, the processors selected for execution apply the following rules:

- Either the transformation or aggregation has the all types pattern.
  - In Java, this is achieved by returning null or an empty string.
  - In Groovy, this is achieved with an empty string.
- Or the document type inheritance matches the defined processor type.

With the following sequence of Groovy aggregation processors, the document presented before is executed in order within Processor 1, Processor 3, and Processor 4.

Processor #	Groovy code
1	`process("cat") {  ...}`
2	`process("dog") {  ...}`
3	`process("mamal") {  ...}`
4	`process("") {  ...}`

Java Processors

Every Java Processor defined in the Consolidation Server implicitly implements the `CVComponent` Exalead CloudView interface, required to define a Exalead CloudView Component.

For more information, see the "Creating custom components for CloudView" in the Exalead CloudView Programmer's Guide.

Consequently, it is possible to:

- Create Java processors externally within your IDE,
- Package this appropriately in a Jar/Zip,
- And deploy it into the Exalead CloudView instance to enable your processors selectively.

This is one of the key advantages over Groovy, as Groovy processors are added and written within the Exalead CloudView Administration Console. With the Exalead CloudView component mechanism, you can also define runtime properties that to customize the component behavior. It thus becomes possible to write a generic processor that can be customized using runtime properties defined within the Administration Console later on.

Define Java Transformation Processors

You can define transformation processors using a set of default processors made for generic simple operations, or through custom java code if your needs are more specific.

Use Default Transformation Processors

1. Under **Transformation processors**, click **Add processor**.
2. In **Add processor**, select **Java**, give a **name** to the processor, and then choose one of the following default processors.

Transformation Processor	Description
Basic Arc Creation Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.javaCreateArcBasedOnMetaValueTransformationProcessor` Creates an arc from the processed document. The target is the value of the given meta name.
Basic Document Creation Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.javaCreateDocumentBasedOnMetaValueTransformationProcess` Creates a managed document from the processed document. The target is the value of the given meta name.
Set Directive Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.javaSetDirectiveTransformationProcessor` Sets the given directive on the processed document
Set Meta Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.javaSetMetaTransformationProcessor` Set the given meta on the processed document
Set Type Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.javaSetTypeTransformationProcessor` Sets the given type on the processed document
Split Text Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.javaSplitTextTransformationProcessor` Splits the given source meta using the specified delimiting regex pattern, and add/set the result to the target meta.

Transformation Processor	Description
	Note: The target meta must be multivalued to contain all text chunks resulting from the split operation.
Storage Service Key Linker Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.javaStorageServiceKeyLinkerTransformationProcessor` Create arcs between the Storage Service data and the document it is linked to. For a use case example, see UC-8: Consolidating Data from Storage Service.

3. Click **Apply**.

Create Custom Transformation Processors

To define a Java Transformation processor in the create/update action context, you need to implement the `IJavaAllUpdatesTransformationProcessor` interface.

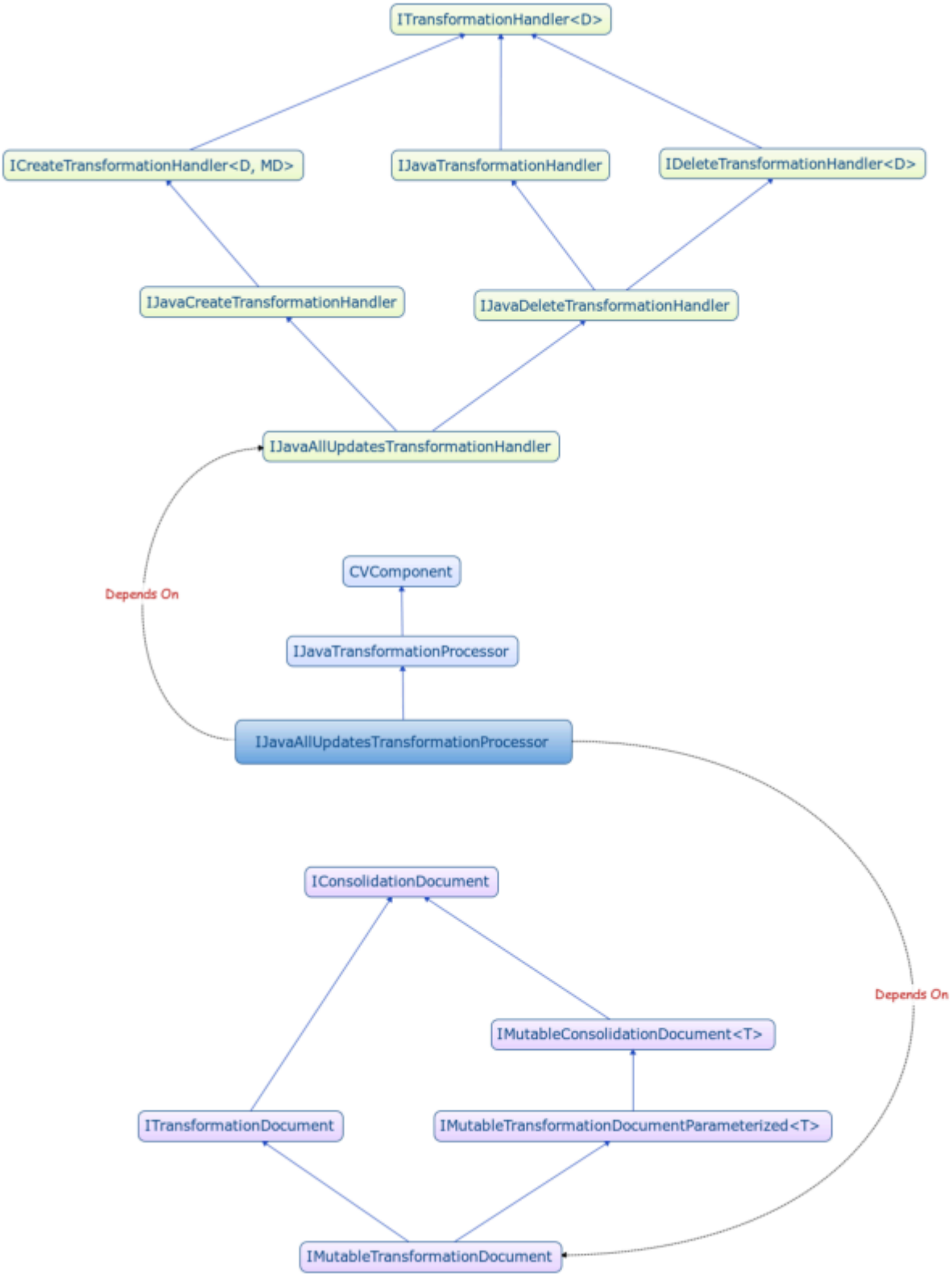
```
/**
 * Defines the interface for all the Java transformation processors that
 * need to perform operations in a non-delete context.
 */
public interface IJavaAllUpdatesTransformationProcessor extends IJavaTransformationPr
    /**
    * Performs the add or update operations of the clientâ€™s processor for the
    * the document transmitted, with the help of the handler provided.
    * @param handler The transformation handler providing the allowed operations for
    * @param document The reference document.
    * @throws Exception Occurs for whatever reason in the client's implementation.
    * The exception will most likely be wrapped with contextual information before fu
    */
    public void process(final IJavaAllUpdatesTransformationHandler handler,
                        final IMutableTransformationDocument document) throws Excepti
}
```

The parent interface is defined as follows:

```
/**
 * Defines the common operations for all Java transformation processors.
 */
public interface IJavaTransformationProcessor extends CVComponent {
    /**
    * Returns a document type on which the processor will perform a transformation.
    * If one returns null or an empty string, then it will be applied on all source d
    * @return A valid document type or null or empty string.
```

```
    */
    public String getTransformationDocumentType();
}
```

The following picture shows the complete class hierarchy associated with the Java transformation processors in the create/update action context.



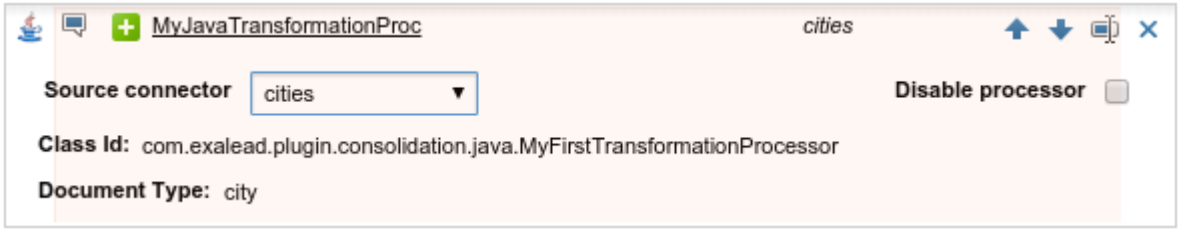
The handlers hierarchy (in green) defines the list of operations allowed for the processor and for the particular action context. The documents hierarchy (in purple) defines the document received with the transformations allowed on them.

If you implement the `IJavaAllUpdatesTransformationProcessor` interface as requested, you then have to implement the two following methods with a particular constructor receiving the component config.

Java Example 1

```
@CVComponentDescription("My First Transformation Processor Component")
@CVPluginVersion("1.0")
@CVComponentConfigClass(configClass=MyComponentConfig.class)
public class MyFirstTransformationProcessor implements IJavaAllUpdatesTransformationP
    public MyFirstTransformationProcessor(final MyComponentConfig config) {
    }
    @Override
    public String getTransformationDocumentType() {
        return "city";
    }
    @Override
    public void process(final IJavaAllUpdatesTransformationHandler handler,
                        final IMutableTransformationDocument document) throws Except
        // Do nothing, that is transmit all "city" documents to the next processor
        // or to the Consolidation Store.
        logger.info("Processing " + document.getURI());
    }
        private final Logger logger = Logger.getLogger("app-name.conso-server.transfo
}
```

Once the code above in packaged and deployed on the Exalead CloudView instance, you can define its associated source as shown below.



In this simple example:

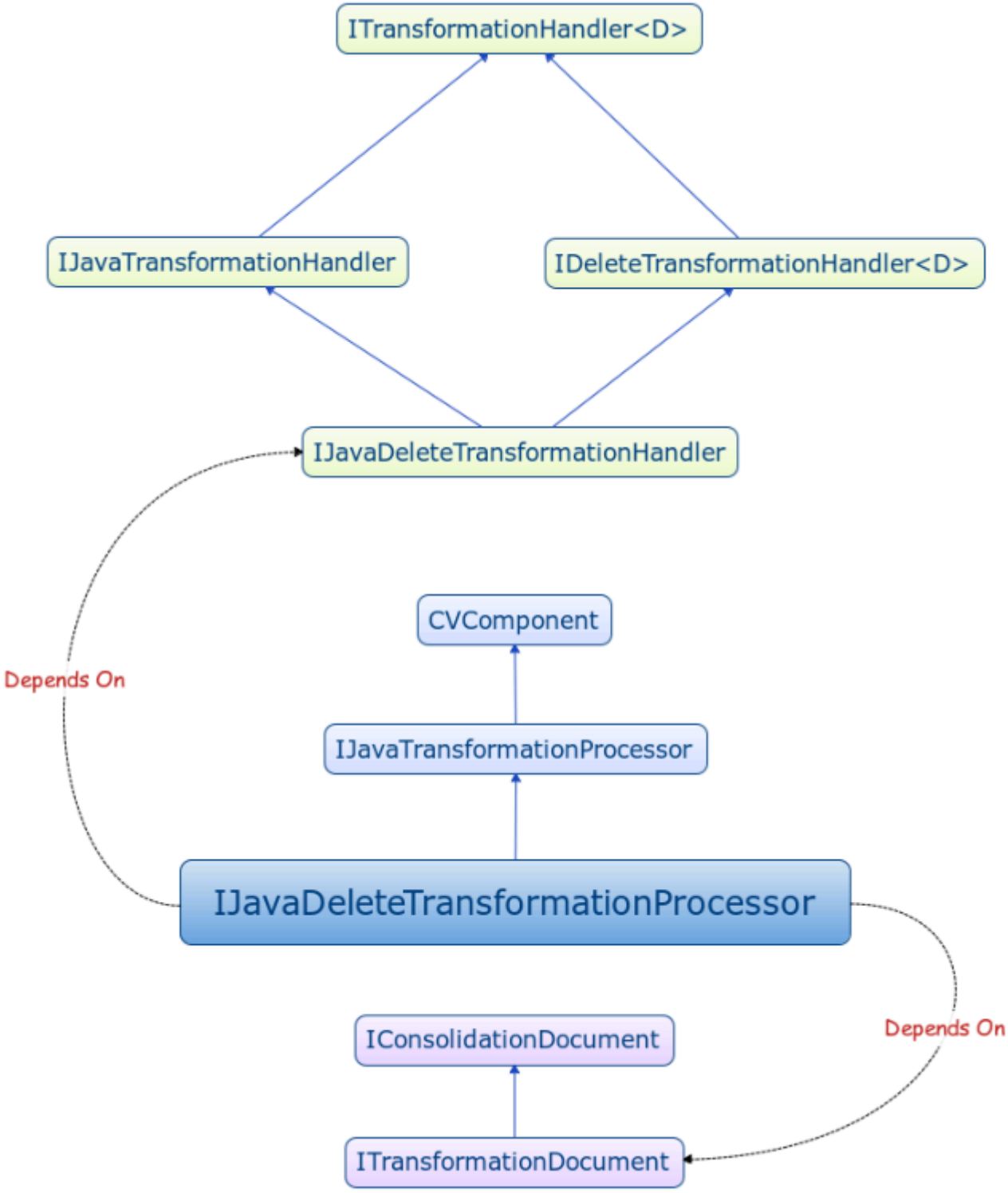
- We defined a constructor with the component config instance that you can use to customize the processor using end-user properties. Here we do not store the instance because we have no use for it. In general, we would save the instance in the processor class and use it in the process method to read specific properties. A component configuration is always required for the definition of each of your Java processors.

- The processor treats the documents coming from the `cities` source connector.
- The processor also treats, from such source, documents with the `city` type only. The rules of selection are detailed in Processor Type Inheritance and Runtime Selection.
- The `process` method contains the processor implementation. Here it contains no action (except from the logging), so all `city` documents from the `cities` source connector is automatically transmitted to the next available transformation processor, or to the Consolidation Store.

You can reduce the config class to the following implementation:

```
public final class MyComponentConfig implements CVComponentConfig {
  // No property defined
}
```

Similarly, in the delete action context you need to implement the `IJavaDeleteTransformationProcessor` interface. The methods to implement are mostly the same, except from the `process` method, which has a different signature, to emphasize that in such context, allowed operations are different from in the other one. Here is the class hierarchy defined in a delete action context:



Transformation Operations

This section lists the available transformation operations.

ITransformationHandler

The base interface of the transformation handler provides the two following methods, which control how documents are transformed within the processing pipeline.

Method	Description
`discard()`	Discards the current processor document, that is to say, prevents it from going to the next processor or next stage.
`yield(doc)`	Yields the newly created document to the next processor or to the Consolidation Store. Use this call for documents created in a transformation processor with the `IJavaCreateTransformationHandler` methods.

ICreateTransformationHandler

The interface to add new documents to the Consolidation Store provides three different create methods.

Tip:

When child URI is forged using the method:

IMutableTransformationDocument childDoc (IJavaAllUpdatesTransformationHandler) handler.createCh
childDoc.getUri()

The created URI is "document.URI" + childSeparator + "sub-URI" but as the childSeparator is a private string that is not visible in generated URIs, it is impossible to reforge this URI later without the same method.

Recommendation: To handle partial update use cases, create links from child to parent from the child only.

Instead of:

document.addArcTo("hasTextualElement", child.getUri());

Prefer:

child.addArcFrom("hasTextualElement", document.getUri());

Method	Description
`createDocument(uri,type,parentTypes)`	Creates a transformation document with the required given properties and with automatic memory management. In other words, if no edges point on it at the end of the transformation phase, the document is deleted by the Consolidation Server automatically.

Method	Description
`createChildDocument(parentDoc,subURI`	Creates a transformation document from a parent one with the given properties.
`createUnmanagedDocument(uri,type,par`	Creates a transformation document with the given properties without automatic memory management. This is the opposite behavior of the `createDocument` method in terms of memory management.
`getDocumentChildrenPath(String parentURI, String childURI)`	This method is useful to create a child URI when you do not have access to the child himself. Never forge a URI by hand.

`IDeleteTransformationHandler`

The interface to send delete orders to the Consolidation Store.

Method	Description
`deleteDocument()`	Sends a recursive delete order for the current document.
`deleteDocument(uri)`	Sends a recursive delete order for the document with the given URI prefix.
`deleteDocument(uri,boolean)`	Sends a delete order for the specified document URI, recursively or not. If the boolean flag is `true`, then all URIs with a prefix matching the given URI are also deleted.
`deleteDocument(doc)`	Sends a recursive delete order for the specified document and possibly all documents with a prefix matching the document URI.
`deleteDocument(doc,boolean)`	Sends a delete order for the specified document, recursively or not. If the boolean flag is `true`, then all URIs with a prefix matching the document URI are also deleted.
`deleteDocumentChildren(doc,path`	Sends a delete order for all document children matching the given path. The document itself is not deleted.

Method	Description
`deleteDocumentChildren(doc)`	Sends a delete order for all document children. The document itself is not deleted.
`deleteDocumentChildren(parentUR`	Sends a deletion order for all document children matching the path of the given parent URI. The document itself is not deleted.
`deleteDocumentChildren(parentUR`	Sends a deletion order for all document children with the given parent URI prefix. The document itself is not deleted.
`deleteDocumentRootPath(rootURI)`	Sends a deletion order for all documents matching the root URI prefix.

`IMutableTransformationDocument`

The following interface provides the operations specific to a Transformation document.

Method	Description
`addArcFrom(arcType, fromDoc)`	Registers an arc addition from the specified document to the current one.
`addArcFrom(arcType, fromDocURI)`	Registers an arc addition from the document specified by the URI to the current one.
`addArcTo(arcType, toDoc)`	Registers an arc addition from the current document to the specified document.
`addArcTo(arcType, toDocURI)`	Registers an arc addition from the current document to the document specified by the URI.
`removeAllPredecessorArcs`	Registers for deletion all adjacent arcs heading to the current one.
`removeAllSuccessorArcs()`	Registers for deletion all adjacent arcs starting from the current one.
`removeArcFrom(arcType, fromDoc)`	Registers for deletion the arc starting from the specified document to the current one, with the given type.
`removeArcFrom(arcType, fromDocURI)`	Registers for deletion the arc starting from the specified document to the current one, with the given type.

Method	Description
`removeArcTo(arcType, toDoc)`	Registers for deletion the arc starting from the current document to the specified document, with the given type.
`removeArcTo(arcType, toDocURI)`	Registers for deletion the arc starting from the document specified by the URI to the current one, with the give type.
`setType(documentType, parentTypes)`	Defines the document type, as well as its possible parents as defined in `getTypeInheritance()`.

`IConsolidationDocument`

The following interface gives access to the default data encapsulated within a consolidation document, either for transformation or aggregation.

Method	Description
`isOfType(type)`	Indicates if the type transmitted is among the list of the current document types.
`getAllDirectives()`	Returns all the directives defined in this document.
`getAllMetas()`	Returns all the metas defined in this document.
`getAllParts()`	Returns all the parts defined in this document.
`getDirectiveNames()`	Returns all the document directive names.
`getDirective(name)`	Returns the first directive value for the given name.
`getDirectives(name)`	Returns all the directives for the given name.
`getMetaNames()`	Returns all the meta names.
`getMeta(name)`	Returns the first meta value for the given name.
`getMetas(name)`	Returns all the meta values for the given name.
`getOriginalSources()`	Returns the list of original sources for the given document.
`getPartNames()`	Returns all the document part names.
`getPart(name)`	Returns the first document part for the given name.
`getParts(name)`	Returns the list of document parts for the given name.
`getSource()`	Returns the document original source that produced it.
`getType()`	Returns the document representative type.

Method	Description
`getTypeInheritance()`	Returns the type inheritance for the document. The first one in the list is a descendant of the second one, the second one of the third one, and so on. So types are ordered from the most specific to the most generic.
`getUri()`	Returns the document unique identifier.
`hasDirective(name)`	Indicates if the directive name has an associated value within the document.
`hasMeta(name)`	Indicates if the meta name has an associated value within the document.
`hasPart(name)`	Indicates if the part name has an associated value within the document.

`IMutableConsolidationDocument`

This interface enriches the operations available within `IConsolidationDocument` with a list of operations allowing the modifications of internal data.

Method	Description
`deleteDirective(name)`	Deletes all the directive values associated to the specified directive name.
`deleteDirectives(name, values)`	Deletes only the given values for the specified directive name.
`deleteMeta(name)`	Deletes all the meta values associated to the specified meta name.
`deleteMetas(name, values)`	Deletes only the given meta values from the specified meta name.
`deleteParts(name)`	Deletes the document parts related to the specified part name.
`deleteParts(name, documentParts)`	Deletes all the part directive values for the specified part name.
`setDirective(name, value)`	Assigns the given value to the specified directive name.
`setAllDirectives(directi`	Assigns all the directive name/values associated to the current document.

Method	Description
`setMeta(name, value)`	Assigns the given meta value to the specified meta name.
`setMeta(name, values)`	Assigns the given meta values to the specified meta name.
`setAllMetas(metas)`	Assigns all the meta name/values associated to the current document.
`setPart(name, docPart)`	Assigns the given document part to the specified part name.
`setParts(name, docParts)`	Assigns the given document parts to the specified part name.
`setAllParts(parts)`	Assigns all the parts associated to the current document.
`withDirective(name, value)`	Adds the value of a specific directive to the possible list of predefined directive values. If none is defined, a new list is created.
`withDirectives(name, values)`	Adds the values of a specific directive to the possible list of predefined directive values. If none is defined, a new list is created.
`withDirectives(directive`	Adds the list of directive key-values to the possible list of predefined directive values.
`withMeta(name, value)`	Adds the value of a specific meta to the possible list of predefined meta values. If none is defined, a new list is created.
`withMeta(name, values)`	Adds the values of a specific meta to the possible list of predefined meta values. If none is defined, a new list is created.
`withMetas(metas)`	Adds the list of meta key-values to the possible list of predefined meta values.
`withPart(name, docPart)`	Adds the document part to the list of existing predefined parts. If none is defined, a new list is created.
`withPart(name, docParts)`	Adds the sequence of document parts to the list of existing predefined parts. If none is defined, a new list is created.
`withParts(allParts)`	Adds the list of parts associated to the current document.

Define Java Aggregation Processors

In the Transformation phase, you have possibly filtered, modified, linked, and pushed documents into the Consolidation Store. In the Aggregation phase, you are then ready to aggregate or enrich

them together for the Exalead CloudView Index. You can also decide to notify the Indexer to delete some documents generated during the Aggregation.

You can define aggregation using default processors made for generic operations, or through custom java code if your needs are more specific.

Use Default Aggregation Processors

1. Under **Aggregation processors**, click **Add processor**.
2. In the **Add processor** dialog box, select **Java**, give a **name** to the processor, and then choose one of the following default processors.

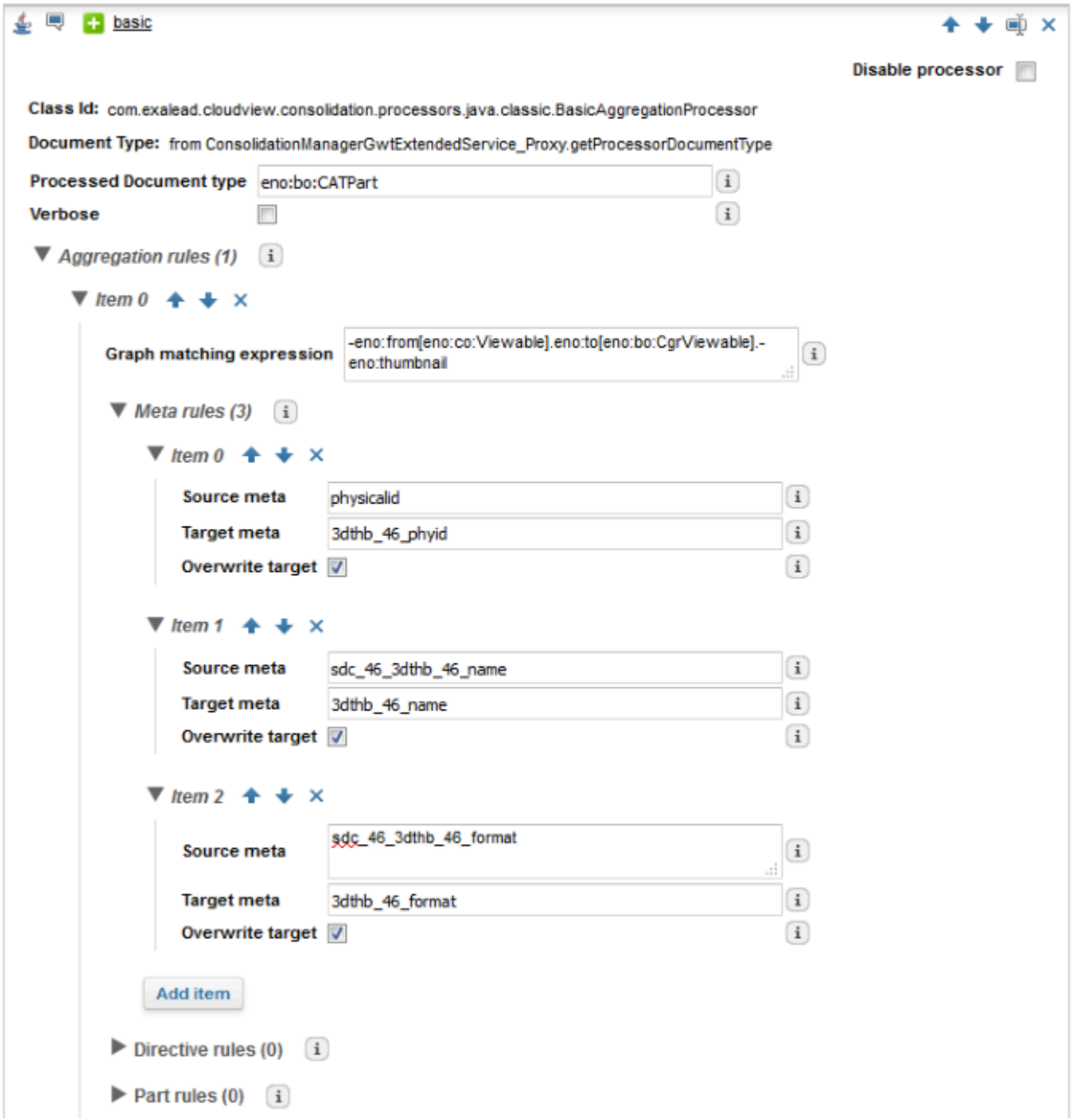
Aggregation Processor	Description
Basic Aggregation Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.java.classi BasicAggregationProcessor` Add/set metas, directives, or parts from documents at the end of paths, returned by the given graph matching expression. See the example below this table.
Classification Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.java.classi ClassficationAggregationProcessor` Generates classification metadata representing path nodes ('node1_id/node2_id/node3_id...')
Discard Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.java.classi DiscardAggregationProcessor` Discards documents matching the given document types. For a use case example, see UC-8: Consolidating Data from Storage Service.
Set Directive Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.java.classi SetDirectiveAggregationProcessor` Sets the given directive on the processed document.
Set Meta Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.java.classi SetMetaAggregationProcessor`

Aggregation Processor	Description
	Sets the given meta on the processed document.
Storage Service Key Flattener Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.java.classi StorageServiceKeyFlattenerAggregationProcessor` Sets metas on a document coming from the Storage Service. For a use case example, see UC-8: Consolidating Data from Storage Service.
Interconnector Aggregator Processor	**Class Id**: `com.exalead.cloudview.consolidation.processors.java. InterconnectorAggregatorProcessor` Aggregates a parent document with its child document, given a graph path from parent to child. For a use case example, see in the Exalead CloudView Connectors Guide.

For example, with the **Basic Aggregation Processor**, you can replace the following Groovy code, which rewrites metas at the end of paths:

```
process("eno:bo:CATPart") {
 for (node in match(it, "-eno:from[eno:co:Viewable].eno:to[eno:bo:CgrViewable].-en
  it.metas."3dthb_46_phyid" += node.metas."physicalid";
  it.metas."3dthb_46_name" += node.metas."sdc_46_3dthb_46_name";
  it.metas."3dthb_46_format" += node.metas."sdc_46_3dthb_46_format";
 }
}
```

By the following configuration:



3. Click **Apply**.

Create Custom Aggregation Processors

In Java, to define an Aggregation processor in the create/update action context, you need to implement the `IJavaAllUpdatesAggregationProcessor` interface. Here is the actual interface definition:

```
/**
 *  Defines the interface for all Java aggregation processors that need to perform dc
 *  in a non-delete context.
 */
```

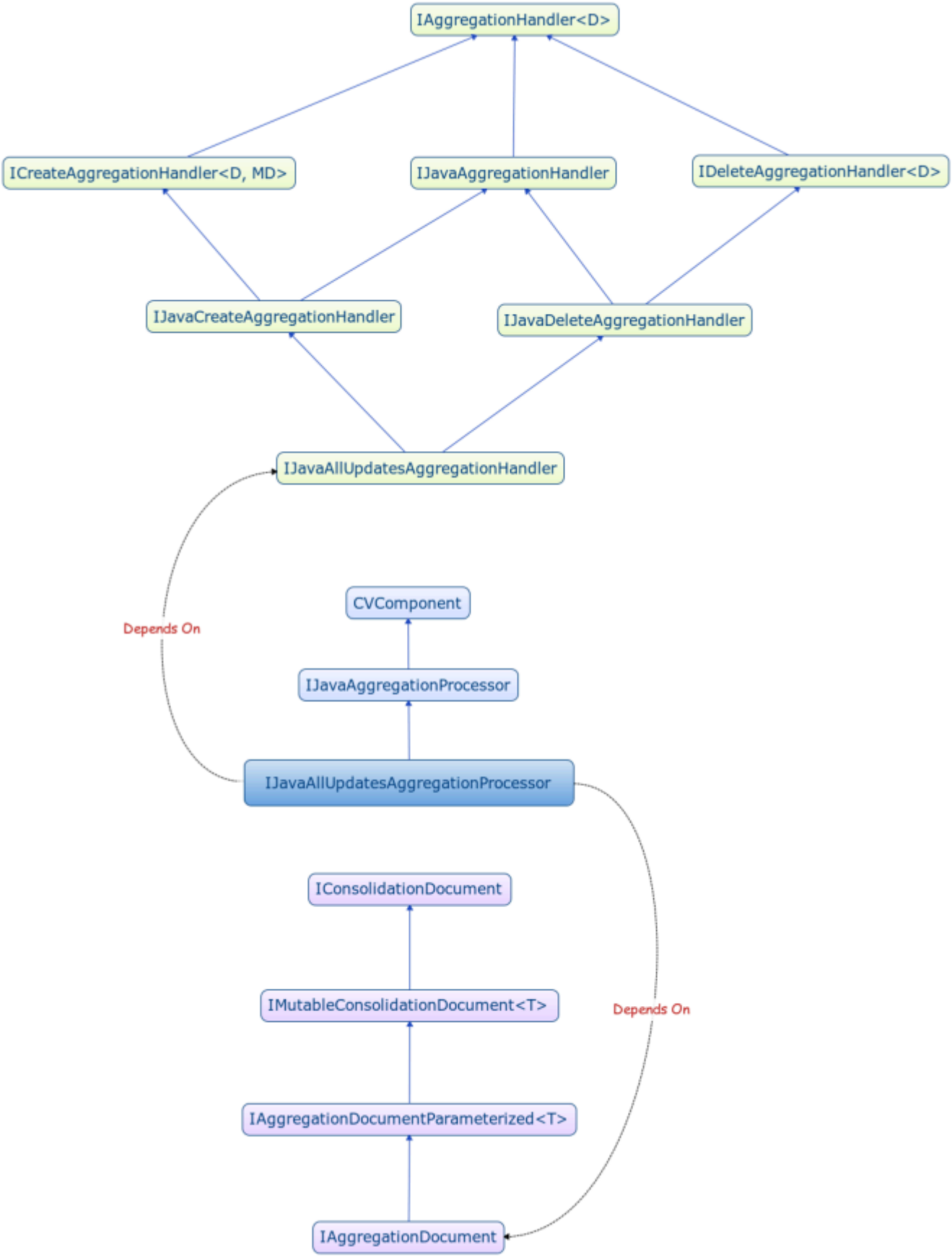
```
 public interface IJavaAllUpdatesAggregationProcessor extends IJavaAggregationProcess
   /**
   * Performs the aggregation operations of the client's processor for the document t
   * with the help of the handler provided.
   *
   * @param handler The aggregation handler with the allowed operations for the proce
   * @param document The reference document.
   * @throws Exception Occurs for whatever reason in the client's implementation.
   * The exception will most likely be wrapped with contextual information before fur
   */
   public void process(final IJavaAllUpdatesAggregationHandler handler, final IAggreg
 throws Exception;
}
```

The parent interface is defined as follows:

```
/**
 * Defines the interface for all Java aggregation processors that need to
 * perform document operations in a non-delete context.
 */
public interface IJavaAllUpdatesAggregationProcessor extends IJavaAggregationProcesso
   /**
   * Performs the aggregation operations of the client's processor for the document t
   * with the help of the handler provided.
   *
   * @param handler The aggregation handler with the allowed operations for the proce
   * @param document The reference document.
   * @throws Exception Occurs for whatever reason in the client's implementation.
   * The exception will most likely be wrapped with contextual information before fur
   */
    public void process(final IJavaAllUpdatesAggregationHandler handler, final IAggre
 throws Exception;
}
```

Note: This time, there is no need to specify the source connector within the Administration Console, since all documents are loaded from the Consolidation Store.

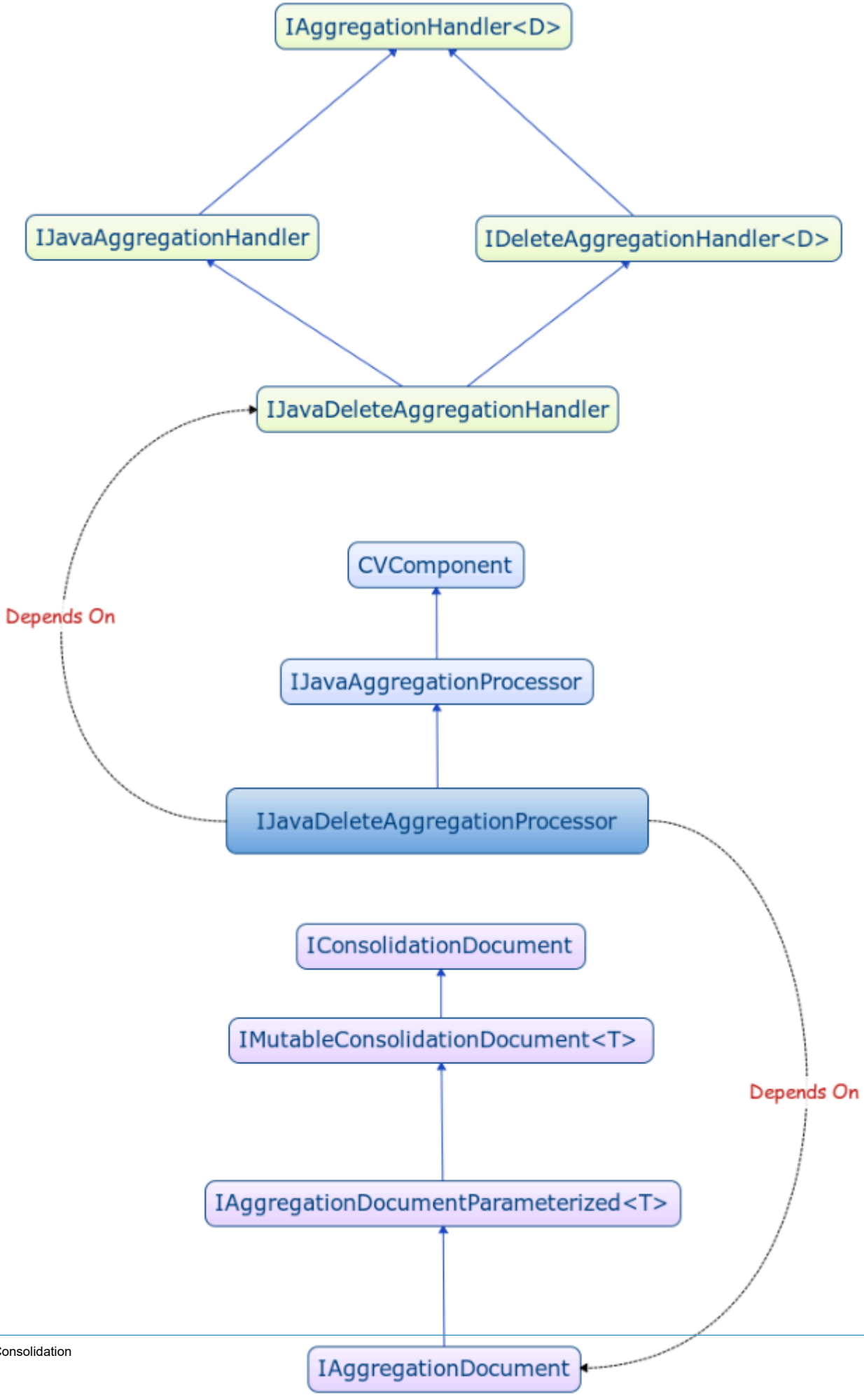
The class hierarchy is the following:



For the delete action context, you have to implement the `IJavaDeleteAggregationProcessor` interface as follows:

```
/**
 * Defines the interface for all Java aggregation processors that need to perform del
 * processing on documents.
 */
public interface IJavaDeleteAggregationProcessor extends IJavaAggregationProcessor {
   /**
   * Performs the aggregation operations of the client's processor for the document t
   * with the help of the handler provided.
   *
   * @param handler The aggregation handler with the allowed operations for the proce
   * @param document The reference document.
   * @throws Exception Occurs for whatever reason in the client's implementation.
   * The exception will most likely be wrapped with contextual information before fur
   */
   public void process(final IJavaDeleteAggregationHandler handler, final IAggregatio
 throws Exception;}
```

And finally, the class hierarchy is:


IAggregationHandler<D>
IJavaAggregationHandler
IDeleteAggregationHandler<D>
IJavaDeleteAggregationHandler
Depends On
CVComponent
IJavaAggregationProcessor
IJavaDeleteAggregationProcessor
IConsolidationDocument
IMutableConsolidationDocument<T>
Depends On
IAggregationDocumentParameterized<T>
IAggregationDocument

Aggregation Operations

This section lists the available aggregation operations.

`IAggregationHandler`

The base interface of the aggregation handler provides the next fundamental methods.

Method	Description
`discard()`	Discards the current processor document, that is to say, prevent it from going to the next processor or next stage.
`getReason()`	Returns a string representing the reason why the document is pushed to aggregation. It can have one of the following values: `ADDED`, `DELETED`, `IMPACTED`.
`match(doc,graphMatchingEx`	Finds the list of paths in the graph that start from the specified `IAggregationDocument` and that satisfy the `graphMatchingExpression`. Returns them as a list of documents.
`matchPathEnd(doc,graphMat`	Finds the documents at the end of each path in the graph, that starts from the specified `IAggregationDocument` and that satisfy the `graphMatchingExpression`. Returns them as a list of documents. This is useful when you do not want to overload the Consolidation Server with a lot of useless intermediary documents, found on the path between the starting document and the document level you chose as path end. In other words, instead of considering all the vertices on a given path, it only considers the one at the end.
`matchPathEnd(doc,graphMat`	Finds the documents at the end of each path in the graph, that starts from the specified `IAggregationDocument`, satisfy the `graphMatchingExpression`. Returns them as a list of documents. The goal of this method is to avoid impacting elements if the meta that changed is not used. Instead of considering all the vertices on a given path, it only considers the one at the end, only if the meta used has changed. This is triggered when the impact detection is launched during the incremental scan.

Method	Description
	Warning: This method does not work with Date metas.
`matchPathEnd(doc,graphMat testDirectives,testParts,`	Finds the documents at the end of each path in the graph, that starts from the specified `IAggregationDocument`, satisfy the `graphMatchingExpression`. Returns them as a list of documents. The goal of this method is to avoid impacting elements if the meta that changed is not used, and if directives and parts are the same. Instead of considering all the vertices on a given path, it only considers the one at the end, only if the meta used has changed, or if directives are different, or if parts are different. This is triggered when the impact detection is launched during the incremental scan. **Warning:** This method does not work with Date metas.
`yield(doc)`	Yields the newly created document to the forward rules without passing through the whole pipeline of aggregation processors. Use this call for documents created in an aggregation processor with the `IJavaCreateAggregationHandler` methods.
`yieldAndForward(doc)`	Yields the documents newly created in an aggregation processor to the next aggregation processor in the pipeline of aggregation processors. Use this call for documents created in an aggregation processor with the `createDocument` or the `createChildDocument` methods. This is to make sure that the document is forwarded to the next processor and not sent to the specified forward rules directly, unlike the `yield(doc)` method.

`IJavaAggregationHandler`

This interface extends the `IAggregationHandler` interface to provide a different approach for collecting graph matching results when using Java.

Method	Description
`match(doc,graphMatchingExpression,ma`	Finds the list of paths in the graph that start from the specified `IAggregationDocument` and that satisfy the `graphMatchingExpression`.

Method	Description
	Unlike the other `match` method, it provides the results using the `matchResultVisitor` instance with all unique documents matching the graph matching expression (independently of the paths reached).

`ICreateAggregationHandler`

The interface to add new documents to the forward rules provides two different `create` methods and a specific service to fetch document parts from a connector instance.

Method	Description
`createDocument(uri,type,parentTypes)`	Create an aggregation document with the given properties. Unlike `ICreateTransformationHandler.createDocumer` this document is not automatically deleted if there are no edges point on it at the end of the aggregation phase. It is pushed as is to the forward rules, and sent (or not) to an Indexing Server or another Consolidation Instance.
`createChildDocument(parentDoc,subURI`	Creates an aggregation document from a parent one with the given properties.
`isFetchOperation()`	When a Fetch Server performs a fetch operation request to the Consolidation Server, this handler (and in this case only) returns `true`. When this is the case, all the aggregation operations performed in the processor are directed in return to the Fetch Server. None of the documents aggregated proceed to the forward rules handler, and thus to the Indexing Server. The operations allowed in such event are the ones of a create/update context, and the `fetchParts` operation. In most cases, you do not have to deal with this kind of situation.

Method	Description
`fetchParts(document,connectorName, connectorDocumentURI)`	Fetches the parts corresponding to the `connectorDocumentURI` document from the connector specified by `connectorName` and appends them to the given document. This call makes sense only when the `isFetchOperation()` method returns true.

`IDeleteAggregationHandler`

The interface to send delete orders to the forward rules. Unlike `IDeleteTransformationHandler`, all methods are similar, apart from an extra parameter, which receives a possible list of document types, that is added to all signatures.

When you create new custom documents during the aggregation phase using the `create'*'` methods of the `ICreateAggregationHandler` interface in one processor, and later try to send a `delete` order for these documents in another processor, you no longer have access to any of the document metadata, especially the document types.

Such information is only known by the Indexing Server or by another Consolidation Server instance, depending on the routing strategy applied by the forward rules handler.

As a result, if you want to send a `delete` order to custom aggregated documents, you need to specify their types so that the forward rules handler can apply a dedicated routing strategy.

You do not need to specify the types for all documents present in the Consolidation Store that are processed during the aggregation phase (unlike the transformation phase). The Consolidation Server provides all required metadata to the forward rule handler so that it can operate accurately.

Method	Description
`deleteDocument()`	Sends a recursive deletion order for the document being aggregated, and all the other documents with a prefix matching the current document URI.
`deleteDocument(docTypes...)`	Sends a recursive deletion order for the document being aggregated, and all the other subdocuments with a prefix matching the current document URI. Moreover, to delete documents not recognized in the Consolidation Store and allow correct routing/filtering by the forward rules handler, a recursive deletion order is sent for the specified document types matching the current document URI.

Method	Description
`deleteDocument(uri,boolean)`	Sends a deletion order for the specified document URI, recursively or not. If the boolean flag is `true`, then all URIs with a prefix matching the given URI are also deleted.
`deleteDocument(uri, docTypes...)`	Sends a recursive deletion order for the document with the specified URI prefix.
`deleteDocument(uri,boolean,docT`	Sends a deletion order for the specified document URI, recursively or not. If the boolean flag is `true`, then all URIs with a prefix matching the given URI are also deleted. Moreover, to delete documents not recognized in the Consolidation Store and allow correct routing/filtering by the forward rules handler, a deletion order (recursive or not) is sent for the given document types matching the specified document URI.
`deleteDocument(doc)`	Sends a recursive deletion order for the specified aggregated document and possibly all documents with a prefix matching the document URI.
`deleteDocument(doc,docTypes...)`	Sends a recursive deletion order for the specified aggregated document and possibly all documents with a prefix matching the document URI. Moreover, a recursive deletion order with the given document is sent with the additional forward rule types provided, to delete documents not recognized in the Consolidation Store while allowing correct routing/filtering by the forward rules handler (if required). Moreover, to delete documents not recognized in the Consolidation Store and allow correct routing/filtering by the forward rules handler, a recursive deletion order is sent for the given document types matching the current document URI.
`deleteDocument(doc,boolean)`	Sends a deletion order for the given document, recursively or not. If the boolean flag is `true`, then all URIs with a prefix matching the document URI are also deleted.

Method	Description
`deleteDocument(doc,boolean,docT`	Sends a deletion order for the given document, recursive or not. If the boolean flag is true, then all URIs with a prefix matching the document URI are also deleted. Moreover, to delete documents not recognized in the Consolidation Store and allow correct routing/filtering by the forward rules handler, a deletion order (recursive or not) is sent for the specified document types matching the document URI.
`deleteDocumentChildren(doc,path`	Sends a deletion order for all document children matching the given path. The document itself is not deleted.
`deleteDocumentChildren(doc,path`	Sends a deletion order for all document children matching the given path. The document itself is not deleted. Moreover, to delete documents not recognized in the Consolidation Store and allow correct routing/filtering by the forward rules handler, a recursive children deletion order is sent for the specified document types matching the current document URI.
`deleteDocumentChildren(uri,path`	Sends a deletion order for all document children of the given URI matching the given path. The document itself is not deleted.
`deleteDocumentChildren(uri,path`	Sends a deletion order for all document children of the given URI matching the given path. The document itself is not deleted. Moreover, to delete documents not recognized in the Consolidation Store and allow correct routing/filtering by the forward rules handler, a recursive children deletion order is sent for the specified document types matching the specified document URI.
`deleteDocumentChildren(doc)`	Sends a deletion order for all document children. The document itself is not deleted.
`deleteDocumentChildren(doc,docT`	Sends a deletion order for all document children. The document itself is not deleted.

Method	Description
	Moreover, to delete documents not recognized in the Consolidation Store and allow correct routing/filtering by the forward rules handler, a recursive children deletion order is sent for the specified document types matching the current document URI.
`deleteDocumentChildren(uri)`	Sends a deletion order for all document children of the given URI. The document itself is not deleted.
`deleteDocumentChildren(uri,docT`	Sends a deletion order for all document children of the given URI. The document itself is not deleted. Moreover, to delete documents not recognized in the Consolidation Store and allow correct routing/filtering by the forward rules handler, a recursive children deletion order is sent for the specified document types matching the specified document URI.
`deleteDocumentRootPath(rootURI)`	Deletes all the documents matching the root URI prefix.
`deleteDocumentRootPath(rootURI,`	Deletes all the documents matching the root URI prefix, and with some forward rule types to allow correct routing/filtering by the forward rules handler.

`IConsolidationDocument`

The following interface gives access to the default data encapsulated within a consolidation document, either for transformation or aggregation.

Method	Description
`isOfType(type)`	Indicates if the type transmitted is among the list of the current document types.
`getAllDirectives()`	Returns all the directives defined in this document.
`getAllMetas()`	Returns all the metas defined in this document.
`getAllParts()`	Returns all the parts defined in this document.
`getDirectiveNames()`	Returns all the document directive names.
`getDirective(name)`	Returns the first directive value for the given name.
`getDirectives(name)`	Returns all the directives for the given name.

Method	Description
`getMetaNames()`	Returns all the meta names.
`getMeta(name)`	Returns the first meta value for the given name.
`getMetas(name)`	Returns all the meta values for the given name.
`getOriginalSources()`	Returns the list of original sources for the given document.
`getPartNames()`	Returns all the document part names.
`getPart(name)`	Returns the first document part for the given name.
`getParts(name)`	Returns the list of document parts for the given name.
`getSource()`	Returns the document original source that produced it.
`getType()`	Returns the document representative type.
`getTypeInheritance()`	Returns the type inheritance for the document. The first one in the list is a descendant of the second one, the second one of the third one, and so on. So types are ordered from the most specific to the most generic.
`getUri()`	Returns the document unique identifier.
`hasDirective(name)`	Indicates if the directive name has an associated value within the document.
`hasMeta(name)`	Indicates if the meta name has an associated value within the document.
`hasPart(name)`	Indicates if the part name has an associated value within the document.

`IMutableConsolidationDocument`

This interface enriches the operations available within `IConsolidationDocument` with a list of operations allowing the modifications of internal data.

Method	Description
`deleteDirective(name)`	Deletes all the directive values associated to the specified directive name.
`deleteDirectives(name, values)`	Deletes only the given values for the specified directive name.

Method	Description
`deleteMeta(name)`	Deletes all the meta values associated to the specified meta name.
`deleteMetas(name, values)`	Deletes only the given meta values from the specified meta name.
`deleteParts(name)`	Deletes the document parts related to the specified part name.
`deleteParts(name, documentParts)`	Deletes all the part directive values for the specified part name.
`setDirective(name, value)`	Assigns the given value to the specified directive name.
`setAllDirectives(directi`	Assigns all the directive name/values associated to the current document.
`setMeta(name, value)`	Assigns the given meta value to the specified meta name.
`setMeta(name, values)`	Assigns the given meta values to the specified meta name.
`setAllMetas(metas)`	Assigns all the meta name/values associated to the current document.
`setPart(name, docPart)`	Assigns the given document part to the specified part name.
`setParts(name, docParts)`	Assigns the given document parts to the specified part name.
`setAllParts(parts)`	Assigns all the parts associated to the current document.
`withDirective(name, value)`	Adds the value of a specific directive to the possible list of predefined directive values. If none is defined, a new list is created.
`withDirectives(name, values)`	Adds the values of a specific directive to the possible list of predefined directive values. If none is defined, a new list is created.
`withDirectives(directive`	Adds the list of directive key-values to the possible list of predefined directive values.
`withMeta(name, value)`	Adds the value of a specific meta to the possible list of predefined meta values. If none is defined, a new list is created.
`withMeta(name, values)`	Adds the values of a specific meta to the possible list of predefined meta values. If none is defined, a new list is created.

Method	Description
`withMetas(metas)`	Adds the list of meta key-values to the possible list of predefined meta values.
`withPart(name, docPart)`	Adds the document part to the list of existing predefined parts. If none is defined, a new list is created.
`withPart(name, docParts)`	Adds the sequence of document parts to the list of existing predefined parts. If none is defined, a new list is created.
`withParts(allParts)`	Adds the list of parts associated to the current document.

Company Hierarchy Example

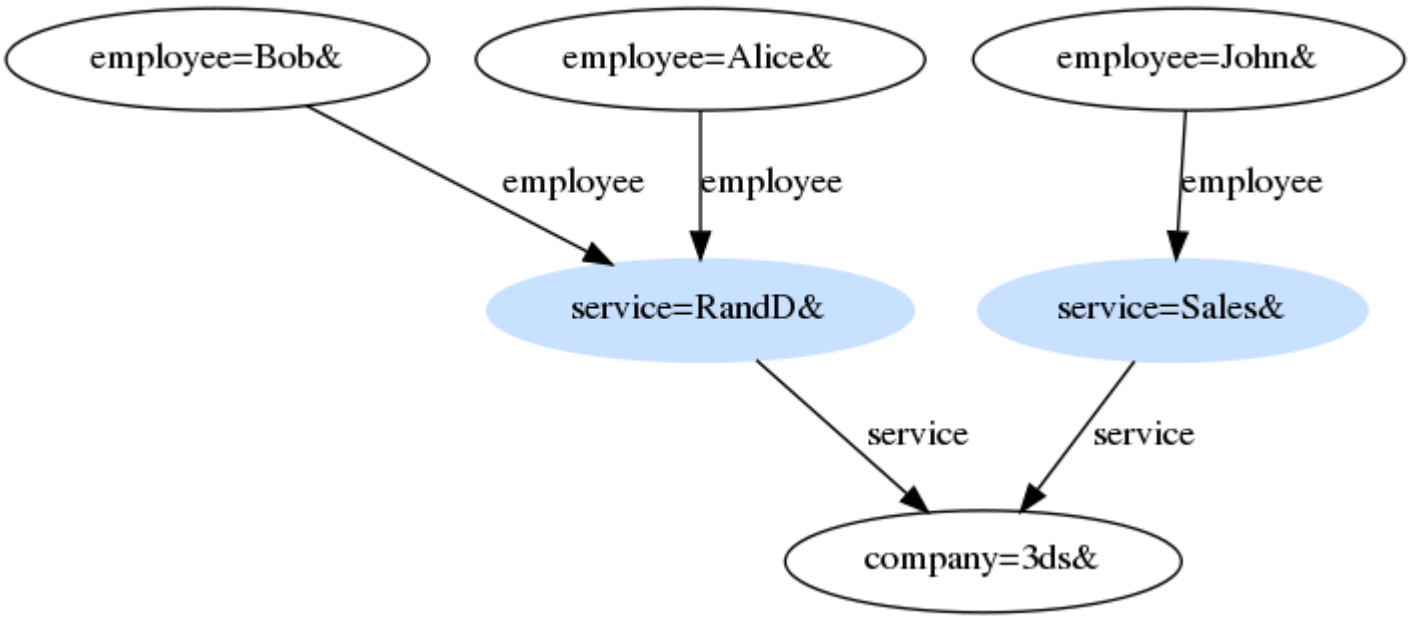
In the following use case, we have people and companies, and we want to enrich the company with a meta indicating the number of employees present at any time.

We have two types of documents:

- `company`: Contains the company name in its URI. It holds possibly many other metas that identify the company.
- `employee`: Contains the employee's name in its URI. It holds possibly many other metas that identify the employee, but contains at least two metas:
  - `company_name` contains the company's name in which the employee is working.
  - `service_name` contains the service in which the employee is working (sales, R&D, marketing, etc.).

Connect Employees to Services and Services to Companies

We want to connect each employee to the service, and the service to the appropriate company with the following data model.



The code for such a transformation may look like the following:

Example 1. Employee's Transformation Processor

```
@CVComponentConfigClass(configClass=CVComponentConfigNone.class)
public class EmployeeTransformationProcessor implements IJavaAllUpdatesTransformation
    public EmployeeTransformationProcessor(final CVComponentConfigNone config) {
    }
    @Override
    public String getTransformationDocumentType() {
        return "employee";
    }
    @Override
    public void process(final IJavaAllUpdatesTransformationHandler handler,
      final IMutableTransformationDocument document) throws Exception {
        final String companyName = document.getMeta("company_name");
        final String serviceName = document.getMeta("service_name");
        if ((companyName != null) && (! companyName.isEmpty()) && (serviceName != nul
(! serviceName.isEmpty())) {
          final ITransformationDocument serviceDoc = addService(handler, document, se
          document.addArcTo("employee", serviceDoc.getUri());
        }
    }
    private ITransformationDocument addService(final IJavaAllUpdatesTransformationHan
      final IMutableTransformationDocument document, final String serviceName, final
        final IMutableTransformationDocument newDoc = handler.createDocument("service
 "service");
       newDoc.addArcTo("service", "company=" + companyName + "&");
       handler.yield(newDoc);
          // Note that the yield here is required because it is a document created
          // during the Transformation phase
          return newDoc;
```

```
    }
}
```

The drawback of this implementation is that it pushes the arcs that link services to the company several times. In the end, since these arcs have the same type, only the relevant ones persist (with no redundancies in the store).

However, it is always better to minimize the number of redundant operations. If we had the required information, we could create the different services that the company has, with unique URIs, and then at the employee level, we would link employees to services.

A possible implementation could be:

```
...
@Override
    public void process(final IJavaAllUpdatesTransformationHandler handler,
       final IMutableTransformationDocument document) throws Exception {
        final String serviceName = document.getMeta("service_name");
        if ((companyName != null) && (! companyName.isEmpty()) && (serviceName != nul
(! serviceName.isEmpty())) {
            document.addArcTo("employee", "service=" + serviceName + "_" + companyNam
        }
    }
...
```

Despite its imperfection, let us stick to this first implementation from now on. For more information about the method used in this sample, see IMutableTransformationDocument .

Keep the Business Logic Within the Connector

Sometimes, you might want to keep the business logic within your connector, even if it is not recommended. You can do that using the `com.exalead.cloudview.consolidationapi.PushAPITransformationHelpers`. The sample below shows how to embed the graph modeling done by the `EmployeeTransformationProcessor` directly within your connector.

```
final PushAPI employeePushAPI = CloudviewAPIClientsFactory.newInstance(GATEWAY_URL).n
(PUSH_API_SERVER, CONNECTOR_NAME);
final List<Document> employees = new ArrayList<Document>();
Document employee = new Document("employee=Alice&");
employee.addMeta("company_name", "3ds");
employee.addMeta("service_name", "RandD");
employees.add(employee);
employee = new Document("employee=Bob&");
employee.addMeta("company_name", "3ds");
employee.addMeta("service_name", "RandD");
employees.add(employee);
employee = new Document("employee=John&");
employee.addMeta("company_name", "3ds");
```

```
employee.addMeta("service_name", "Sales");
employees.add(employee);
final Iterator<Document> employeesIt = employees.iterator();
while (employeesIt.hasNext()) {employee = employeesIt.next();
final String serviceURI = getServiceURI(employee.getMetaContainer().getMeta("service_
// Create service managed document
PushAPITransformationHelpers.createDocument(employee, serviceURI, "service");
// Add arc from employee to service
PushAPITransformationHelpers.addArcTo(employee, "employee", serviceURI);
// Add arc from service to company
final String companyURI = getCompanyURI(employee.getMetaContainer().getMeta("company_
PushAPITransformationHelpers.addArcTo(employee, "service", serviceURI, companyURI);
employeePushAPI.addDocument(employee);
}
```

Count the Number of Employees and Push Updated Documents

Now, for each `company` document, we want to add a `nb_employees` meta that counts the total number of employees, and push updated document to the Indexing Server. You can perform this kind of task during the aggregation phase.

A possible implementation could be:

```
@CVComponentConfigClass(configClass=CVComponentConfigNone.class)
public final class CompanyAggregationProcessor implements IJavaAllUpdatesAggregationP
    public CompanyAggregationProcessor(final CVComponentConfigNone config) {
    }
    @Override
    public String getAggregationDocumentType() {
       return "company";
    }

    @Override
    public void process(final IJavaAllUpdatesAggregationHandler handler, final IAggre
 throws Exception {
        int nbEmployees = 0;
        for (final IAggregationDocument serviceDoc : GraphMatchHelpers.getPathsEnd(ha
 "-service"))) {
            nbEmployees += handler.match(serviceDoc, "-employee").size();
        }
        document.withMeta("nb_employees", String.valueOf(nbEmployees));
    }
}
```

We first retrieve all the services that belong to a given company with the following call:

```
handler.match(document, "-service")
```

This returns all the paths starting from the `company` document that follow the `service` edge in reverse order.

We know by design, and also from the match query, that such paths contain only one document, the neighbors of the company document. So `GraphMatchHelpers.getPathsEnd` is responsible for accessing it. The Java code for such helper method must be equal (or equivalent) to:

```
public static <T> List<T> getPathsEnd(final List<List<T>> paths) {
    return Lists.transform(paths, new Function<List<T>, T>() {
       @Override
       public T apply(final List<T> path) {
           return Iterables.getLast(path);
        }
    });
}
```

Then for each service document:

```
handler.match(serviceDoc, "-employee").size()
```

Returns all the paths leading to a unique employee in the service. We need to get the number of paths to get the number of employees in the service. The company document is then enriched with the `nb_employee` meta with the variable that allowed us to sum up all the different paths that were found.

A better and simpler implementation is:

Example 2. Company's Aggregation Processor

```
@CVComponentConfigClass(configClass=CVComponentConfigNone.class)
public final class CompanyAggregationProcessor implements IJavaAllUpdatesAggregationP
    public CompanyAggregationProcessor(final CVComponentConfigNone config) {
    }
    @Override
    public String getAggregationDocumentType() {
        return "company";
    }
    @Override
    public void process(final IJavaAllUpdatesAggregationHandler handler, final IAggre
 throws Exception {
        int nbEmployees = handler.match(document, "-service.-employee").size()
        document.withMeta("nb_employees", String.valueOf(nbEmployees));
    }
}
```

Reach Employee Documents from the Company Document

The graph matching expression language allows us to specify an arbitrary long path, with various quantifiers (see Appendix - Matching Expressions Grammar). We can therefore reach the employee documents from the company document directly, with the expression:

```
handler.match(document, "-service.-employee")
```

Push the Number of Employees Present in Each Service

We could also want, as a refinement, to push the number of employees present in each service. Writing the following code would then be enough:

```
@CVComponentConfigClass(configClass=CVComponentConfigNone.class)
public final class ServiceAggregationProcessor implements IJavaAllUpdatesAggregationP
        public ServiceAggregationProcessor(final CVComponentConfigNone config) {
    }
    @Override
    public String getAggregationDocumentType() {
        return "service";
    }
    @Override
    public void process(final IJavaAllUpdatesAggregationHandler handler, final IAggre
 throws Exception {
        document.withMeta("nb_employees", String.valueOf(handler.match(document, "-em
    }
}
```

Important: If you have written the above processor first, avoid writing the following processor afterward to aggregate the number of employees for the company.

```
@CVComponentConfigClass(configClass=CVComponentConfigNone.class)
public final class CompanyAggregationProcessor implements IJavaAllUpdatesAggregationP
        public CompanyAggregationProcessor(final CVComponentConfigNone config) {
    }
    @Override
    public String getAggregationDocumentType() {
        return "company";
    }
    @Override
    public void process(final IJavaAllUpdatesAggregationHandler handler, final IAggre
 throws Exception {
        int nbEmployees = 0;
        for (final IAggregationDocument serviceDoc : GraphMatchHelpers.getPathsEnd(ha
 "-service"))) {
            final String nbServiceEmployees = serviceDoc.getMeta("nb_employees");
            if ((nbServiceEmployees != null) && (! nbServiceEmployees.isEmpty()) {
                nbEmployees += Integer.valueOf(nbServiceEmployees);
```

```
            }
        }
        document.withMeta("nb_employees", String.valueOf(nbEmployees));
    }
}
```

The code above collects all `service` documents, and for each of them, sums up the values coming from its `nb_employees` meta.

This code works because even if we sum up the services meta values while more documents are still arriving, the impact detection ensures that the processor for these specific company documents is re-evaluated.

What may prevent this code from working properly is that the data visible during the Aggregation phase comes from the data pushed to the Consolidation store only, and nothing more! In other words, all document modifications and newly created custom documents that occur during the Aggregation phase are not visible to one another. So the company meta does not have any visibility on the new `nb_employees` meta created during the aggregation phase by the service processor.

Manage Documents Explicitly

You can set aside the subtleties about the lifecycle management of documents created during the transformation and aggregation phases if rather than creating custom URIs (that is, documents with URIs that do no share anything in common with the document that created them) you create child documents.

Creating child documents from a given document managed by a connector, ensures that when the connector pushes the document deletion, the Consolidation Server registers for deletion all child documents automatically. This behavior is true for both the transformation and aggregation phases.

Note: In such case, the deletion occurs whether the document is attached to another one or not. The deletion criteria is URI-based.

As as result, this type of document creation is the preferred one if you do not want to bother with the lifecycle management of these objects. If you choose this method, it is unlikely that you ever need to write a processor in delete action context.

In the Transformation Phase

In the Company Hierarchy Example code, we pushed the creation and updates of employees to the Consolidation Store. With them, we have possibly created manually new documents representing the service they belong to.

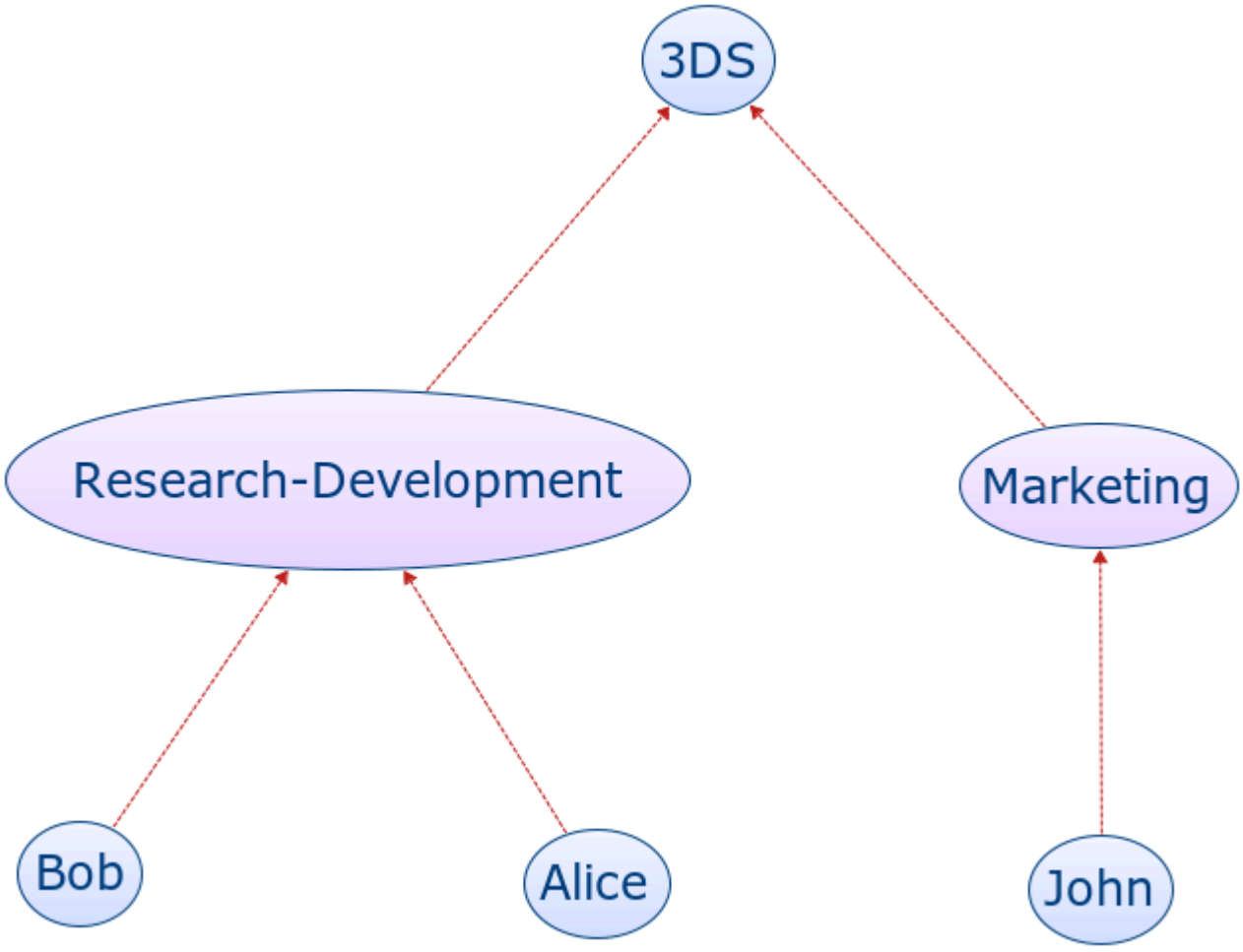
Note: Connectors do not manage `service` documents. In our use case, the CDIH would refuse `delete` orders for a service document, as it is created afterward.

Deleting the documents created within the transformation phase is within the hands of the developer writing the processor logic. If you create managed documents, like we did with the call to `createDocument`, then the document is removed from the Store automatically once no other documents are attached to it. If the connectors send `delete` orders for the 3ds company as well as all its employees, then service documents become orphaned, and garbage collected automatically.

What would happen if you sent an order to delete all the employees of a given service? In such case, it would ultimately delete all employees from the Store, and with them all the edges that were pointing to the associated service. However as services would still point to the company, these documents would stay in the Store. Remember that managed documents are garbage collected only when no edges are attached to them at the end of the transformation phase.

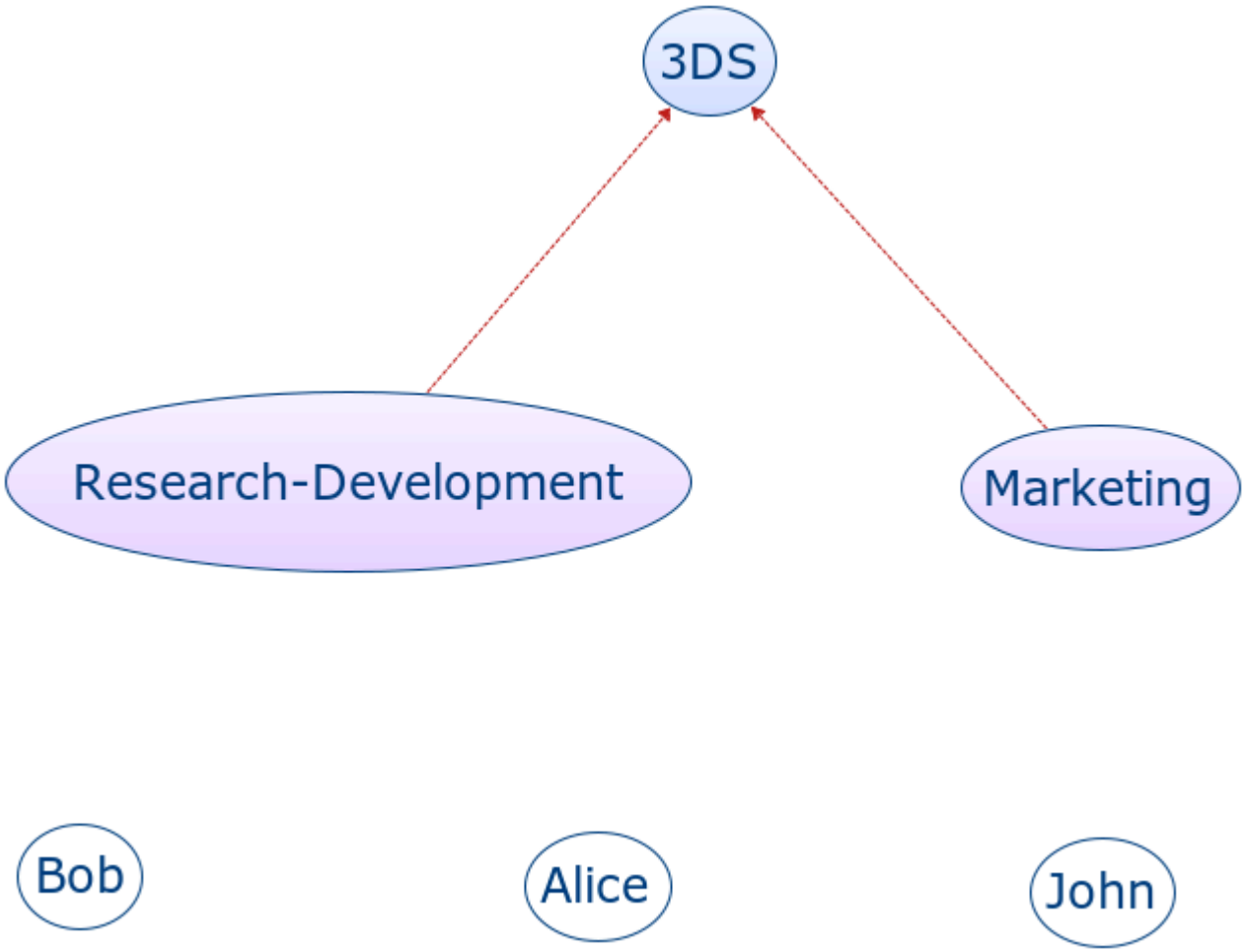
Consider that we have the following graph in the Consolidation Store.

Company's Hierarchy in the Consolidation Store



Connectors push the blue documents. The Consolidation Server creates the purple ones (as written in the previous section). If connectors send `delete` orders to all employees and companies, all nodes and edges are properly deleted from the Store. In the following graphic, transparency means that documents disappeared from the Store.

But if we send `delete` orders for all employees only, we end up in the following case, in which, the colored nodes and arcs stay in the Consolidation Store.



What about the delete processor?

If we take the case of the Company's Hierarchy and `send` delete orders for all employees, we can safely write a delete processor that for each employee, calls a `deleteDocument` method. For example, this sends twice the same `delete` order for the Research-Development service when we delete Bob and Alice sequentially. But this is okay, since the second one would become a no-operation (like `null` or `void`).

What if the `delete` orders for the employees are incremental?

Do we know for sure that the employees `delete` operation is always global? We must not send a delete order to the service that an employee belongs to. If you send a `delete` order for Bob, you cannot delete the Research-Development service since it still has an employee (Alice) attached to it. To do so, we would need to traverse the graph during the transformation phase, but such operation is only allowed at the aggregation phase. To deal with a similar case, writing a custom delete processor is not a viable solution. You would rather keep the default `delete` processor, which deletes the employee visited.

In the Aggregation Phase

Every document manually created in the aggregation processors is pushed as is to the Indexing Server once it has passed the forward rules phase.

If you want to associate that manually created document with the lifecycle of the "master" document, use the `createChildDocument` method.

When a master document is deleted, the Consolidation Server does not send a `delete` order on all existing child documents automatically, if any. This is because they are not in the storage and the Consolidation Server cannot determine their types. To delete child documents automatically, you must create a `delete` processor.

If you create a document manually, you have to handle its deletion by yourself. To do so, you can:

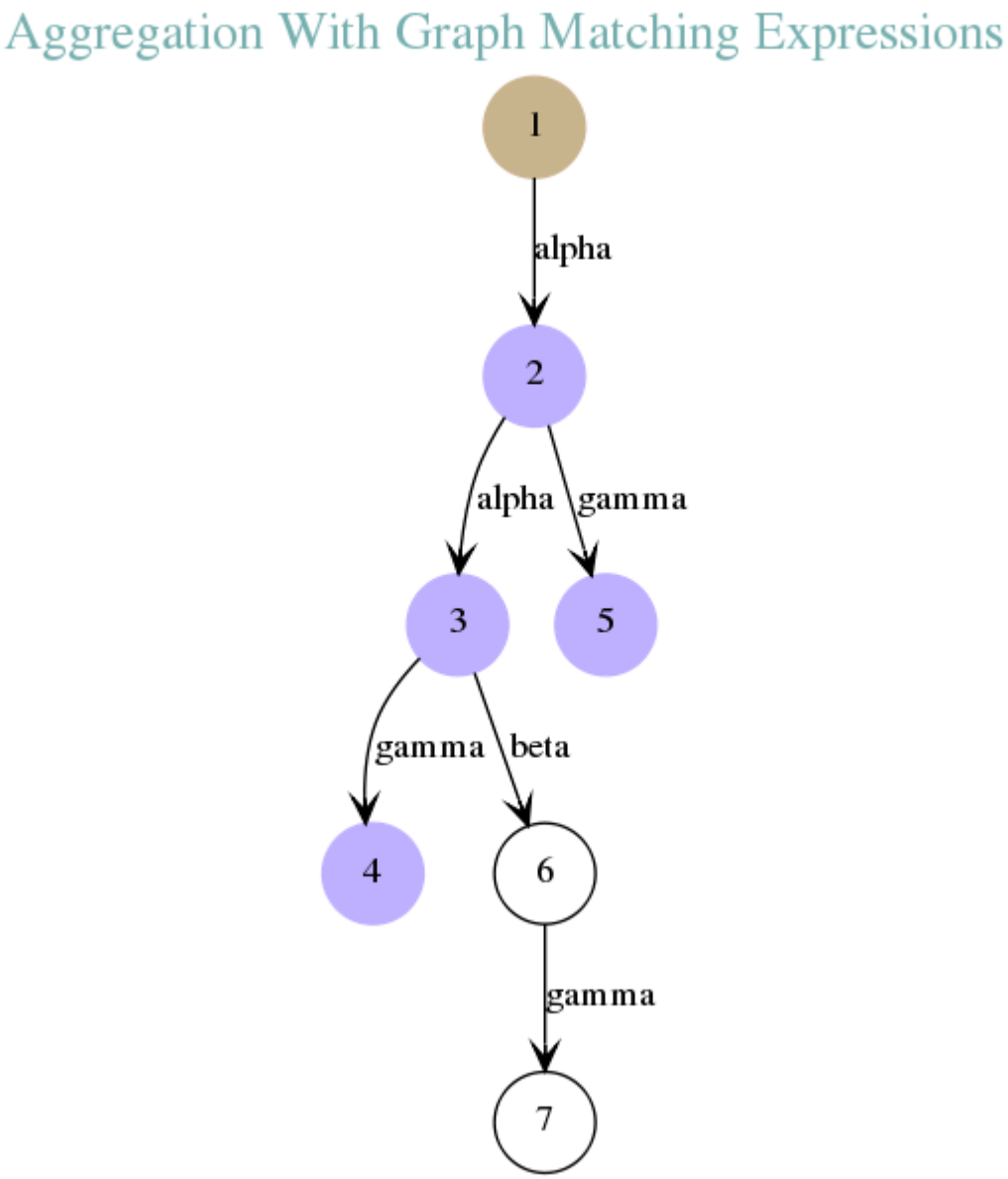
- Send delete orders to the PushAPI server of the Indexing Server directly.
- Write a custom aggregation delete processor, which would send delete orders only on the documents/URIs known/managed by you.

Impact Detection

The Impact Detection for the create/update and delete action contexts occurs only during the aggregation phase.

In the Consolidation Store, there are typically some documents that are linked with other documents, and some that are not. For the linked ones, we might want to aggregate/enrich some information from the structural properties present in these graphs.

For example, if you have document 1 that has an aggregation processor to collect all documents in purple, what will happen later on in the aggregation process if document 5 gets modified?



Since document 1 is collecting some information from document 5 to enrich its own metadata, you have to relaunch the aggregation on document 1. This is precisely what the impact detection algorithm does for you. The benefit of having such calculation happening for you undercover, is that you can code your aggregation processors (and your transformation processors) independently of the documents arrival order.

For more example of object graph matching expressions, see Appendix - Matching Expressions Grammar.

Internally, the Impact Detection algorithm is based on the strings that flow to the match operations. Consequently, every modification to your transformation and aggregation processor implementations that might change such strings, need a force aggregation action in the Administration Console or the API Console.

Two things may occur depending on how your trigger the action:

- Either you apply the action on a subset of pushed documents, by including or excluding some URIs or types.

  In that case, the internal state for the strings identified by Exalead CloudView stay as is. Keep in mind that if you choose this option, the behavior of the impact detection might be affected negatively. You might have missed to select documents that have to be reprocessed because of past modifications in the processors. If you know this is not the case, then the operation is safe.

- Or you apply the action on the whole set of documents, without specifying any URIs or Types.

  In such case, the Impact Detection reconstructs its appropriate internal states as expected. Such operation is safe in all cases.

In the contex of big bookmarks arborescence, you can reduce the number of impacted documents to reduce index latency.

When modifying or updating a document, one or more metas are changed, or one or more arcs are created or deleted in this document. Therefore, all documents below in the tree are reindexed.

To avoid reindexing all these documents, you can add a meta in addition to the name of the document to reduce the impact of update. For more information on the meta, see Appendix - Matching Expressions Grammar.

Thanks to this meta, there are less impactful aggregations, which results in smaller and faster jobs.

Troubleshooting the Configuration

This chapter describes useful tips to troubleshoot and debug your Consolidation Server configuration.

Where Can I Find the Consolidation Server Logs?

The log file is located in: `<DATADIR>\run\consolidationserver-<instance name>\`.

You can also view logs in the Administration Console > **Logs** menu.

These logs contain the Consolidation Server process logs, and all the logs emitted by the transformation and aggregation processors.

You can use a log function for debugging your processors, as in the following sample:

```
process("Foo") {
    // log the content of the processed node
    log.info it
}
```

Monitoring the Object Graph

An introspection console is available in the **Consolidation > Introspect** tab. It is a simple debugging tool, which lets you monitor your object graph graphically.

The object graph is useful to:

- Have a view of the graph node scope (what is included in the graph) to help you with refining aggregation rules.
- Understand why aggregation rules do not behave as expected. For example, when no document goes out of the Consolidation Server.

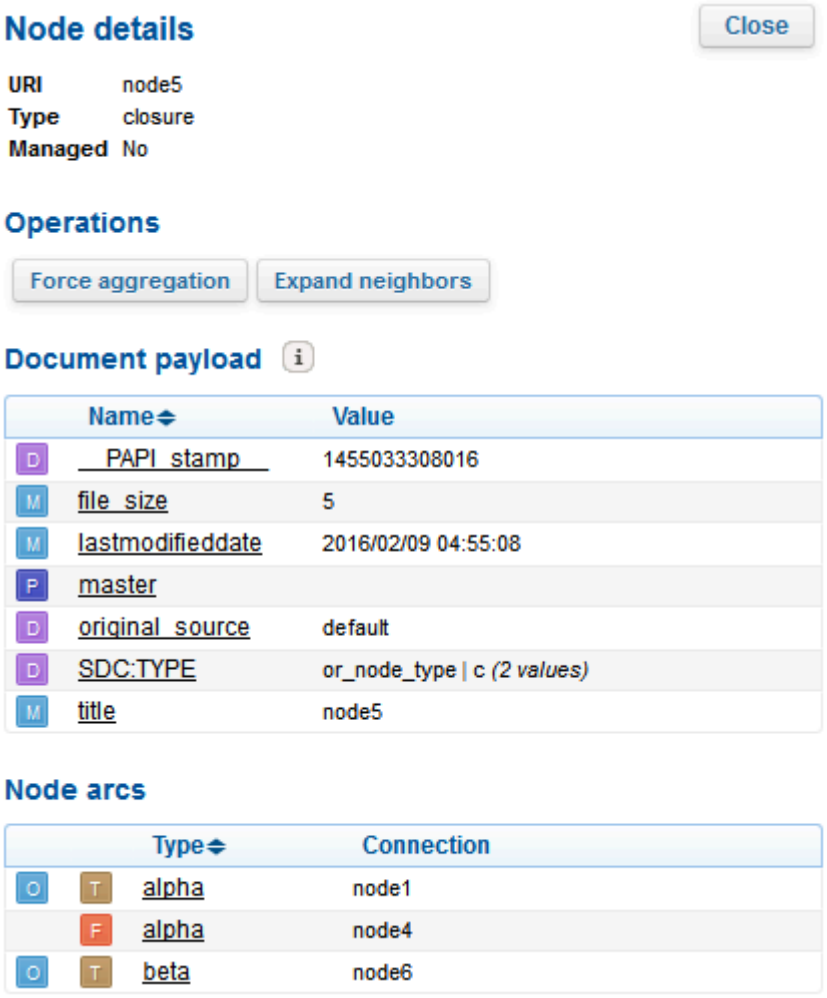
Use the Consolidation Server Introspection

Launch a full scan to fill the object graph with data.

1. From **Consolidation Server**, select the Consolidation Server instance for which you want to build an object graph.
2. In **URI(s)**, select node URIs to generate the object graph starting from these nodes. It can be helpful to filter on the node type.
3. In **Max. depth**, specify the maximum exploration depth of the graph. The nodes which are beyond this maximum depth are not displayed in the graph.
4. In **Max. arcs per node**, specify the width of the object graph. It takes the *n* first arcs of each node.
5. From **Color**, you can switch the highlight of either **Nodes** or **Arcs**.
6. Click **Submit** to generate and display the object graph.
7. With the generated object graph, you can:
   - Zoom in and out using the **+** and **-** sign or using your mouse wheel, and also pan the view.
   - Click **Export** to export the graph to a DOT file. You can also do that with the `cvdebug` command line tool. See Exporting the Object Graph.
   - Double-click a node to define it as the new root of the object graph.
   - Click a node to view its details. In the **Node details** panel, you can then:

Possible Action	to ...
Click **Force aggregation**	Start the aggregation on a node URI or on a specific node type. This is useful when you want to see the impact of the changes made on your aggregation processors, without having to rescan all sources.
Click **Expand neighbors**	• If the selected node has no arcs, it fetches its arcs with a depth = 1, and displays at most the number of arcs specified in **Max. arcs per node**. • If the selected node has some arcs, it replaces truncated arcs by real arcs for at most the number of arcs specified in **Max. arcs per node**. For example, if a node has 25 arcs, and **Max. arcs per node** is 10. When the graph is displayed, only 10 arcs are displayed for this node, and an extra arc labeled "15 truncated arcs" is added. To see these 15 hidden arcs, select the node and click **Expand neighbors**. 10 extra arcs from the truncated arcs are added to the graph. The node now

Possible Action	to ...
	has 20 arcs and 5 truncated arcs. Click **Expand neighbors** again, and the node have its 25 arcs displayed.
Check **Document payload**	See the metas, parts, and directives contained in the document.
Check **Node arcs**	See the arcs pointing to the selected node. For each, you can see its name, its direction (**F**rom/ **T**o) and if the node is the **O**wner of arc (if not, the arc comes from another document).



Simulate Matching Elements and Impact Detection

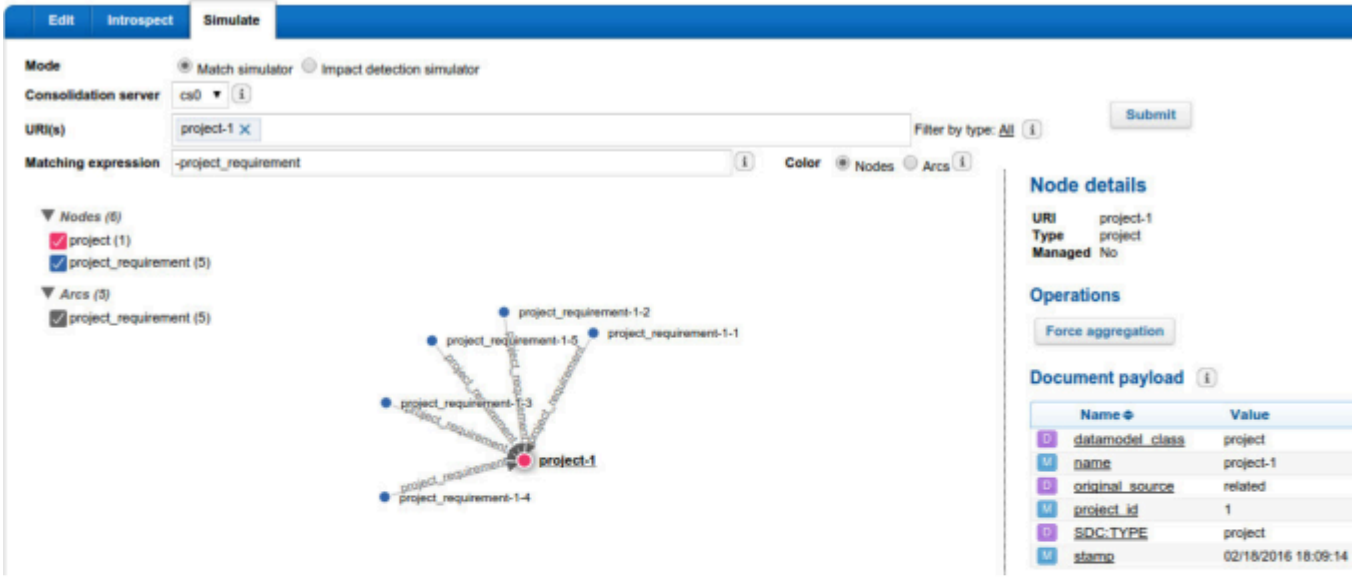
The **Simulate** tab allows you to test matching rules on a node to identify which graph elements are impacted. With this tool, you can also check for impacted nodes, according to existing detected rules, when a change occurs.

1. Choose between the two following modes:
   - **Match simulator** to enter a matching expression and see its results on the object graph (see step 4).

- **Impact detection simulator** to see the results of the impact detection for all existing rules present in your aggregation processors that have already been executed.

2. From **Consolidation Server**, select the Consolidation Server instance for which you want to simulate the impact detection an object graph.
3. In **URI(s)**, select node URIs to simulate the impact detection starting from these nodes.
4. In **Matching expression**, enter the matching rule.
5. From **Color**, you can switch the highlight of either **Nodes** or **Arcs**.
6. Click **Submit**.

You see the impact of the matching rule on the selected node URIs.



Introspection Client API Usage

The following code snippet shows the java introspection client used by the Consolidation Introspection Console for the object graph and document store introspection.

```
import com.exalead.consolidationapi.client.answer.Arc;
import com.exalead.consolidationapi.client.answer.Arcs;
import com.exalead.consolidationapi.client.answer.Document;
import com.exalead.consolidationapi.client.answer.DocumentDetails;
import com.exalead.consolidationapi.client.answer.Documents;
```

```
import com.exalead.consolidationapi.client.answer.Vertices;
import com.exalead.consolidationapi.client.answer.Type;
import com.exalead.consolidationapi.client.answer.Types;
import com.exalead.consolidationapi.client.query.GetDocumentQuery;
import com.exalead.consolidationapi.client.query.ListArcsQuery;
import com.exalead.consolidationapi.client.query.ListDocumentsQuery;
import com.exalead.consolidationapi.client.query.ListVerticesQuery;
import com.exalead.consolidationapi.client.query.ListTypesQuery;
import com.exalead.consolidationapi.client.answer.Vertex;
/**
 * Demonstrate the use of the Consolidation Server introspection client
 */
public class IntrospectionClientDemo {
    public static void main(String[] args) {
        final IntrospectionClient iC = new IntrospectionClientImpl("localhost", "1055
 // product host name, Consolidation Server monitoring port
        try {
   // ------------------
   // Graph introspection
   // ------------------
   // List arcs from uris "project-1" & "project-2", using a max exploration depth of
            final Arcs arcs = iC.listArcs(new ListArcsQuery().withUris("project-1", "
.withMaxExplorationDepth(5));
            for (final Arc arc : arcs) {
                System.out.println("Arc:  " + arc.getSource() + " -> " + arc.getTarge
            }
   // List arc types starting with prefix "rel", and returns only five types
            Types types = iC.listArcTypes(new ListTypesQuery("rel").withLimit(5));
            for (final Type type : types) {
                System.out.println("Arc type starting by 'rel': " + type.getName());
            }
   // List vertex types starting with prefix "a", and returns an unlimited number of
            types = iC.listVertexTypes(new ListTypesQuery("a").withLimit(0));
            for (final Type type : types) {
                System.out.println("Vertex type starting by 'a': " + type.getName());
            }
   // List vertices starting with prefix "a", returns only five nodes
            final Vertices nodes = iC.listVertices(new ListVerticesQuery("a").withLim
            for (Vertex vertex : vertices) {
                System.out.println("Vertex with a uri starting by 'a' : " + vertex.ge
 [type='" + vertex.getType() + "']");
            }
   // ------------------
   // Storage introspection
   // ------------------
   // List documents with uri starting with prefix "a", and print details for each
            final Documents documents = iC.listDocuments(new ListDocumentsQuery("a"))
            for (final Document document : documents) {
```

```
                System.out.println("Stored document with a uri starting by 'a': " + d
                System.out.println("Details");
                final DocumentDetails details = iC.getDocument(new GetDocumentQuery(d
                if (details != null) {
                    System.out.println("No. of metas: " + details.getMetas().size());
                    System.out.println("No. of directives: " + details.getDirectives(
                    System.out.println("No. of parts: " + details.getParts().size());
                }
            }
        } catch (final IntrospectionClientException e) {
            System.err.println("An error happened during introspection: " + e.getMess
        }
    }
}
```

Example: My Aggregation Does Not Perform What I Am Expecting

1. Make sure that the objects are correctly connected in the object graph. To do so, use the Consolidation **Introspection Console**.
2. Then you can look for stack traces in the Consolidation Server logs. You can also modify your transformation and aggregation processors to add logs.

Exporting the Object Graph

If Exalead CloudView is not running, use the `cvdebug` command-line tool solution.

Otherwise, use the Consolidation Introspection Console described in Monitoring the Object Graph

The goal is to generate an image from a text file describing the object graph in DOT format.

Export the Object Graph to a DOT File

Launch at least one full scan to fill the object graph with data.

1. Go to the `<DATADIR>/bin` directory and start the `cvdebug` command-line tool.
2. Run the following command:

   ```
   consolidation export-object-graph outputFile=<filepath> [instanceDir=<instance dir
   [instance=<Consolidation Server instance name>] [seedNode=<nodes to export>] [maxA
   [depth=<integer>]
   ```

   Where:

Argument	Description
`outputfile`	Required to indicate the file path and name of the exported `.dot` file.

Argument	Description
`[instanceDir]`	Optionally, it can be useful if you do not have a standard CV instance (for example, a debug instance or a copy of the object graph) and need to specify a Consolidation Server instance directory for the object graph to generate properly.
`[instance]`	Optionally, you can specify the Consolidation Server instance for which you want to generate the object graph. If no instance is specified, the default Consolidation Server instance `cs0` is used.
`[seedNodes]`	Optionally, you can specify a comma-separated subset of nodes to export only a subpart of the object graph starting from these nodes. You cannot generate and display an SVG with millions of nodes and millions of arcs. This option therefore allows you to drastically reduce the graph to be exported.
`[maxArcsPerNode]`	Optionally, you can specify the object graph width. It takes the *n* first arcs of each node.
`[depth]`	Optionally, you can limit the graph exploration starting from the nodes specified with the `seedNodes` argument. The nodes which are beyond this maximum depth are not displayed in the graph.

Once the DOT file is generated, you see all the nodes and arcs according to the arguments passed to the `export-object-graph` command. Nodes that do not exist, but to which arcs are pointing, are highlighted in red in the object graph. This is useful to spot them.

Convert the DOT File to Another Image Format

From the generated DOT file, it is then possible to generate the image to SVG, PNG, etc. formats, using the dot binary delivered with the GraphViz free suite.

1. Use SVG as the output format, since it allows you to search for text within the graphical display. This is convenient when you want to find a node in the generated graph. Here is the typical

command line used to create an SVG image from a text file describing the object graph in DOT format: `dot -Tsvg store.dot -o store.svg`

Checking the Consolidation Storage Content

If you have pushed many documents to the Consolidation Server and observe missing output views or unexpected behavior, you can directly export the documents from the Consolidation storage and check their consistency.

For example, you can verify that metas have correct values.

1. Go to the `<DATADIR>/bin` directory and start the `cvdebug` command-line tool.
2. Run the following command:

   ```
   consolidation export-document-store outputFile=<filepath> [instanceDir=<instance d
    [instance=<Consolidation Server instance name>]
   ```

   Where:

   - `outputfile` – (Required) This argument indicates the file path and name of the exported `.dot` file.
   - `[instanceDir]` – Optionally, it can be useful if you do not have a standard CV instance (for example, a debug instance or a copy of the object graph) and need to specify a Consolidation Server instance directory for the object graph to generate properly.
   - `[instance]` – Optionally, you can specify the Consolidation Server instance for which you want to generate the object graph. If you do not specify any instance, the default Consolidation Server instance `cs0` is used.

Once the Consolidation storage is exported (to a JSON file), you are able to see all the documents (nodes) it contains, and check their metas.

Observing the Processors' Consumption

Get a Global View of the Consolidation Server Processors

1. Go to the Monitoring Console.
2. Expand **<HOSTNAME> > Services > Exalead > Consolidation >** `cbx`.

   The graphs show you:

   - In **Latency** – the duration of the overall consolidation job and the duration for each processing phase in seconds.

- In **Volume** – the overall number of documents treated by the consolidation job and the number of documents that went out of each processing phase.
- You also have separate folders containing the details of each processor used by the Consolidation Server, if **perfMonitored="true"** in the `ProcessorConfig` defined in the API Console. See Get a Finer Debugging Granularity on a Specific Processor).

Using the graphs, you can spot which processing phase takes too long to perform its job, and if the number of documents going out of it is not consistent.

Check If the Consolidation Storage Compact Works Properly

1. Go to the Monitoring Console.
2. Expand **<HOSTNAME> > Services > Exalead > Consolidation >** `cbx` **> Compacter > Slot counts**.

   If the number of slots for the object graph store and the number of slots for the document store, exceeds 100 slots or keeps growing, start a full compact operation on the Consolidation storage, as explained in the following steps.
3. Go to the API Console.
4. Select **Manage**.
5. Search for `compactStorage` and:
   a. Specify your instance name.
   b. Click **Send**.
6. Restart Exalead CloudView processes
   a. Search for the `restartHost` method
   b. Click **Send**.

Get a Finer Debugging Granularity on a Specific Processor

1. Go to the API Console.
2. Select **Manage**.
3. Search for `setConsolidationConfigList` and:
   a. For one processor, define `perfMonitored="true"` as attribute. For example:

   ```
   <conso:AggregationProcessorConfig perfMonitored="true" enabled="true"
   code="process(&amp;quot;&amp;quot;) {&amp;#xA; yield it&amp;#xA;}" mime="text/x-groovy" name="de
   ```

   b. Click **Save**.
4. Restart Exalead CloudView processes
   a. Search for the `restartHost` method.

b. Click **Send**.

5. Go to the Monitoring Console.
6. Expand **<HOSTNAME> > Services > Exalead > IConsolidation > cbx**.

You now have monitoring logs specific to the processor specified in step 3.

Consolidation Server Fails with Out of Memory Error

This procedure details what you can do if the Consolidation Server crashes with a `java.lang.OutOfMemoryError` during the transformation phase.

1. First, you may want to increase the `Xmx` of your Consolidation Server, but it is not the only solution.

   It is better to add a **Commit trigger based on size** which fits your use case. Consolidation server commits are cheap. Do not hesitate to commit regularly but review your aggregation trigger conditions, as committing frequently does not necessarily mean that you want to run an aggregation for each commit.

Use Cases

This chapter describes use cases for the Consolidation Server illustrated by sample application scenarios.

About Consolidation Use Cases

This chapter shows typical consolidation use cases through a predefined application. This means that you do not have to create the data model nor the mashup application pages. You only have to create and configure connectors, transformation and aggregation consolidation processors.

What Are Our Data Sources

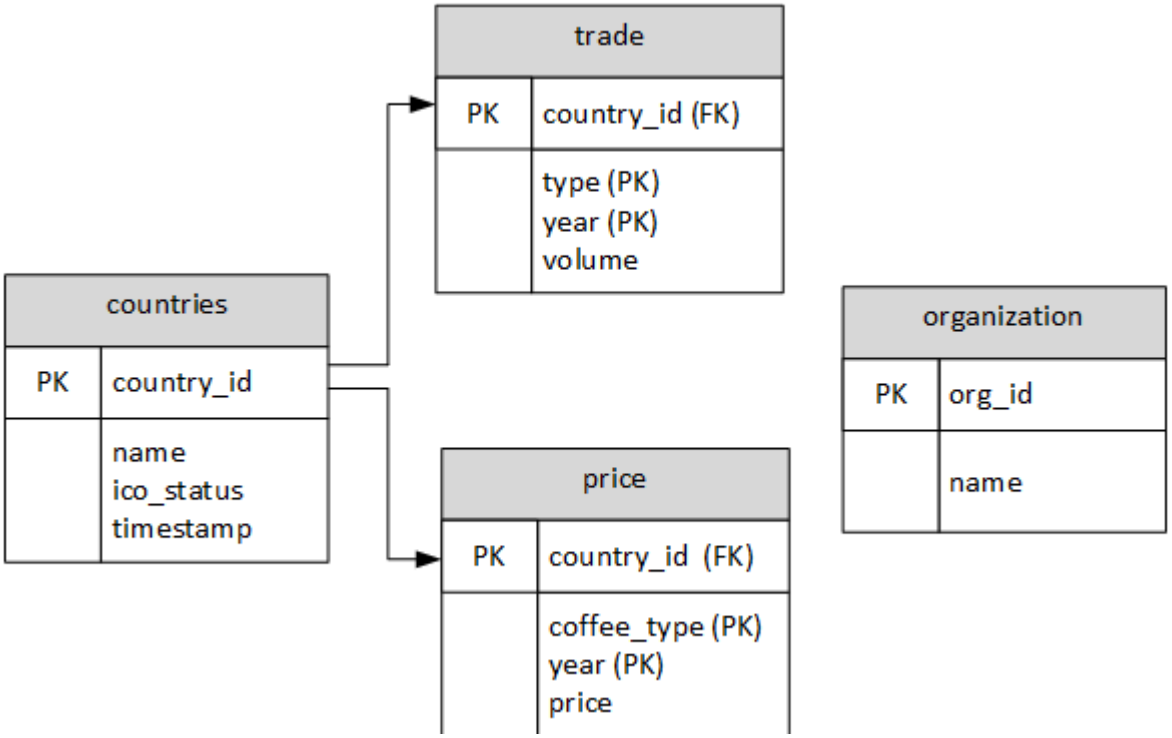
For this tutorial, we use two kinds of data sources both contained in `<INSTALLDIR>/docs/cvapp-coffee-sample/coffee_data.zip`:

- the `coffee.db` sample SQLite database contains four tables and the `country_id` field as primary key (PK) to make inner joins between them.

  Note: `organization` does not have `country_id` as primary key.

- A set of pdf files containing text and graphics.

Database Schema for the Sample Database



What We Want to Do Functionally

We want to use this database and these PDF files to consolidate data in a prepackaged application contained in `<INSTALLDIR>/docs/cvapp-coffee-sample/cvapps_coffee_v1_4.zip`. Through several use cases, we will try to expose the various capabilities offered by the Consolidation Server.

About Code Samples

The code samples in this chapter use cases are written in Groovy. Groovy allows you to add inline coding within the product is therefore easier for Training purpose.

Recommendation: For production deployment and maintainability, use Java language.

Deploy the Coffee Sample Data

The sample is provided in: `<INSTALLDIR>/docs/cvapp-coffee-sample`

It contains two archives: `coffee_data.zip` and `cvapps_coffee_v1_4.zip`.

Exalead CloudView must be installed and running.

Extract Coffee Data

1. Create a directory for coffee data outside of the `<INSTALLDIR>`.

This directory is referred to as <INPUTDIR>.

2. Copy the coffee_data.zip archive within this directory and unzip it.

Deploy the Coffee Sample Configuration

1. Create a temporary directory <TMPDIR> outside of the <INSTALLDIR>.
2. Copy the cvapps_coffee_v1_4.zip archive within this directory.
3. Go to the <DATADIR>/bin directory.
4. Start the import using the following command:

   ```
   cvadmin apps install apps-file=<TMPDIR>/cvapps_coffee_v1_4.zip noversioncheck=true
   ```

   For <TMPDIR>, enter the full path of the zip file.

   If successful the following lines display on your prompt:

   ```
   [debug] [main] [gateway.cvapps-installer] Applying configuration...
   [info] [main] [gateway.cvapps-installer] Installation of application completed.
   ```

5. Wait until Exalead CloudView is fully restarted.

   You now have a coffee data model in your instance and a coffee application.

6. Clear the index.

UC-1: Consolidating Data from Two Sources

For this use case, we want to gather document information about countries coming from a database and a file system on the same index.

The prerequisite is that there is a known link between database records and files. In the provided sample, the file name contains the id of the database record (stored in the country_id field).

What we want to do is create a link between a country record and a PDF document inside the Consolidation Server. The country object is enriched with the PDF file during the aggregation step.

Step 1 - Define the Connectors Corresponding to Each Source

Create the Filesystem Connector

In this use case, we have a very small set of PDF documents to push to the Consolidation Server using the Files connector. To reproduce this example with a real corpus, if your documents have large binary parts, the Consolidation Server cache ends up with a disk footprint close to the size of the indexed corpus.

For real use cases, convert document binary parts before pushing documents into the Consolidation Server:

- Go to the **Advanced** tab and a **Convert PushAPI Filter** to the Files connector.
- Extract text content only (and exclude binary parts) by setting the **Conversion mode** to **Text**.

1. In the Administration Console, go to **Index > Connectors** and click **Add connector**.
   a. In **Name**, enter `countryfiles`.
   b. For **Type**, select the **Files** connector.
   c. For **Push to PAPI server**, select the `Consolidation server cbx0` instance.
   d. Click **Accept**.
2. For **Store documents in data model class**, choose the `document` class.
3. In **Filesystem paths**, enter the following path: `<INPUTDIR>/pdf`
4. Click **Save**.

Create the Database Connector

1. In the Administration Console, go to **Index > Connectors** and click **Add connector**.
   a. In **Name**, enter `country`.
   b. For **Type**, select the **JDBC** connector.
   c. For **Push to PAPI server**, select the `Consolidation server cbx0` instance.
   d. Click **Accept**.
2. For **Store documents in data model class**, choose the `country` class.
3. In **Connection parameters**:
   a. For **Driver**, enter `org.sqlite.JDBC`
   b. For **Connection string**, enter `jdbc:sqlite://<INPUTDIR>/coffee.db`
   c. Click **Test connection**. The database connector automatically connects to the database.
4. In **Query parameters**:
   a. For **Synchronization mode**, select **Full synchronization**
   b. For **Initial query**, enter: `Select country_id, ico_status, name from countries`
5. Click **Retrieve fields**.
6. Define the `country_id` field as primary key.
   a. Click the `country_id` field to expand it.
   b. Select **Use as primary key**.
7. Click **Save**.

Step 2 - Configure Consolidation

Configure the Transformation Processor

1. Go to **Index > Consolidation**
2. Add a new **transformation** processor:
   a. Select **Groovy** as format
   b. For **Name**, enter `Files`
   c. Click **Accept**
3. For **Source connector**, select `countryfiles`
4. Replace the default code by the following one:

   Groovy code

```
// Process all nodes coming from the selected source connector
process("") {
 // Extract the country id from the filename.
 // For example, for "brazil.pdf", we want to extract "brazil".
 String filename = it.metas.getValue("file_name");
 def values = filename.split('\\.');
 log.info "doc uri:[" + it.getUri() + "] countryId:[" + values[0] + "]";
 // Link the filesystem document to its related “countries” database record.
 // The default URI of a database record is: "<fieldname>=<value>&"
 it.addArcFrom("describedBy", "country_id=" + values[0] + "&");
 }
```

   Java equivalent code

```
@Override
 public void process(IJavaAllUpdatesTransformationHandler handler, IMutableTransfo
        document.setType("document");
        final String filename = document.getMeta("file_name");
        if (filename == null || filename.isEmpty()) {
            throw new Exception("File name not available");
        }
        final String[] values = filename.split("\\.");
        if (values == null || values.length == 0) {
            throw new Exception("Invalid file name");
        }
        LOGGER.info("doc uri:[" + document.getUri() + "] countryId:[" + values[0]
        document.addArcFrom("describedBy", "country_id=" + values[0] + "&");
}
```

With this transformation processor, we have achieved to link files to their related database records.

Configure the Aggregation Processor

1. Add an **aggregation** processor:
   a. Select **Groovy** as format
   b. For **Name**, enter `Countries_UC_1`
   c. Click **Accept**
2. Replace the default code by the following one:

   Groovy code

```
// Process nodes having the “country” type
// The node type is deduced by the document class automatically
process("country") {
 log.info "country found: " + it.metas.name;
 // Find nodes related to the country
 // Goal: Create a “country” consolidated document with information coming from th
    it.metas.hasfile = "no";
    for (path in match(it, "describedBy[document]")) {
 // If a valid path is found, retrieve its last element
 last = path.last()
 log.info "File found: " + last.getUri();
 // Retrieve the binary parts of the found nodes
 // To get all parts: it.parts.getMap().putAll(last.parts.getMap());
 // To get the master part only:
        it.parts.master += last.parts.master;
        it.metas.hasfile = "yes";
  }
}
```

   Java equivalent code

```
@Override
 public void process(IJavaAllUpdatesAggregationHandler handler, IAggregationDocume
        final String countryName = document.getMeta("name");
        if (countryName == null || countryName.length() == 0) {
            throw new Exception("Invalid country name '" + countryName + "'");
        }
        LOGGER.info("Country found: " + countryName);
        // find document related to the country
        // Goal: be able to consolidate information of pdf document with country d
        final List<IAggregationDocument> pathsEnds = GraphMatchHelpers.getPathsEnd
        for (IAggregationDocument file : pathsEnds) {
            LOGGER.info("File found: " + file.getUri());
            document.withPart("master", file.getPart("master"));
            document.withMeta("hasfile", "yes");
        }
}
```

3. Save and apply the configuration.

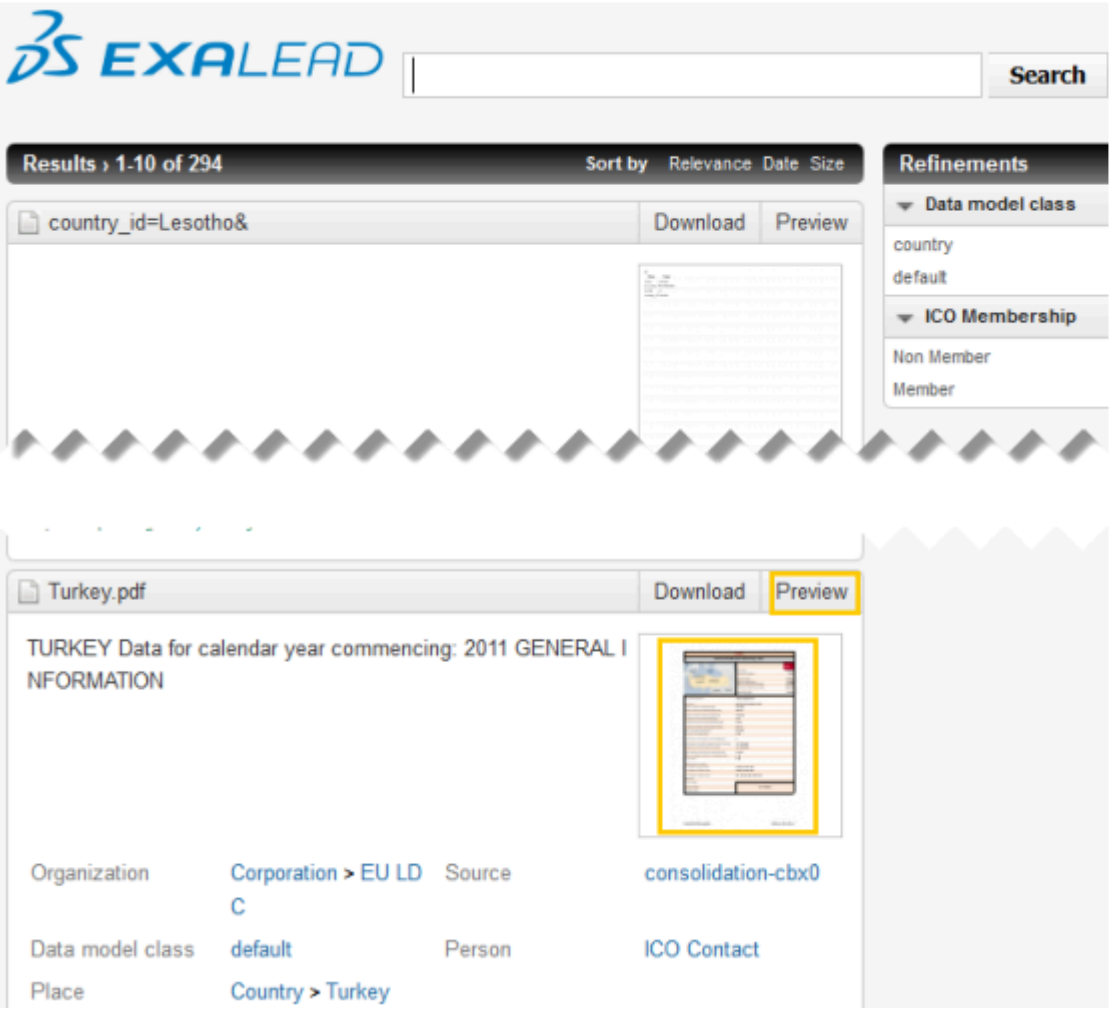
   Note: It is also possible to consolidate security tokens, using the `security` meta. After `it.metas.hasfile = "yes";` add `it.metas.security += last.metas.security;`

Step 3 - Scan Source Connectors and Check What Is Indexed

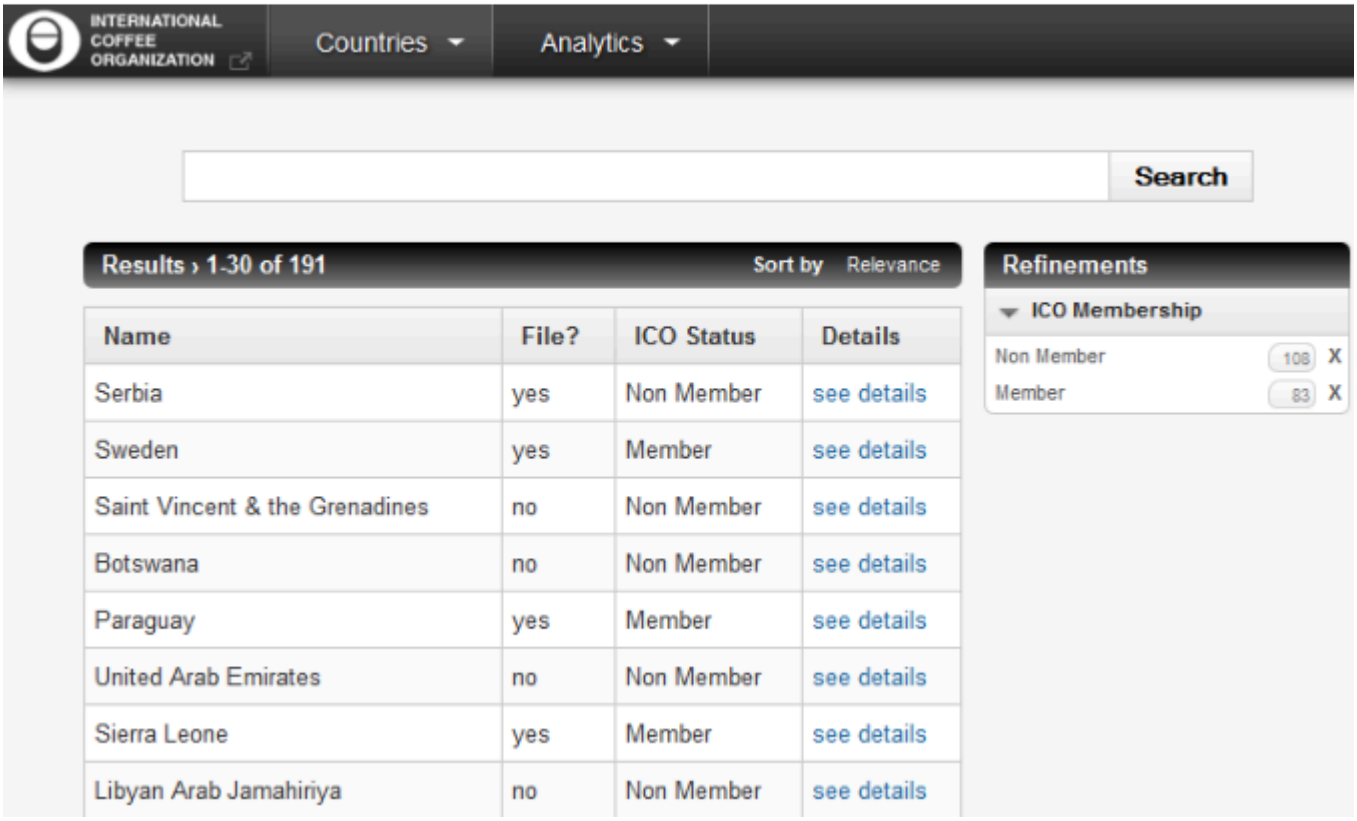
1. Go to the **Home** page and under the connectors list, click **Scan** for the Files and JDBC connectors.

   Note: In the **Connectors** list, a **consolidation-<instance name>** row displays status information about consolidation. All documents and countries are indexed.

2. Open the Mashup UI application search page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/search`
3. Check that country documents have associated parts (thumbnails/previews are available).



4. To get a consolidated view, go to: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/searchcountry_v1`



The following graphic shows what we achieved on the object graph.

UC-2: Enriching Child Documents with Parent Document Metas

Flattening data allows you to build powerful queries in Exalead CloudView, and the Consolidation Server is the right tool to achieve this kind of operation.

In the provided coffee sample, trade records contain, for each year and each country, a volume of exchanges for each type of trade (import, export, re-export). However, the ICO membership status is only present on the country record. For a relational database, you could write an SQL join query to retrieve trade only for the countries that are members of the ICO. For an index engine, it is more efficient to move down this information directly to the trade record at indexing time.

We assume that UC-1 has been completed.

Step 1 - Define the Source Connector for Trades

1. In the Administration Console, go to **Index > Connectors** and click **Add connector**.
   a. In **Name**, enter `trades`.
   b. For **Type**, select the **JDBC** connector.
   c. For **Push to PAPI server**, select the `Consolidation server cbx0` instance.
   d. Click **Accept**.
2. For **Store documents in data model class**, choose the `trade` class.
3. In **Connection parameters**:
   a. For **Driver**, enter `org.sqlite.JDBC`
   b. For **Connection string**, enter `jdbc:sqlite://<INPUTDIR>/coffee.db`
   c. Click **Test connection**. The database connector automatically connects to the database.
4. In **Query parameters**:
   a. For **Synchronization mode**, select **Full synchronization**
   b. For **Initial query**, enter `select country_id, type, volume, year from trade`
5. Click **Retrieve fields**.
6. Define the `country_id`, `type`, and `year` fields as primary keys.
   a. Click the `country_id` field to expand it.
   b. Select **Use as primary key**.
   c. Repeat the operation for the `type` and `year` fields.
7. Click **Apply**.

Step 2 - Configure Consolidation

Configure the Transformation Processor

1. Go to **Index > Consolidation**
2. Add a new **transformation** processor:
   a. Select **Groovy** as format

b. For **Name**, enter `Trades`

c. Click **Accept**

3. For **Source connector**, select `trades`

4. Replace the default code by the following one:

```
// Process all nodes
process("") {
 // Link trade records to nodes having the “country” type with a link based on the
 // (i.e.; Import / Export / reExport) as arc label
 it.addArcFrom(it.metas.getValue("type"), "country_id=" + it.metas.getValue("count
}
```

With this processor, we have achieved to link trades to their related countries.

Configure the Aggregation Processors

1. Add an **aggregation** processor:

a. Select **Groovy** as format

b. For **Name**, enter `Trades_UC_2`

c. Click **Accept**

2. Replace the default code by the following one:

```
// Process nodes having the “trade” type
process("trade") {
 log.info "trade found : " + it.metas.year + "_" + it.metas.country_id + "_" + it.
 // Find the ICO status member from nodes.
 // It is now possible to use a dynamic path based on node meta
 for (path in (match( it, "-" + it.metas.getValue("type") + "[country]" )) ) {
 // Retrieve the last path element
 last = path.last();
 log.info "Country found : " + last.getUri();
 // Get the “membership” meta value from nodes having the “country” type
 it.metas.membership = last.metas.getValue("ico_status");
 }
}
```

3. Save and apply the configuration.

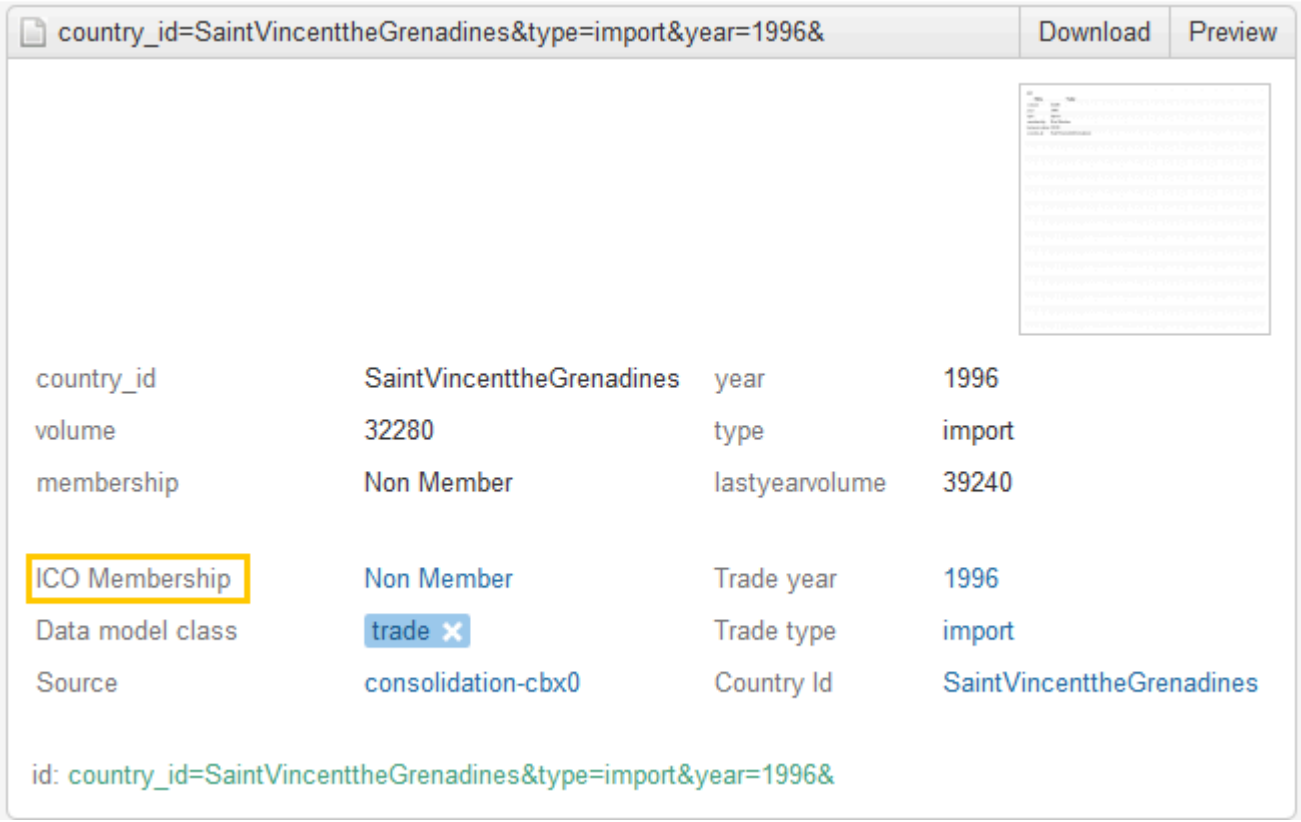
Step 3 - Scan Source Connectors and Check What Is Indexed

1. Go to the **Home** page and under the connectors list, click **Scan** for the `trades` JDBC connector.

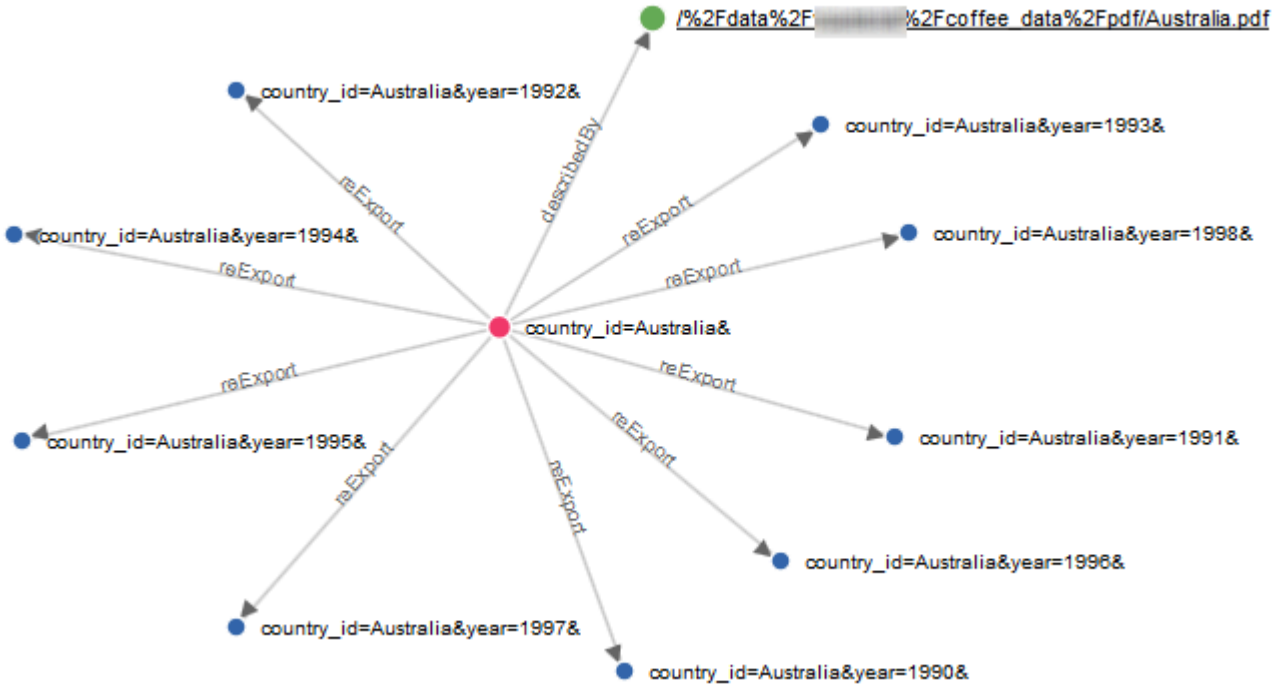
   Trades are indexed.

2. Open the Mashup UI application search page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/search`

3. Check that trade documents have an **ICO Membership** facet available.



The following graphic shows what we achieved on the object graph.



UC-3: Consolidating Information on a View Document

When flattening data, it is also interesting to build the most complete "View" to answer global queries.

In the coffee sample, we might want to search for ICO country members, having some trade record of import type, and filter the global volume of trade above a specific threshold. We also want to add the coffee varieties sold by producing countries.

We assume that previous UCs have been completed.

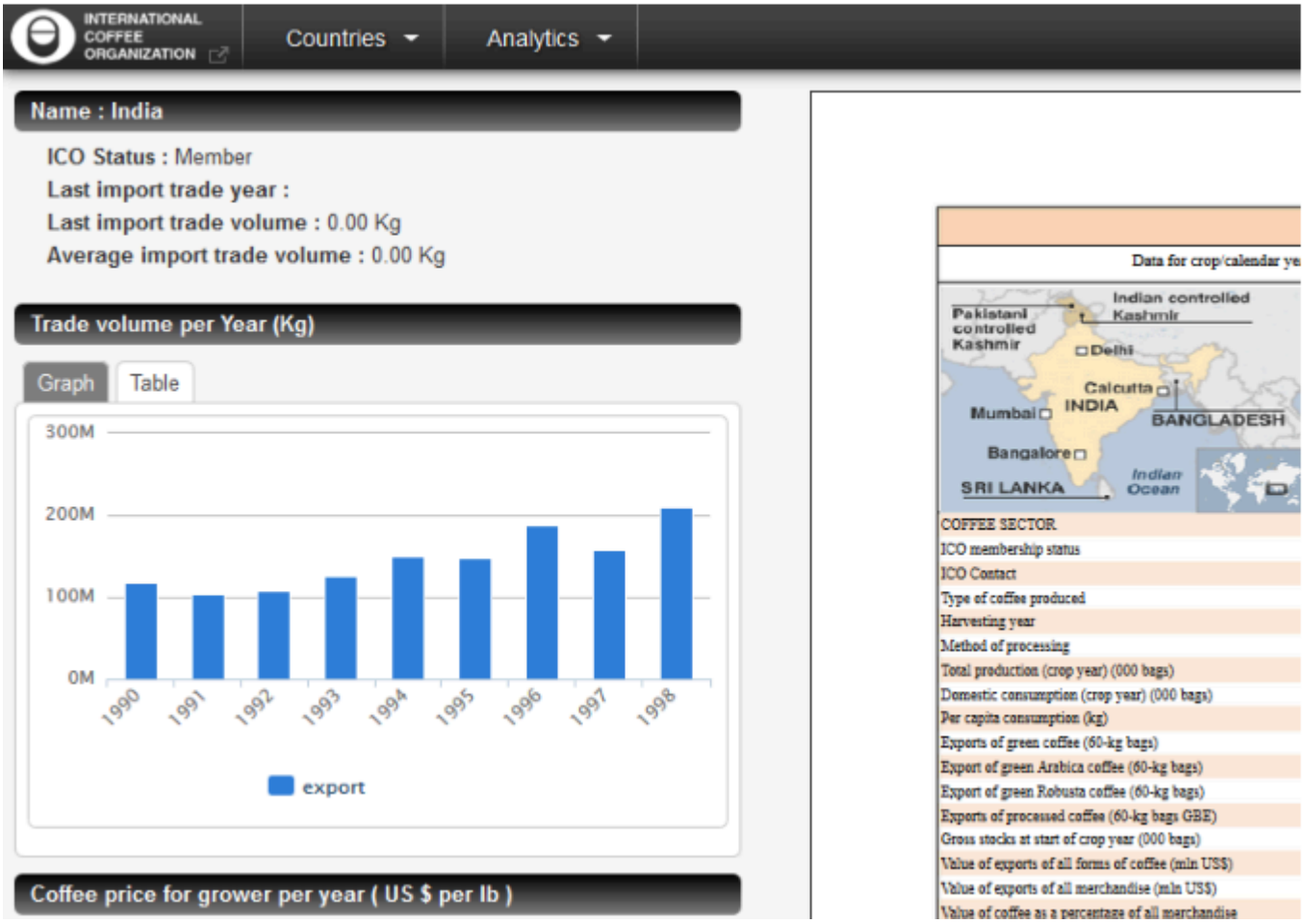
Step 1 - Check Existing Data

You can see the provided application sample. To access its front page:

1. Open the Mashup UI application: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/searchcountry_v1`

   Countries are displayed with their **ICO status** and **yes** flags show if they have associated PDF files (UC-1).

2. You can click the **see details** link of a country. It provides a 360° view of all known data for this country.



Step 2 - Add Trade Info on Countries

This procedure describes how to calculate for each country: the quantity of imported coffee for the last year, and the average quantity of imported coffee through time.

1. Add an **aggregation** processor:
   a. Select **Groovy** as format
   b. For **Name**, enter `Countries_UC_3_1`
   c. Click **Accept**
2. Replace the default code by the following one:

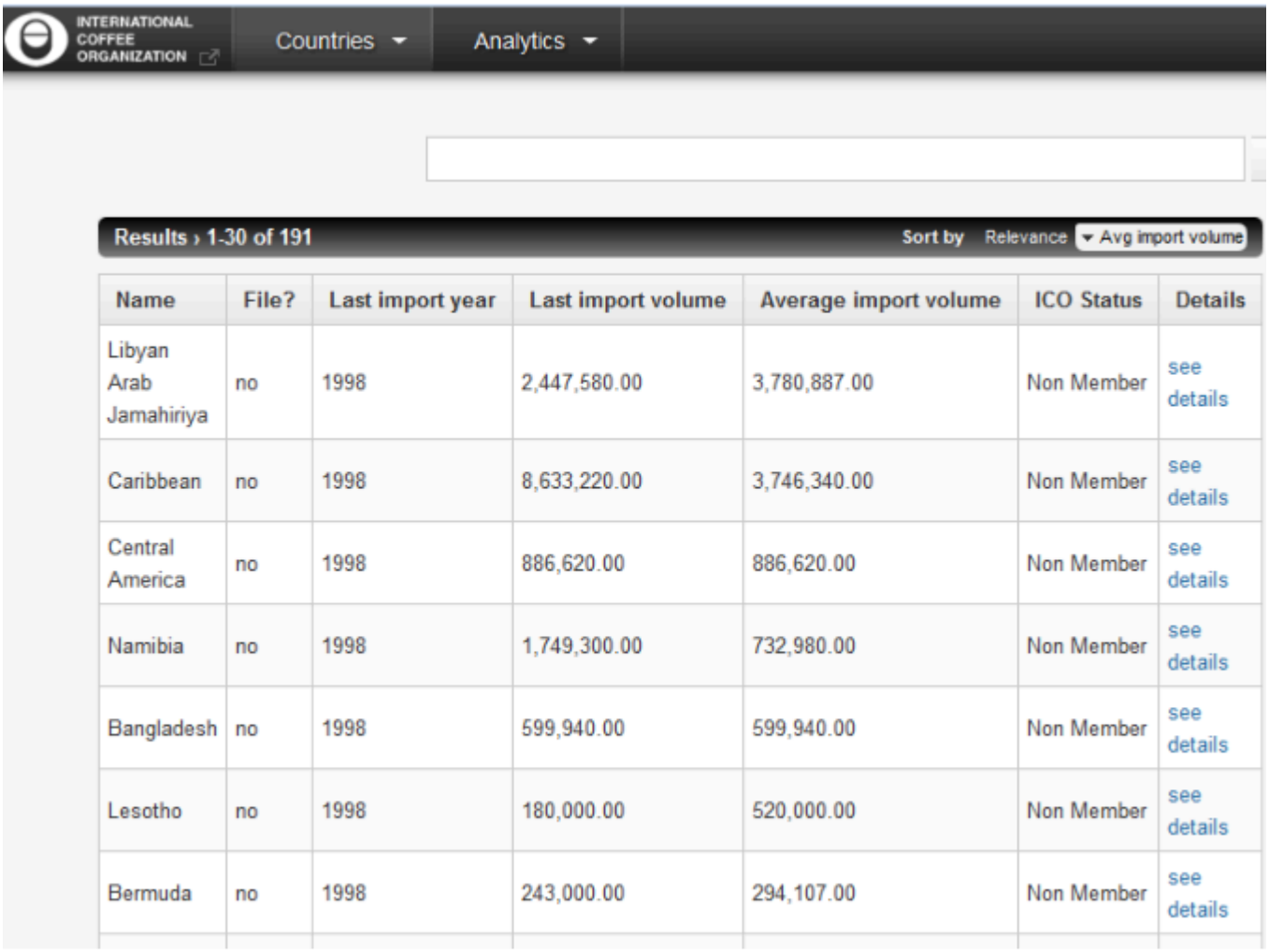
```
// Process nodes having the “country” type
process("country") {
 // Add the import volume value of the last year
 // Goal: Be able to sort countries based on import trade activity
 year = 0;
 volume = 0;
 nbTrade = 0;
 // Big Integer
 def avgVolume = 0G;
 // Get import trade only, using the path label, i.e., "import"
```

```
for (path in match(it, "import[trade]")) {
  // If a valid path is found, retrieve its last element
  last = path.last();
  log.info "trade found: " + last.getUri();
  // Get trade volume for the last year
  if (last.metas.getValue("year")?.toInteger() > year ) {
    year = last.metas.getValue("year")?.toInteger();
  }
  // Add volume to calculate the total import trade volume
  volume = last.metas.getValue("volume")?.toInteger();
  avgVolume += volume;
  nbTrade++;
}
// Add metas to countries having import trade
if (nbTrade!=0) {
it.metas.import_lastvolume = volume;
it.metas.import_lastyear = year;
avgVolume = Math.ceil(avgVolume / nbTrade).intValue();
it.metas.import_averagevolume = avgVolume;
}
}
```

3. Save and apply the configuration.

Step 3 - Scan the Source Connector and Check What Is Indexed

1. Go to the **Home** page.
2. Click **Force aggregation**, and enter `country` as type.
3. Open the following Mashup UI application search page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/searchcountry_v2`
4. Check that countries now have the following metas: **Last import year**, **Last import volume**, **Average import volume**.
5. You can now use the average import volume as search criteria. For example, sort by **Avg import volume**.



Name	File?	Last import year	Last import volume	Average import volume	ICO Status	Details
Libyan Arab Jamahiriya	no	1998	2,447,580.00	3,780,887.00	Non Member	see details
Caribbean	no	1998	8,633,220.00	3,746,340.00	Non Member	see details
Central America	no	1998	886,620.00	886,620.00	Non Member	see details
Namibia	no	1998	1,749,300.00	732,980.00	Non Member	see details
Bangladesh	no	1998	599,940.00	599,940.00	Non Member	see details
Lesotho	no	1998	180,000.00	520,000.00	Non Member	see details
Bermuda	no	1998	243,000.00	294,107.00	Non Member	see details

Step 4 - Add New Categories on Countries

Define the Connector for the Prices Source

1. In the Administration Console, go to **Index > Connectors** and click **Add connector**.
   a. In **Name**, enter `prices`.
   b. For **Type**, select the **JDBC** connector.
   c. For **Push to PAPI server**, select the `Consolidation server cbx0` instance.
   d. Click **Accept**.
2. For **Store documents in data model class**, choose the `price` class.
3. In **Connection parameters**:
   a. For **Driver**, enter `org.sqlite.JDBC`
   b. For **Connection string**, enter `jdbc:sqlite://<INPUTDIR>/coffee.db`
   c. Click **Test connection**. The database connector automatically connects to the database.
4. In **Query parameters**:

   a. For **Synchronization mode**, select **Full synchronization**
   b. For **Initial query**, enter `select country_id, coffee_type, year, price from price`
5. Click **Retrieve fields**.
6. Define the `coffee_type`, `country_id`, and `year` fields as primary keys.
   a. Click the `coffee_type` field to expand it.
   b. Select **Use as primary key**.
   c. Repeat the operation for the `country_id` and `year` fields.
7. Click **Apply**.

Configure the Transformation Processor

1. Go to **Index > Consolidation**
2. Add a new **transformation** processor:
   a. Select **Groovy** as format
   b. For **Name**, enter `Prices`
   c. Click **Accept**
3. For **Source connector**, select `prices`
4. Replace the default code by the following one:

```
// Process all nodes
process("") {
 // Link prices records to nodes having the “country” type
 it.addArcTo("producedBy", "country_id=" + it.metas.getValue("country_id") + "&");
}
```

Configure the Aggregation Processor

1. Add an **aggregation** processor:
   a. Select **Groovy** as format
   b. For **Name**, enter `Countries_UC_3_2`
   c. Click **Accept**
2. Replace the default code by the following one:

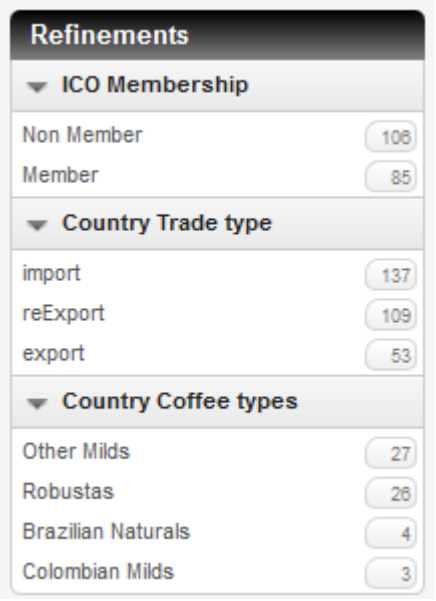
```
// Process nodes having the “country” type
process("country") {
 // Add all trade types on countries
 if (match(it, "import[trade]")) {
 it.metas.tradetype.add("import")
 }
 if (match(it, "export[trade]")) {
 it.metas.tradetype.add("export");
```

```
}
if (match(it, "reExport[trade]")) {
it.metas.tradetype.add("reExport")
}
// Add all coffee types to producing countries
it.metas.coffeetype +=
   // Get all paths to price nodes
   match(it, "-producedBy[price]") *.last()
   // fetch the last node of each path
      // retrieve the coffee_type meta values for all price nodes
      .collect{n-> n.metas.getValue("coffee_type") }
      .unique() // dedup collected meta values
   // or if multi valued: .collect{n-> n.metas.coffee_type}.flatten().unique()
}
```

3. Save and apply the configuration.

Step 5 - Rescan Source Connectors and Check What Is Indexed

1. Go to the **Home** page and under the connectors list, click **Scan** for the `country` JDBC connector and the `prices` JDBC connector.
2. Open the Mashup UI application search page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/searchcountry_v3`
3. Check that countries now have the following facets: **Country Trade type** and **Country Coffee types** (the metas created previously in the aggregation processor are mapped to these facets).



UC-4: Calculating Trends

Calculating trends with an index is really complex to achieve, as you can only calculate aggregates using the data of a single result. The easiest way to calculate trends is therefore to precompute data. You can perform this operation with an aggregation processor.

We assume that previous UCs have been completed.

Step 1 - Configure an Aggregation Processor for Trades

1. Add an **aggregation** processor:
   a. Select **Groovy** as format
   b. For **Name**, enter `Trades_UC_4`
   c. Click **Accept**
2. Replace the default code by the following one:

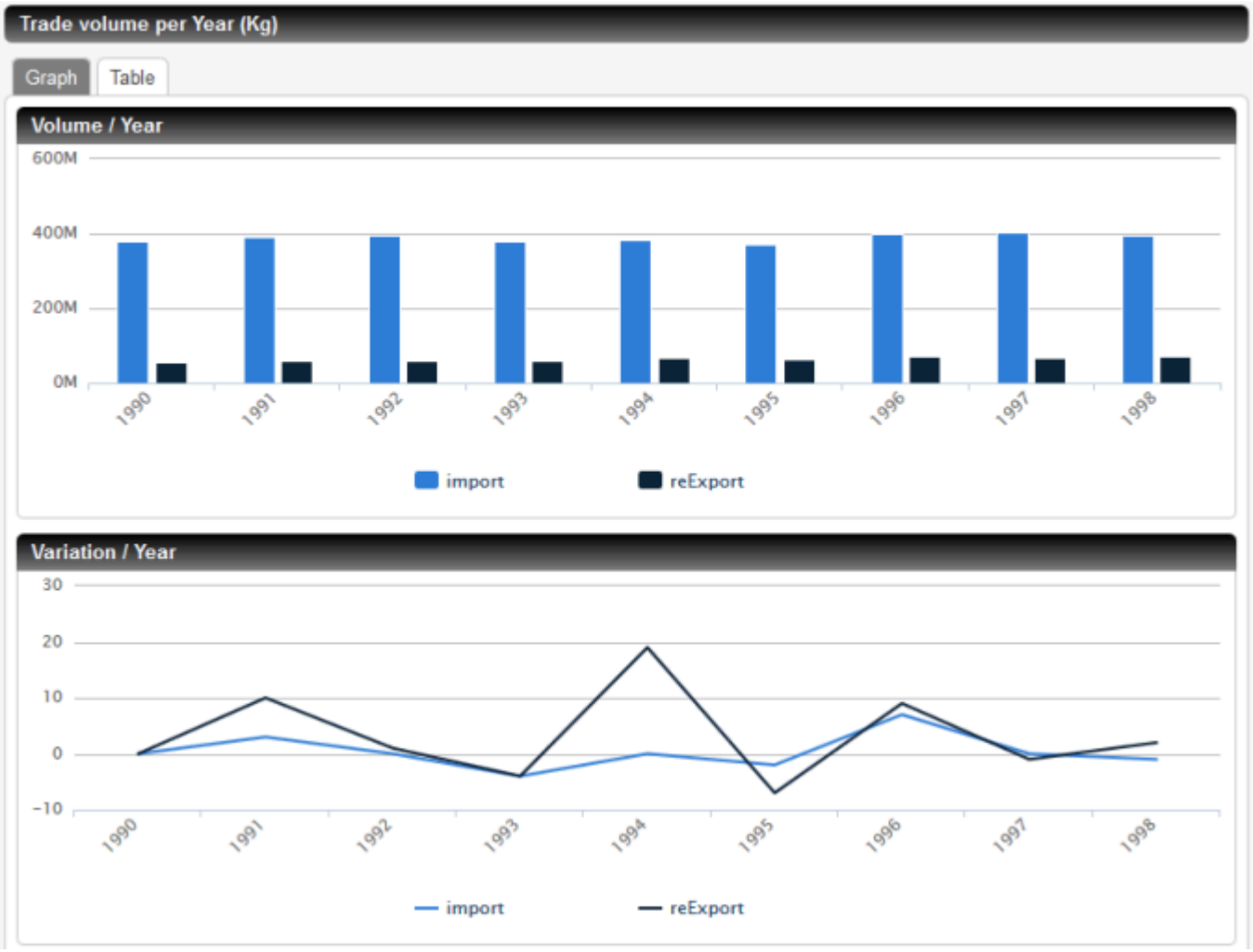
```
// Process nodes having the “trade” type
process("trade") {
   log.info "trade found for tendencies: " + it.metas.year + "_" + it.metas.countr
it.metas.type ;
   // default value
   it.metas.lastyearvolume = it.metas.getValue("volume");
   // Find previous year value to show tendencies
   // It is possible to build the path using a meta of the node
   for (path in match(it, "-" + it.metas.getValue("type") + "[country]" + "." + it
 + "[trade]" )) {
   // searching for path -export.export or -import.import or -reExport.reExport
   // Retrieve the last element of the path
   last = path.last();
   log.info "Node found: " + last.getUri();
   if ( last.metas.getValue("year").toInteger() == (it.metas.getValue("year").toIn
   it.metas.lastyearvolume = last.metas.getValue("volume");
   }
 }
}
```

3. Save and apply the configuration.

Step 2 - Rescan the Trades Connector and Check What Is Indexed

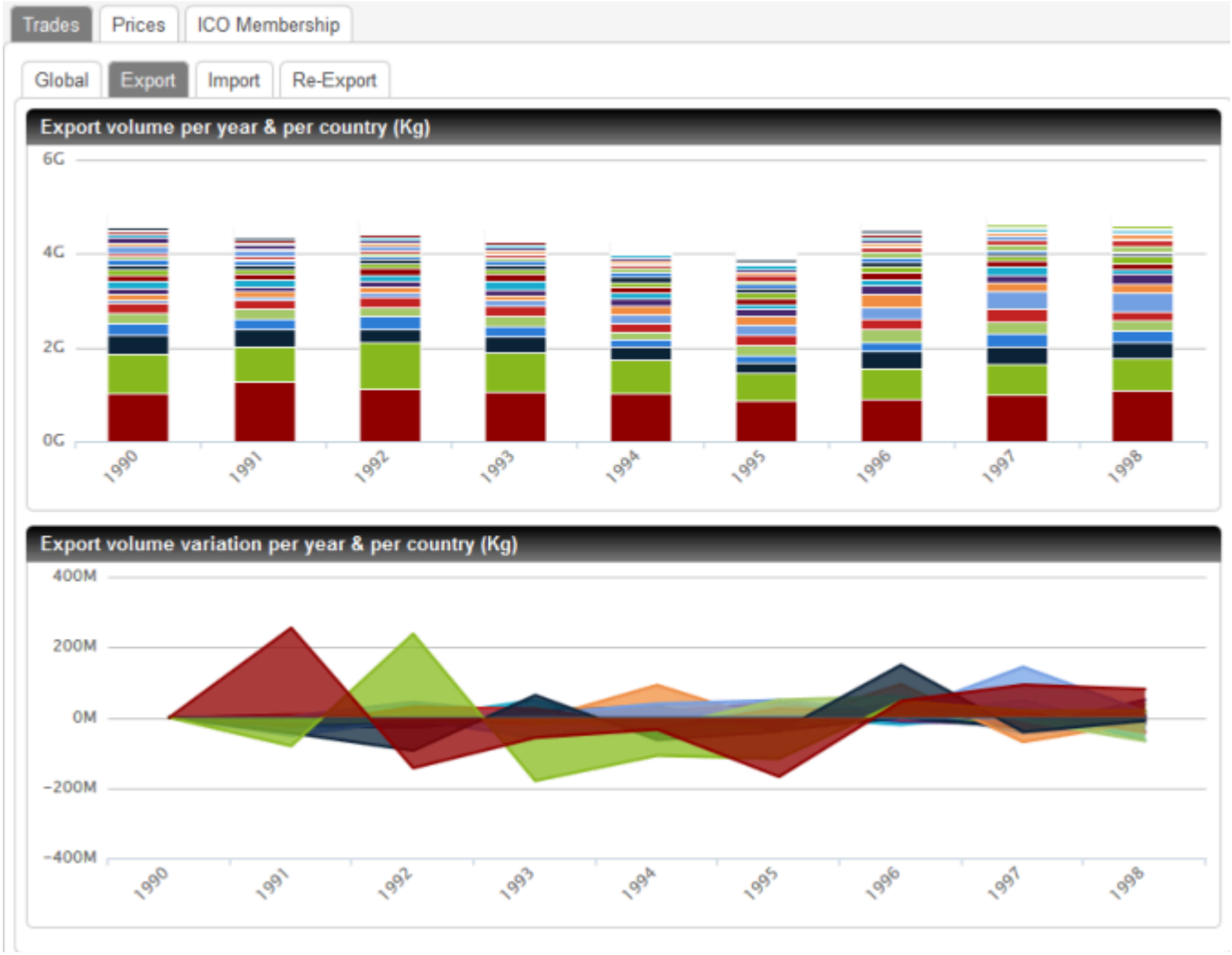
1. Go to the **Home** page and under the connectors list, click **Scan** for the `trades` JDBC connector.
2. Open the Mashup UI application page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/searchcountry_v3`

3. Search for a country, for example, Brazil or France.
4. Click the **see details** link.
5. In the detail page, on the **Trade volume per Year (Kg)** tab, check the **Variation / Year** graph.



6. You can also go to the analytics page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/analytics_v1`

   This page provides various graphics. Choose the **Trades** tab. For each **Export / Import** and **Re-Export** tabs, you can see the trends for each year.



UC-5: Incremental Scan - Propagating Node Changes

One of the most interesting features of the Consolidation Server is the ability to propagate any node change on related views.

In the coffee sample, we provide an extra year of trade values. When adding new trade values, we want to be sure that countries information is updated accordingly (UC-2 and UC-3).

We assume that previous UCs have been completed.

Step 1 - Set the Trades Connector to Incremental Mode

1. In the Administration Console, go to **Connectors** and click the `trades` JDBC connector.
2. In **Query parameters**:
   a. For **Synchronization mode**, select **Query-based incremental synchronization**
   b. For **Checkpoint query**, enter: `select max(year) from trade`
   c. For **Incremental variable**, enter: `YEAR`

d. For **Incremental query**, enter: `select country_id, type, volume, year from trade where year > "$(YEAR)"`

3. Click **Apply**.

Step 2 - Rescan the Trades Connector and Check What Is Indexed

1. Go to the **Home** page and under the connectors list, click **Clear documents** for the `trades` JDBC connector.
2. Once the clear operation is done, click **Scan** for the `trades` JDBC connector.

   Wait for data to be fully indexed.

Check that if you click **Scan** once again for the `trades` JDBC connector, nothing more is pushed to the index.

Step 3 - Add a New Year of Trades

In our example, we are going to add the year 1999 to the coffee database.

For this operation, you need to access the server.

1. Go to the `<INPUTDIR>` containing the coffee sample data.

   You can find the following files: `coffee.db` and `trades_1999.csv`.
2. Import the year 1999 into the coffee database:
   a. In your command-line tool, run `sqlite3 ./coffee.db`
   b. Run the following commands one after the other:

```
.separator ";"
.import trades_1999.csv trade
.exit
```

Step 4 - Rescan the Trades Connector and Check What Is Indexed

1. Go to the **Home** page and under the connectors list, click **Scan** for the `trades` JDBC connector.

   Wait for data to be fully indexed.
2. Open the Mashup UI application: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/searchcountry_v3`
3. Check that countries now have **1999** as last import year.

Results › 1-30 of 191 Sort by

Name	File?	Last import year	Last import volume	Average import volume
India	yes			
Bosnia and Herzegovina	yes	1999	5,061,900.00	3,184,530.00
Swaziland		1999	534,420.00	568,476.00
Guyana				
El Salvador	yes			
Saint Lucia		1999	142,380.00	142,380.00

UC-6: Incremental Scan - Propagating Arc Changes

Another interesting feature of the Consolidation Server is the ability to propagate any arc changes on related views.

We assume that previous UCs have been completed.

Step 1 - Set the Country Connector to Incremental Mode

1. In the Administration Console, go to **Connectors** and click the `country` JDBC connector.
2. In **Query parameters**:
   a. For **Synchronization mode**, select **Query-based incremental synchronization**
   b. For **Initial Query**, enter: `select country_id, ico_status, name, timestamp from countries`
   c. For **Checkpoint query**, enter: `select max(timestamp) from countries`
   d. For **Incremental variable**, enter: `TIMESTAMP`
   e. For **Incremental query**, enter: `select country_id, ico_status, name, timestamp from countries where timestamp > "$(TIMESTAMP)"`
3. Click **Apply**.

Step 2 - Create Organization from Countries

Configure the Transformation Processor

1. Go to **Index > Consolidation**
2. Add a new **transformation** processor:
   a. Select **Groovy** as format
   b. For **Name**, enter `Countries`
   c. Click **Accept**
3. For **Source connector**, select `country`
4. Replace the default code by the following one:

```
// Process all nodes
process("") {
 // Link country documents to the correct organization depending on its membership
 if (it.metas.getValue("ico_status").equals("Member"))
 {
        // create the organization document.
        // This is a managed document, meaning that if no more links are pointing
        // it deletes itself automatically
        organization = createDocument("organization_ICO", "organization")
        organization.metas.org_id="ICO"
        organization.metas.name="International Coffee Organization"
        organization.directives.datamodel_class = "organization"
        // It is required to "yield" created documents explicitly if they should b
        // the aggregation step
        yield organization
        // create the link to the created document
        it.addArcTo("isMemberOf", "organization_ICO");
        } else {
        // create the organization document.
        // This is a managed document, meaning that if no more links are pointing
        // it deletes itself automatically
        organization = createDocument("organization_NONE", "organization")
        organization.metas.org_id="NONE"
        organization.metas.name="not member"
        organization.directives.datamodel_class = "organization"
        // It is required to "yield" created documents explicitly if they should b
        // the aggregation step
        yield organization
        // create the link to the created document
        it.addArcTo("isMemberOf", "organization_NONE");
 }
}
```

Organization documents are generated from countries. If you delete countries, they are deleted too, automatically.

Configure the Aggregation Processor

1. Add an **aggregation** processor:
   a. Select **Groovy** as format
   b. For **Name**, enter `Organization_UC_6`
   c. Click **Accept**
2. Replace the default code by the following one:

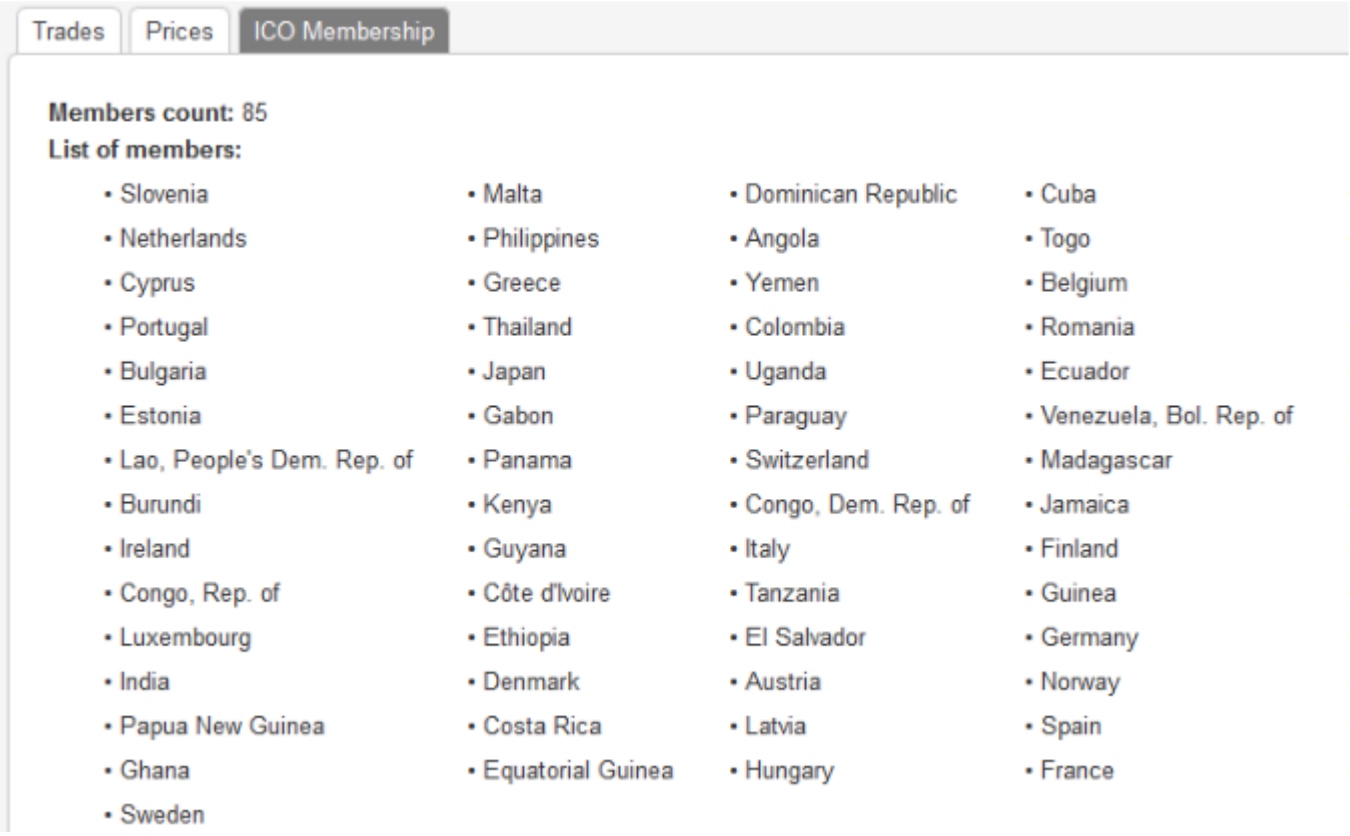
```
// Process nodes having the “organization” type
process("organization") {
 // Log the content of the document passing through this processor
 log.info "Organization: " + it
 // Add all members of Countries to Organization
 it.metas.members +=
 // Get all paths of related country nodes
 match(it, "-isMemberOf[country]") *.last() // fetch last node
 .collect{n-> n.metas.getValue("name") }
 it.metas.number +=
 // Get all paths to related country nodes
 match(it, "-isMemberOf[country]").size();
}
```

3. Save and apply the configuration.

Step 3 - Rescan the Country Connector and Check What Is Indexed

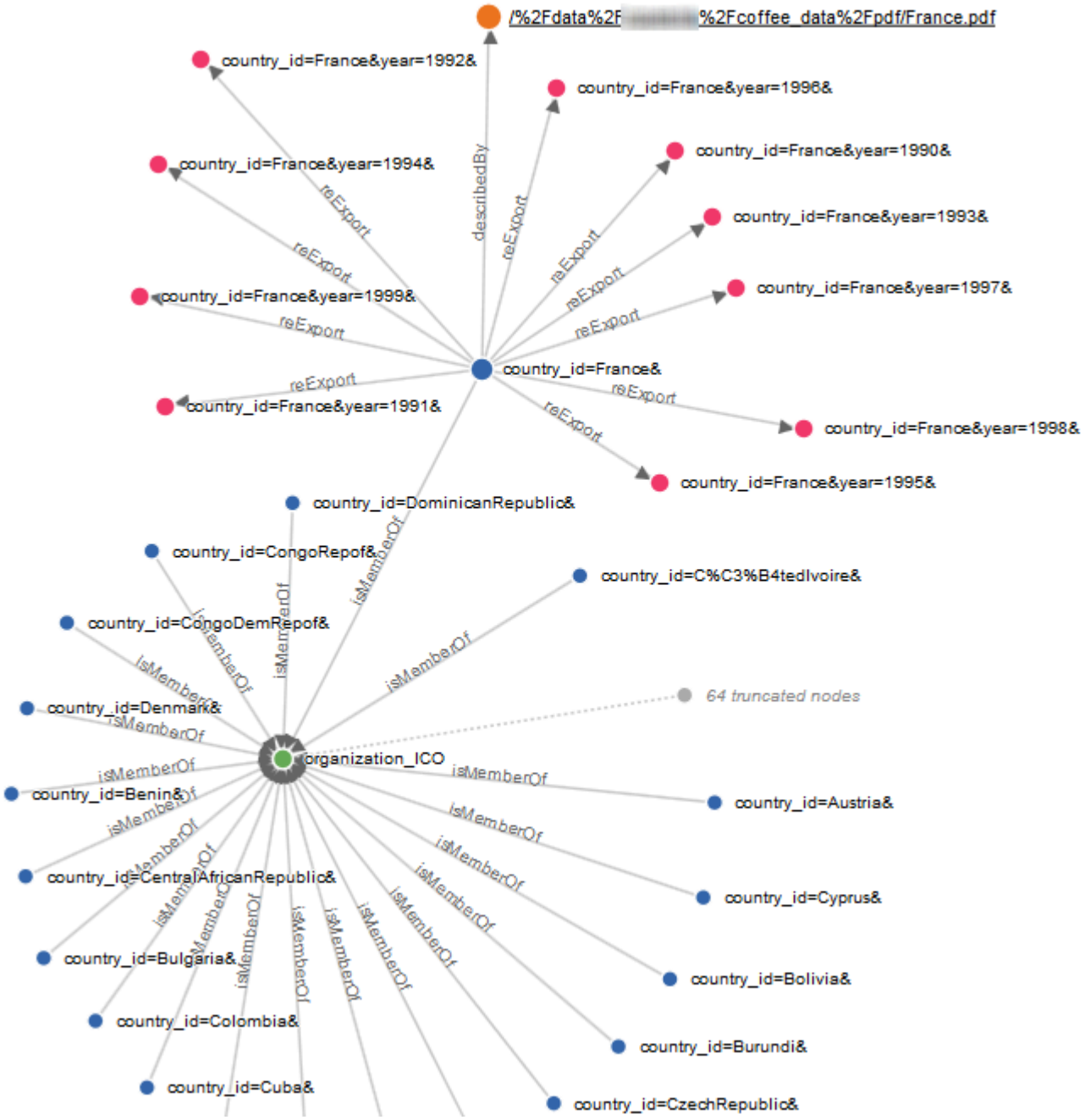
1. Go to the **Home** page and under the connectors list, click **Clear documents** for the `country` JDBC connector.
2. Once the clear operation is done, click **Scan** for the `country` JDBC connector.

   Wait for data to be fully indexed.
3. Check that if you click **Scan** once again for the `country` JDBC connector, nothing more is pushed to the index.
4. Go to the analytics page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/analytics_v1`
5. Select the **ICO Membership** tab. The tab displays the members count and a list of members.



Note: You can also check existing arcs in the **Index > Consolidation > Introspect** tab.

The following graphic shows what we achieved on the object graph at step 3. Arcs (of type `isMemberOf`) are added to a managed document (called `organization_ICO`) linked to countries that are part of the ICO.



Step 4 - Update the Membership of a Country

For this operation, you need to access the server.

1. Go to the `<INPUTDIR>` containing the coffee sample data.
2. Change the membership of a country in the coffee database, for example, Brazil.
   a. In your command-line tool, run `sqlite3 ./coffee.db`
   b. Run the following commands one after the other:

```
delete from countries where country_id="Brazil";
insert into countries(country_id, name, ico_status) values ("Brazil", "Brazil",
.exit
```

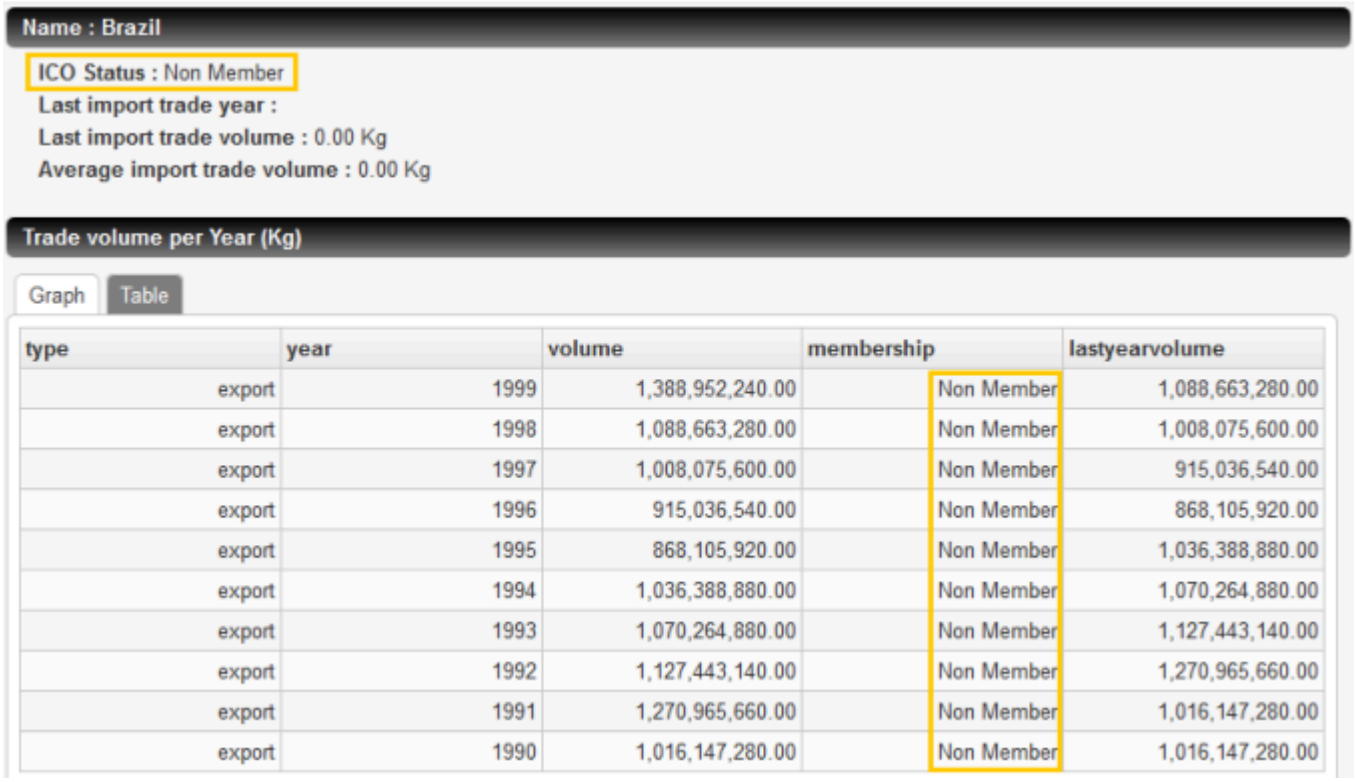
Note: The `insert` statement adds the current timestamp to the record automatically. The JDBC connector uses it to detect this modification.

Step 5 - Rescan the Country Connector and Check What Is Indexed

1. Go to the **Home** page and under the connectors list, click **Scan** for the `country` JDBC connector.

   Wait for data to be fully indexed.

2. Open the Mashup UI application: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/searchcountry_v3`
3. Search for **Brazil**, and click its **see details** link.
   - In the detail page, the **ICO Status** is **Non Member**.
   - If you select the **Trade volume per Year (Kg) > Table** tab, every trade now has its membership updated to **Non Member**.



type	year	volume	membership	lastyearvolume
export	1999	1,388,952,240.00	Non Member	1,088,663,280.00
export	1998	1,088,663,280.00	Non Member	1,008,075,600.00
export	1997	1,008,075,600.00	Non Member	915,036,540.00
export	1996	915,036,540.00	Non Member	868,105,920.00
export	1995	868,105,920.00	Non Member	1,036,388,880.00
export	1994	1,036,388,880.00	Non Member	1,070,264,880.00
export	1993	1,070,264,880.00	Non Member	1,127,443,140.00
export	1992	1,127,443,140.00	Non Member	1,270,965,660.00
export	1991	1,270,965,660.00	Non Member	1,016,147,280.00
export	1990	1,016,147,280.00	Non Member	1,016,147,280.00

4. Go to the analytics page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/analytics_v1`
5. Choose the **ICO Membership** tab.

Brazil is not present in the list anymore.

UC-7: Generating Child Documents

When flattening data, it is sometimes useful to be able to generate multiple documents from a parent document. These child documents are not pushed by any source but are interesting to simplify queries performed later on the index.

We assume that previous UCs have been completed.

Step 1 - Create Child Documents from Organization with an Aggregation Processor

1. Add an **aggregation** processor:
   a. Select **Groovy** as format
   b. For **Name**, enter `Organization_UC_7`
   c. Click **Accept**
2. Replace the default code by the following one:

```
// Process nodes having the “organization” type
process("organization") {
 // Log the content of the document passing through this processor
 log.info "Child creation for organization: " + it
 // Find the top country for import trade per year
 trades =
 // Get all paths to related country nodes
 match(it, "-isMemberOf[country].import[trade]") *.last()  // fetch last node
 year_top = [:].withDefault() { [:].withDefault() {0} }
 // Big Integer
 def bInt = 0G;
 year_top_volume = [:].withDefault() { bInt }
 // Build the child collection
 trades.each {
 trade -> if (year_top[trade.metas.getValue("year")]["volume"] < trade.metas.getVa
.toInteger()) {
 year_top[trade.metas.getValue("year")]["volume"] = trade.metas.getValue("volume")
 year_top[trade.metas.getValue("year")]["country"] = trade.metas.getValue("country
 year_top_volume[trade.metas.getValue("year")] += trade.metas.getValue("volume").t
   }
 }
 // Caution! Before pushing any new document, remove existing child documents, if
 // This operation is yielded automatically.
 deleteDocumentChildren(it, "/year_import/");
 // create child documents
 year_top.each { key, value ->
      log.info "Year:" + key + " - " +
```

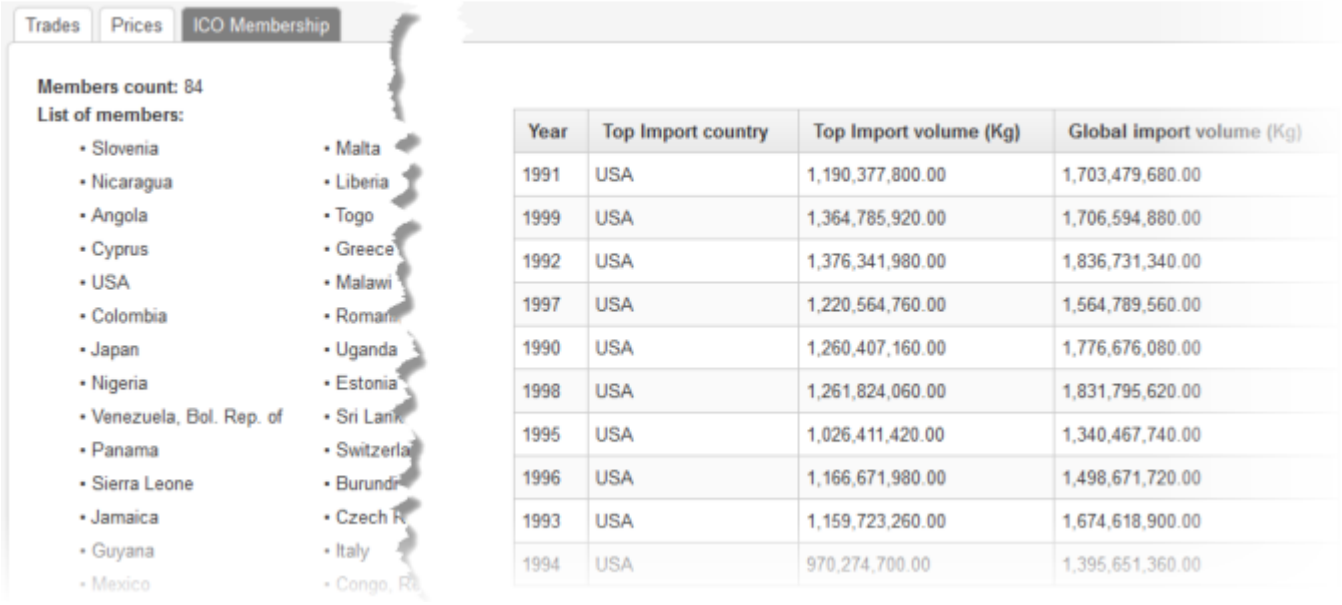
```
          value["volume"] + " - " + value["country"] +
          " - " + year_top_volume[key] ;
      child = createChildDocument(
            it, // root
            '/year_import/' + key, // child URI
            "ico_trade" // type
      );
      // Add metas to the child document
     child.metas.parent_identifier = it.getUri();
     child.directives.datamodel_class = "ico_trade";
     // directly set with the type defined in createDocument but it can be overrid
     child.metas.org_id = it.metas["org_id"];
     child.metas.year = key;
     child.metas.country_id = value["country"];
      child.metas.volume = value["volume"];
      child.metas.globalvolume = year_top_volume[key];
      yield child;
 }
}
```

3. Save and apply the configuration.

Step 2 - Relaunch the Organization Aggregation and Check What Is Indexed

1. Go to the **Home** page.
2. Click **Force aggregation** and enter `organization` as type.
3. Open the following Mashup UI application page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/analytics_v2`
4. Select the **ICO Membership** tab.

You can now see the **Top Import country** and **Global import volume** for each year.



Step 3 - Change the Membership of a Country

For this operation, you need to access the server.

1. Go to the `<INPUTDIR>` containing the coffee sample data.
2. Change the membership of a country in the coffee database, for example, USA.
   a. In your command-line tool, run `sqlite3 ./coffee.db`
   b. Run the following commands one after the other:

```
delete from countries where country_id="USA";
insert into countries(country_id, name, ico_status) values ("USA", "USA", "Non M
.exit
```

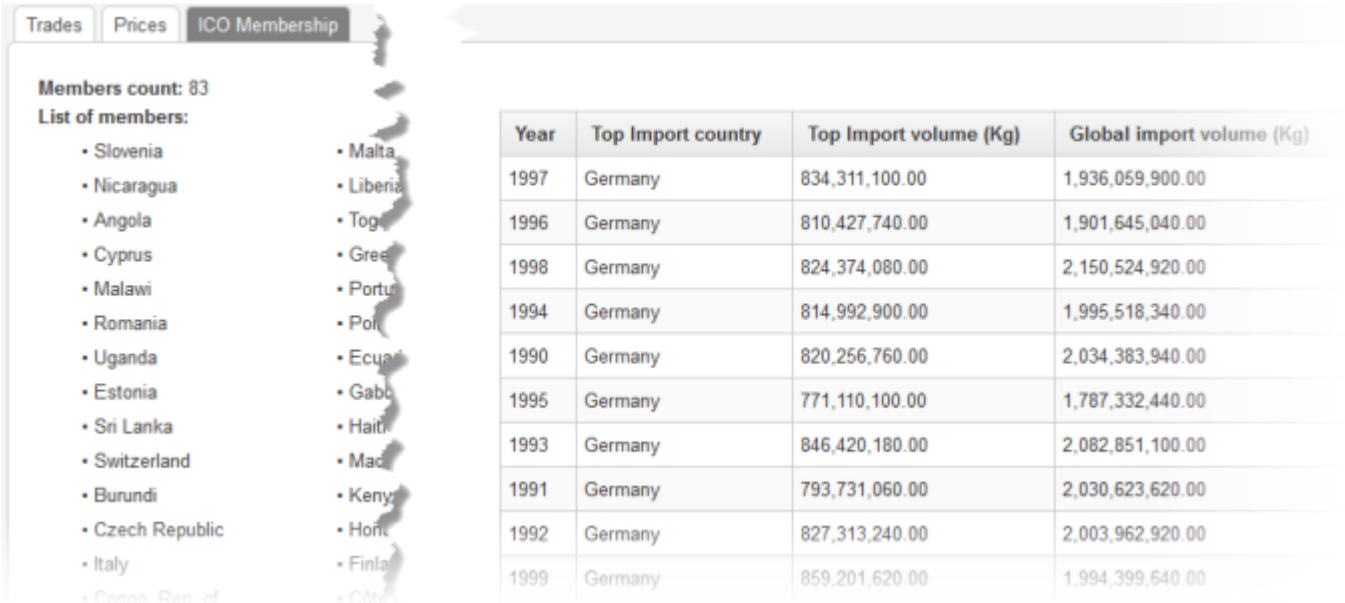
Note: The `insert` statement adds the current timestamp to the record automatically. The JDBC connector uses it to detect this modification.

Step 4 - Rescan the Country Connector and Check What Is Indexed

1. Click **Scan** for the `country` JDBC connector.

   Wait for data to be fully indexed.
2. Go to the analytics page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/analytics_v2`
3. Select the **ICO Membership** tab.

USA is not displayed in the **Top Import country** column anymore.



Year	Top Import country	Top Import volume (Kg)	Global import volume (Kg)
1997	Germany	834,311,100.00	1,936,059,900.00
1996	Germany	810,427,740.00	1,901,645,040.00
1998	Germany	824,374,080.00	2,150,524,920.00
1994	Germany	814,992,900.00	1,995,518,340.00
1990	Germany	820,256,760.00	2,034,383,940.00
1995	Germany	771,110,100.00	1,787,332,440.00
1993	Germany	846,420,180.00	2,082,851,100.00
1991	Germany	793,731,060.00	2,030,623,620.00
1992	Germany	827,313,240.00	2,003,962,920.00
1999	Germany	859,201,620.00	1,994,399,640.00

UC-8: Consolidating Data from Storage Service

It is interesting to combine the information coming from Exalead CloudView features like tagging or comments, relying on the Storage Service, with original data to search or refine on new values.

In this use case, we want to index the tags defined on `country` documents (that is, `storageKey_tags`) and use them as new facets.

- We assume that previous UCs have been completed.
- The Storage Service is activated. For more information, see "Configuring Data Storage for Collaborative Widgets" in the Exalead CloudView Mashup Builder User's Guide.
- The `RepushFromCache` setting must be set to `false` in the `<DATADIR>/config/360/StorageService.xml` file (default configuration).

Step 1 - Define the Source Connector for StorageService

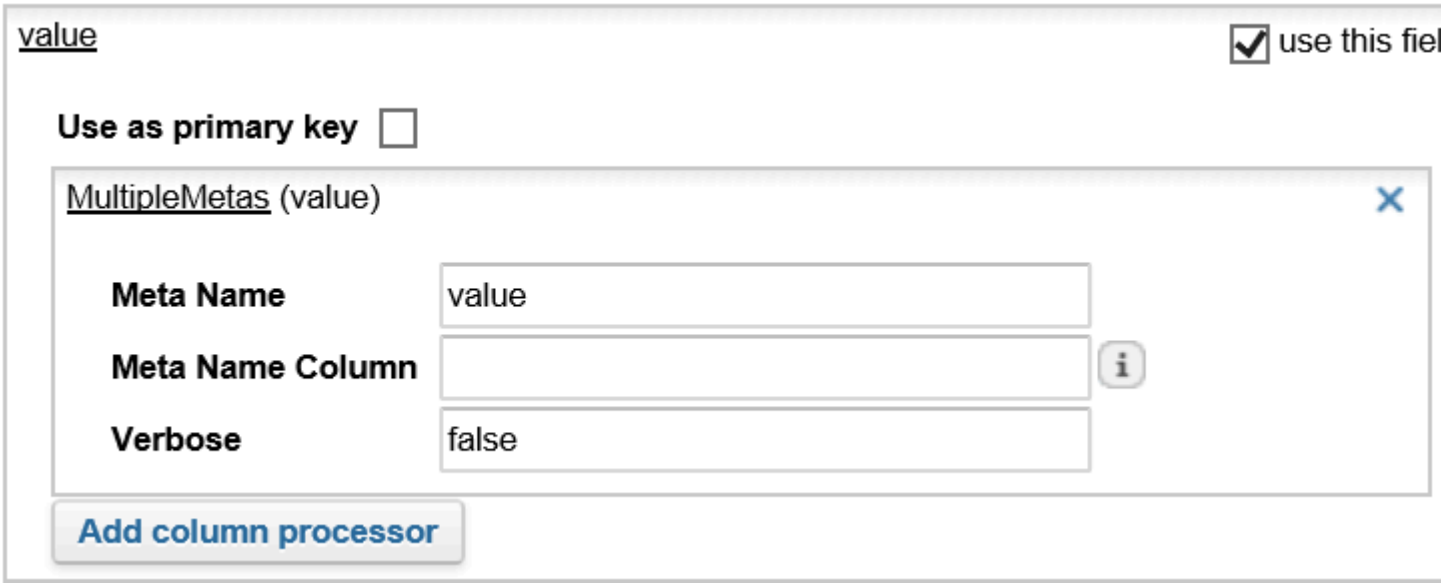
1. In the Administration Console, go to **Index > Connectors** and click **Add connector**.
   a. In **Name**, enter `storageService`.
   b. For **Type**, select the **JDBC** connector.
   c. For **Push to PAPI server**, select the `Consolidation server cbx0` instance.
   d. Click **Accept**.
2. For **Store documents in data model class**, enter `storageValue`.

   Note: This class is not present in the data model yet. It is only used by the Consolidation Server.

3. In **Connection parameters**:
   a. For **Driver**, enter `org.sqlite.JDBC`
   b. For **Connection string**, enter `jdbc:sqlite://<DATADIR>/storageService/storage.db.sqlite`
   c. Click **Test connection**. The database connector automatically connects to the database.
4. In **Query parameters**:
   a. For **Synchronization mode**, select **Query-based incremental synchronization**
   b. For **Initial query**, enter: `select ikey, ukey, value, res_type, res_id, modified_date, source, app_id, build_group from cv360_storage_service`
   c. For **All URI Query**, enter: `select ikey, ukey from cv360_storage_service`

d. For **Checkpoint query**, enter: `select max(modified_date) from cv360_storage_service`
e. For **Incremental variable**, enter: `TS`
f. For **Incremental query**, enter: `select ikey, ukey, value, res_type, res_id, modified_date, source, app_id, build_group from cv360_storage_service where modified_date > '$(TS)'`

5. Click **Retrieve fields**.
6. Define the `ukey` and `ikey` fields as primary keys.
   a. Click the `ukey` field to expand it.
   b. Select **Use as primary key**.
   c. Repeat the operation for the `ikey` field.
7. For the `value` field:
   a. Delete the **automatic** processor.
   b. Click **Add column processor** and add a **MultipleMetas** processor.
   c. For **Meta Name**, enter `value`.



8. Click **Apply**.

Step 2 - Link storageService Tags to Countries

Configure the Transformation Processor

Important: For this Use Case step, we are going to use a Java processor delivered by default.

1. Go to **Index > Consolidation**
2. Add a new **transformation** processor:

a. Select **Java** as format.
b. For **Name**, enter `StorageService`.
c. For **Processor**, select **Storage Service Key Linker Processor**.
d. Click **Accept**.

3. For **Source connector**, select `storageService`
4. Click **Save**.

With this processor, we have achieved to link `storageService` tags to country documents.

Note: To get the same result with Groovy code, replace the default code by the following one:

```
// Process all nodes
process("") {
 // Logs the content of the document passing through this processor
 log.info "Received document: " + it
 // Adding parent type
 it.setType("storageValue", "storage");
 // Create the virtual object depending on the name used for storing tag values
 // URI = <key store name (tags[] for example)> + delimiter + link object key (res_id
 // Type: "storage_" + key store name without [] (tags[] for example)
 keystoreObject = createDocument( 'storageKey_' + it.metas.getValue("ikey") + "-#-"
+ it.metas.getValue("res_id"), // keystore URI
 "storageKey_" + it.metas.getValue("ikey")[0..-3], "storage" // type
 );
 // Add key name to the document (for debugging purpose only)
 // remove last 2 characters:[]
 keystoreObject.metas.name = it.metas.getValue("ikey")[0..-3]
        yield keystoreObject;
 // Add link from keystore value to keystore object
 it.addArcFrom('hasForValue', 'storageKey_' + it.metas.getValue("ikey") + "-#-"
+ it.metas.getValue("res_id"));
 // Add link from keystore object to linked object
 keystoreObject.addArcFrom('hasStorageKey', it.metas.getValue("res_id"));
}
```

Configure the Aggregation Processors

Important: For this Use Case step, we are going to use Java processors delivered by default.

1. Add an **aggregation** processor:
   a. Select **Java** as format.
   b. For **Name**, enter `Countries_UC_8`.
   c. For **Processor**, select **Storage Service Key Flattener Processor**.
   d. Click **Accept**.

2. Configure the processor as follows:
   a. For **Processed Document type**, enter `country`.
   b. For **Key store document type**, enter `storageKey_tags`.
   c. For **Target meta**, enter `tags`.

   Note: To get the same result with Groovy code, replace the default code by the following one:

```
// Process nodes having the “country” type
process("country") {
    // Add "tags[]" keystore values on countries
    // by matching on nodes with the type [storageKey_tags]
    // For other keys, use [storageKey_<whatever>]
     for (node in (match(it, "hasStorageKey[storageKey_tags].hasForValue[storageVa
     log.info "keystore value found : " + node.metas.getValues("value");
              it.metas.tags.addAll(node.metas.getValues("value"))
  }
       log.info "country after tags : " + it
}
```

3. Add another **aggregation** processor to discard storage nodes:
   a. Select **Java** as format.
   b. For **Name**, enter `Storage_UC_8`.
   c. For **Processor**, select **Discard**.
   d. Click **Accept**.
4. For **Discard document types**, click **Add item** and enter `storage`.

   Note: To get the same result with Groovy code, replace the default code by the following one:

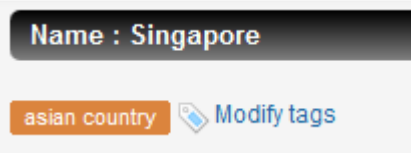
```
process("storage") {
    log.info "discard for : " + it
    // discard storage nodes
    discard()
}
```

5. Save and apply the configuration.

Step 3 - Add Tags to Countries

1. Open the following Mashup UI application page: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/searchcountry_v4`
2. Search for a country, for example, **Singapore** .
3. Click the **see details** link.
4. Click the **Tag this country** link to add a tag to the selected country. For example, for Singapore, enter `asian country` and press `ENTER`.

asian country displays as tag.



5. Perform the 3 previous steps to tag **Japan** as `asian country` too.

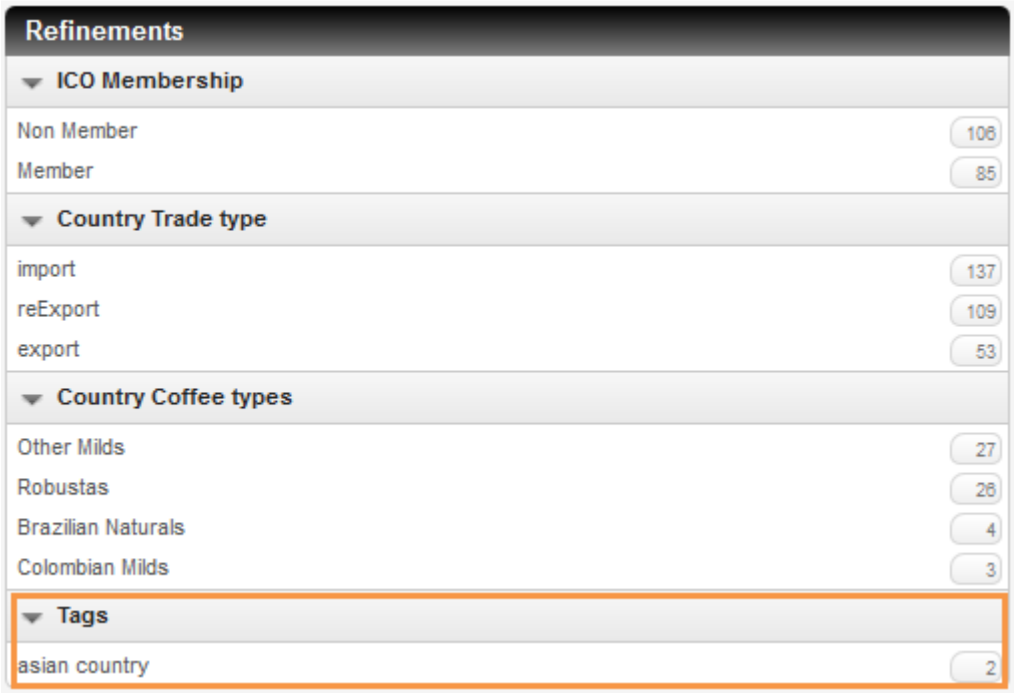
Step 4 - Index Tags

1. Click **Scan** for the `storageService` JDBC connector and wait for data to be fully indexed.

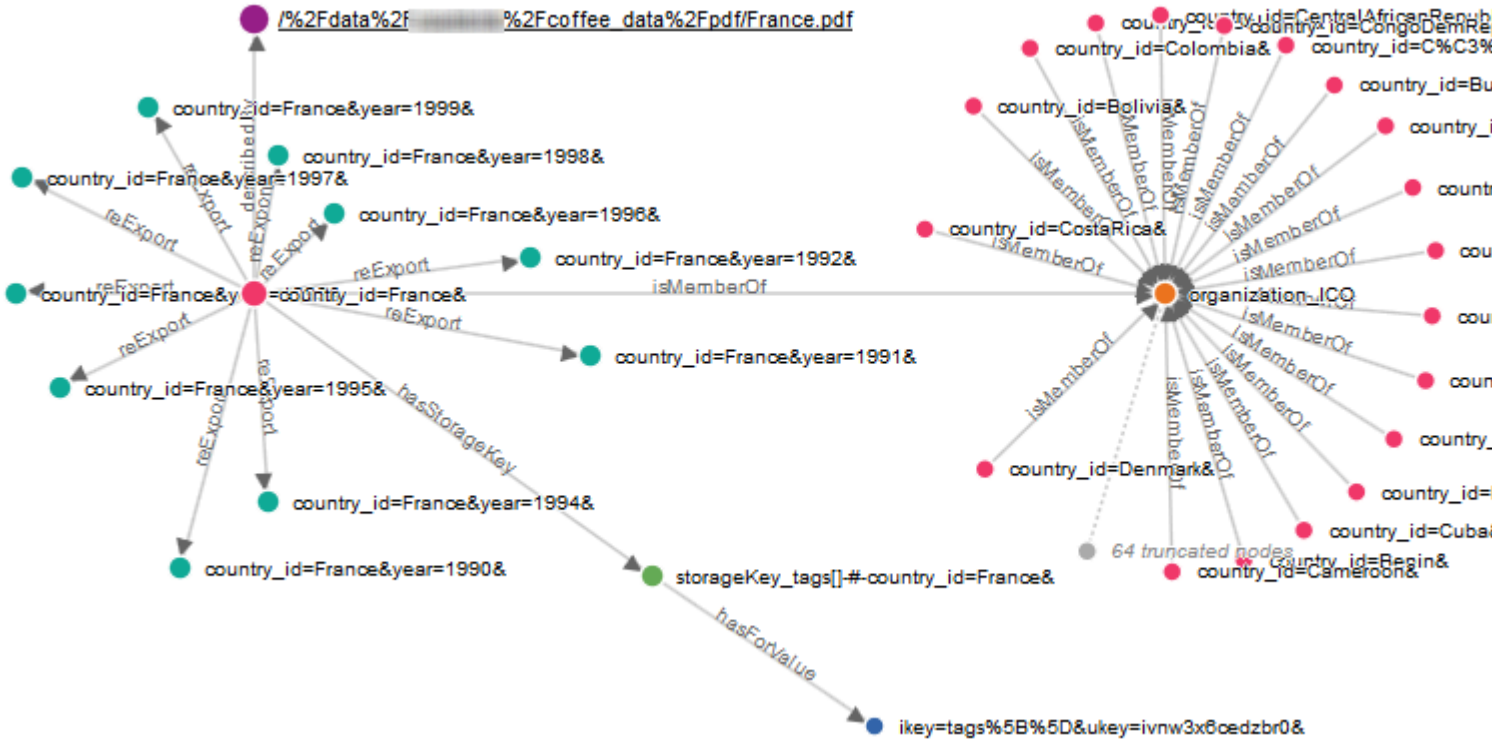
   Two documents are indexed for the `storageService` connector.

2. Open the Mashup UI application again: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/page/searchcountry_v4`

You can now see a new **Tags** facet in the **Refinements** panel, displaying the values entered for the tagged documents.



The following graphic shows what we achieved on the object graph (**Max. arcs per node** has been set to **10** for more readability).


/%2Fdata%2F%2Fcoffee_data%2Fpdf/France.pdf
country_id=France&year=1999&
country_id=France&year=1998&
country_id=France&year=1997&
country_id=France&year=1996&
country_id=France&year=1992&
country_id=France&year=1991&
country_id=France&year=1995&
country_id=France&year=1994&
country_id=France&year=1990&
reExport
isMemberOf
hasStorageKey
storageKey_tags[]-#-country_id=France&
hasForValue
ikey=tags%5B%5D&ukey=ivnw3x6oedzbr0&
organization_ICO
country_id=Colombia&
country_id=Bolivia&
country_id=CostaRica&
country_id=Denmark&
country_id=Cameroon&
country_id=Cuba
64 truncated nodes

Appendix - Groovy Processors

A Groovy processor is a piece of Groovy code defined with a Closure named `process`, taking one constant string (and one only) as parameter.

The string value has two possible interpretations:

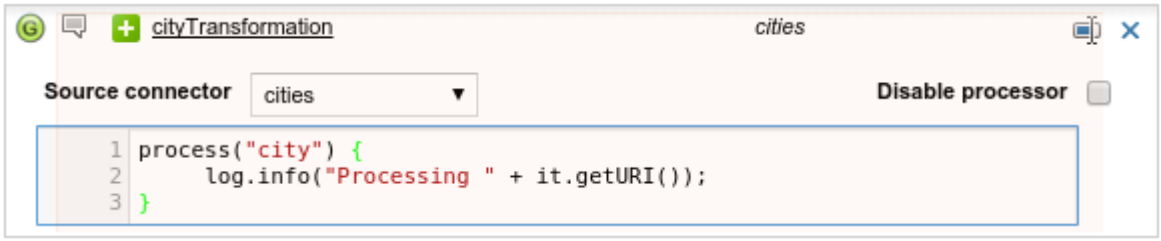
- If empty, it means that the processor is executed on all document types pushed to the Consolidation Server.
- If non-empty, then the processor is executed by checking if the type provided belongs to the document type inheritance. See Processor Type Inheritance and Runtime Selection.

Recommendation: Read the Groovy documentation.

The similar behavior is achieved in Java with `IJavaAggregationProcessor.getAggregationDocumentType()` and `IJavaTransformationProcessor.getTransformationDocumentType()`.

```
process("city") {
  log.info("Processing " + it.getUri());
}
```

The code above is equivalent to Java Example 1. You do not find for which source it is associated to, because it is defined in the Administration Console as shown below.



Although it is not explicitly visible in the method signature, the `process` method receives the current document being processed (transformation or aggregation) using the special `it` variable. You can see it in the above example with `it.getUri()`, which is the equivalent of `IConsolidationDocument.getUri()`.

Groovy Transformation and Aggregation Operations

The Java interfaces defining the allowed operations for Transformation and Aggregation are shared with the Groovy language.

As a result, all the operations present in Java are also available in Groovy. Specific shortcuts are however available in Groovy only:

- You can access all getters directly without specifying a method call and the `get` prefix. For example, you can rewrite `it.getUri()` as `it.uri`.
- You can also access the following properties with similar shorthands:
  - `it.metas`: The document metadata. For example, `it.metas.company_name` returns a Groovy list of strings containing the meta values for the `company_name` meta. You can also specify the meta name between quotes. So you could write `it.metas."company_name"`. It is even more interesting to make it dynamic by writing `it.metas."$myVar"`, where the variable `''myVar''` would be defined with the assignment `myVar = "company_name"`.
  - `it.directives`: The document directives. Usage is similar to `it.metas`.
  - `it.parts`: The document parts. Usage is similar to `it.metas` except that values are now instances of `IDocumentPart` as in Java.

Company's Hierarchy Example in Groovy

Let us see how you can implement the Connect Employees to Services and Services to Companies in Groovy.

```
process("employee") {
    def addService = { serviceName, companyName ->
        serviceDoc = createDocument("service=" + serviceName + "&", "service")
        serviceDoc.directives.datamodel_class = "service"
        serviceDoc.addArcTo("service", "company=" + companyName + "&")
        yield serviceDoc
        serviceDoc //return object
    }
    if (it.metas.company_name && it.metas.service_name) {
      serviceDoc = addService(it.metas.service_name[0], it.metas.company_name[0])
      it.addArcTo("employee", serviceDoc.getUri())
    }
}
```

For Count the Number of Employees and Push Updated Documents, a possible implementation could be:

```
process("company") {
    it.metas.nb_employees = match(it, "-service.-employee").size();
}
```

Discard Processor Code Samples

DiscardAggregationProcessor.java

```
package com.exalead.samples.consolidation;
import com.exalead.cloudview.consolidationapi.processors.IAggregationDocument;
import com.exalead.cloudview.consolidationapi.processors.java.IJavaAllUpdatesAggregat
import com.exalead.cloudview.consolidationapi.processors.java.IJavaAllUpdatesAggregat
import com.exalead.mercury.component.config.CVComponentConfigClass;
@CVComponentConfigClass(configClass = DiscardAggregationProcessorConfig.class, config
DiscardAggregationProcessorConfigCheck.class)
public class DiscardAggregationProcessor implements IJavaAllUpdatesAggregationProcess
    private final String[] discardedDocumentTypes;
    public DiscardAggregationProcessor(final DiscardAggregationProcessorConfig config
        final String[] configDocumentTypes = config.getDocumentTypes();
        if (configDocumentTypes != null) {
            this.discardedDocumentTypes = new String[configDocumentTypes.length];
            for (int i = 0; i < configDocumentTypes.length; i++) {
                this.discardedDocumentTypes[i] = configDocumentTypes[i].trim();
            }
        } else {
            this.discardedDocumentTypes = null;
        }
    }
    @Override
    public String getAggregationDocumentType() {
        return null;
    }
    @Override
    public void process(IJavaAllUpdatesAggregationHandler handler, IAggregationDocume
Exception {
        if (this.discardedDocumentTypes != null) {
            for (int i = 0; i < this.discardedDocumentTypes.length; i++) {
                if (document.getTypeInheritance().contains(this.discardedDocumentType
                    handler.discard();
                }
            }
        }
    }
}
```

DiscardAggregationProcessorConfig.java

```
package com.exalead.samples.consolidation;
import com.exalead.config.bean.IsMandatory;
```

```
import com.exalead.config.bean.PropertyDescription;
import com.exalead.config.bean.PropertyLabel;
import com.exalead.mercury.component.config.CVComponentConfig;
public class DiscardAggregationProcessorConfig implements CVComponentConfig {
        public final static String[] METHODS = {
            "DocumentTypes"
    };
    public static final String[] getMethods() {
        return METHODS;
    }
    private String[] documentTypes;
        public DiscardAggregationProcessorConfig() {
    }
    @IsMandatory(true)
    @PropertyLabel("Discard document types")
    @PropertyDescription("Specifies types of documents to be discarded")
    public void setDocumentTypes(String[] documentTypes) {
        this.documentTypes = documentTypes;
    }
    public String[] getDocumentTypes() {
        return this.documentTypes;
    }
}
```

DiscardAggregationProcessorConfigCheck.java

```
package com.exalead.samples.consolidation;
import com.exalead.config.bean.ConfigurationException;
import com.exalead.mercury.component.config.CVComponentConfigCheck;
public class DiscardAggregationProcessorConfigCheck implements
 CVComponentConfigCheck<DiscardAggregationProcessorConfig> {
    @Override
    public void check(final DiscardAggregationProcessorConfig config, final boolean u
ConfigurationException, Exception {
        if (config != null) {
            final String[] documentTypes = config.getDocumentTypes();
            if (documentTypes != null && documentTypes.length == 0) {
                final ConfigurationException error = new ConfigurationException
                  ("Discard aggregation processor: 'documentTypes' property can't be
                error.setConfigKey("documentTypes");
                throw error;
            }
            for (String documentType : documentTypes) {
                final String trimmedDocumentType = documentType.trim();
                if (trimmedDocumentType.isEmpty()) {
                    final ConfigurationException error = new ConfigurationException
                      ("Discard aggregation processor: empty 'documentTypes' entry");
                    error.setConfigKey("documentTypes");
```

```
                        throw error;
                    }
                }
            }
        }
}
```

Appendix - Matching Expressions Grammar

Define object graph matching expressions with the following grammar.

Element	Syntax
paths	`path { "." path }`
path	`("(" paths ")" { quantifier } { ("|" | "|>") path }) | edge`
edge	`{ "-" } string { "[" nodeTypes "]" } { quantifier } { ("|" | "|>") edge }`
nodeTypes	`(regex | string) { "|" nodeTypes }`
meta	`[regex|string]{'meta'|'meta2'}` **Note:** For more information, see Impact Detection.
quantifier	`("*" | "?" | "+") { "*" }`
regex	`"/" regex_string "/"`

Protect Specific Characters from Interpretation

Theoretically, it is possible that your documents URIs contain some characters that may enter in conflict with the characters used in the grammar described. In such case, to avoid a parsing exception, you can protect these special characters using the simple quote character to protect your graph matching expression.

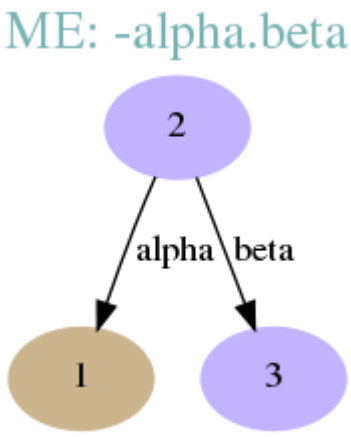
So the expression: `-node['xd/df-ty/x.b*'].'a|b'` is equivalent to the expression: `-node[X].Y`

Examples

In the following examples the starting node is highlighted in maroon, and the matching nodes are highlighted in purple. Resulting paths are listed afterward.

Note: Using the minus sign – before the name of an arc reverses its direction.

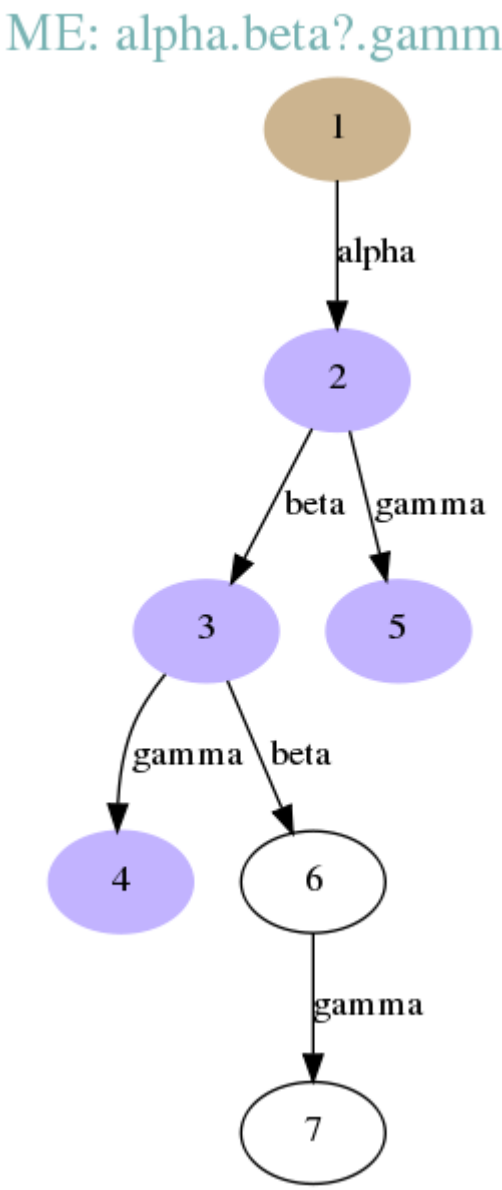
Case Involving a Simple Path



Resulting path:

- 2 -> 3

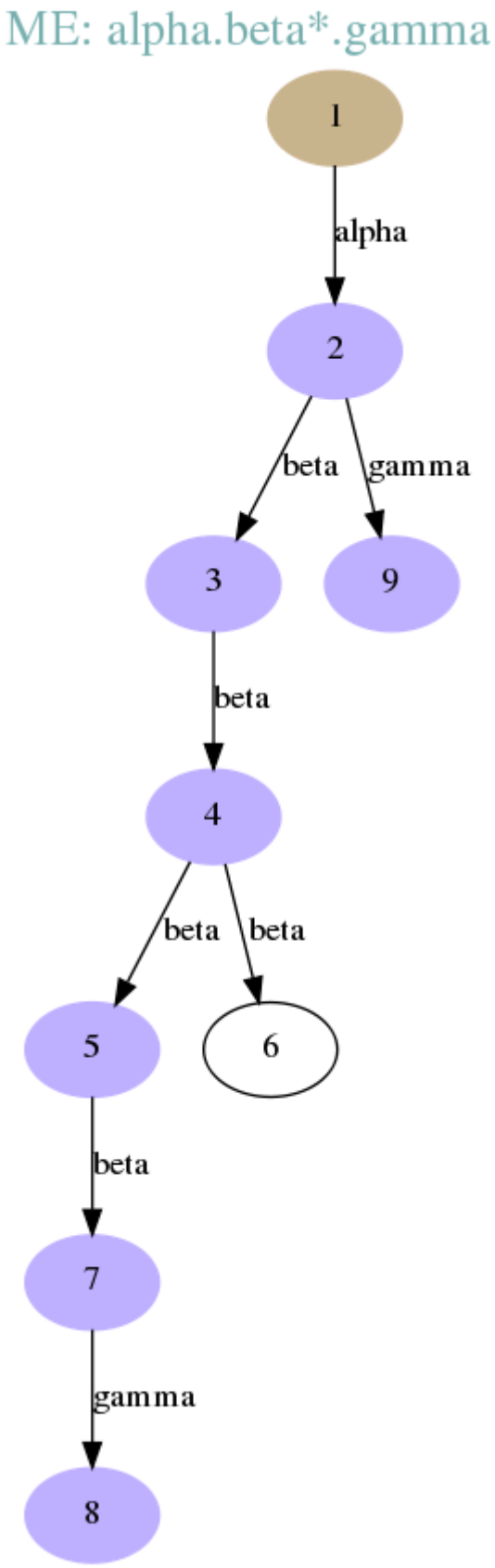
Case with The "?" Operator



Resulting path:

- 2 -> 3 -> 4
- 2 -> 5

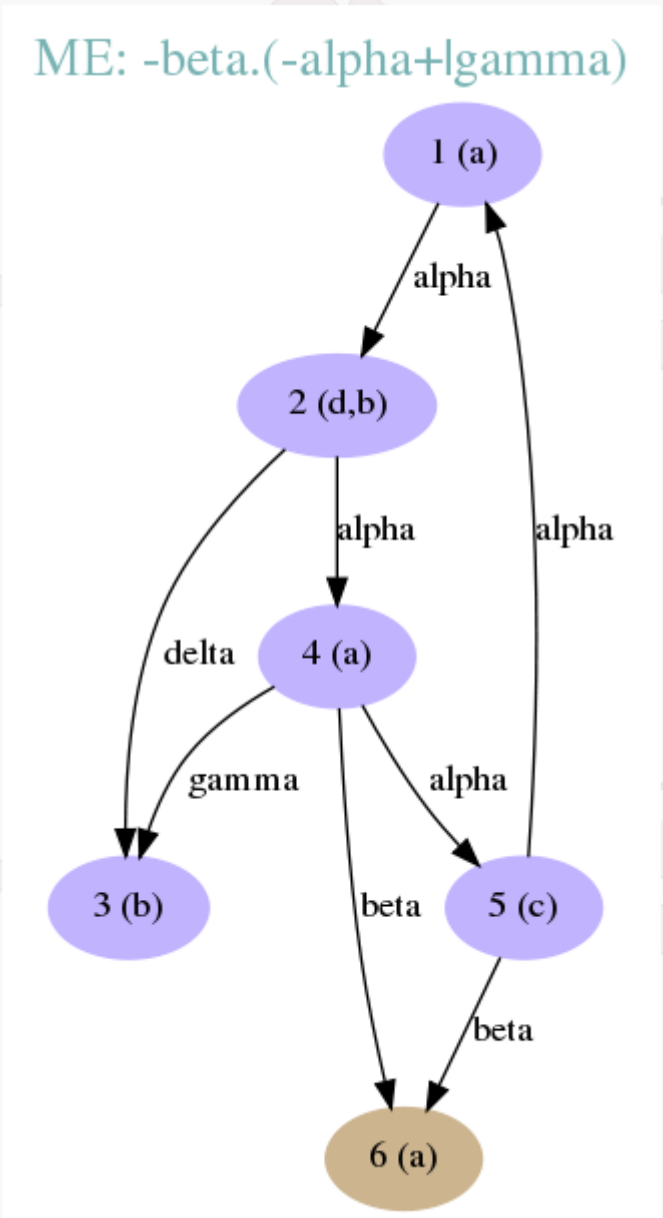
Case Involving a Star



Resulting paths:

- 2 -> 3 -> 4 -> 5 -> 7 -> 8
- 2 -> 9

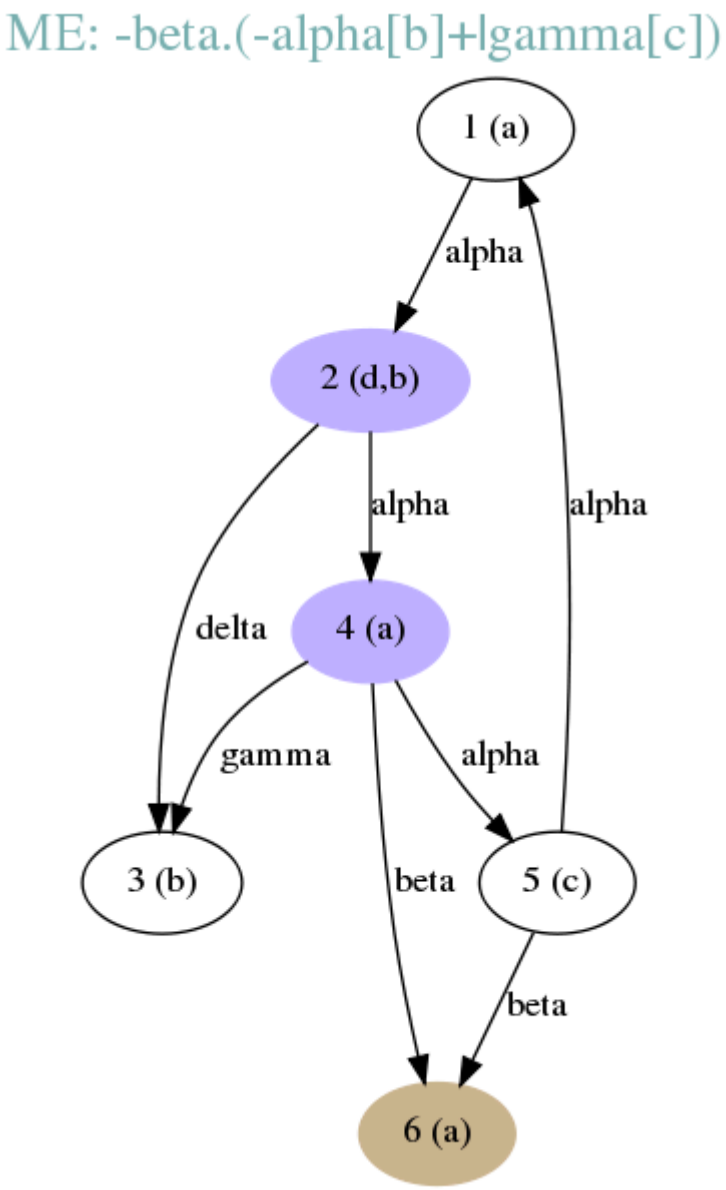
Case with an OR on an Arc



Resulting paths:

- 4 -> 3
- 4 -> 2 -> 1 -> 5
- 5 -> 4 -> 2 -> 1

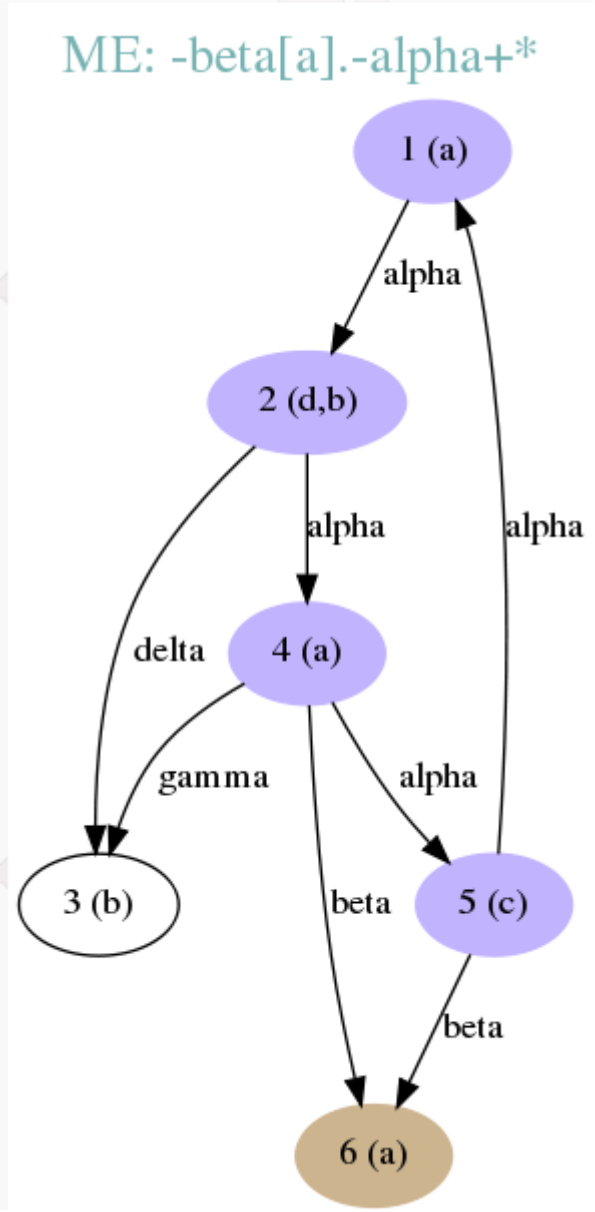
Case with an OR on a Path Element



Resulting path:

- 4 -> 2

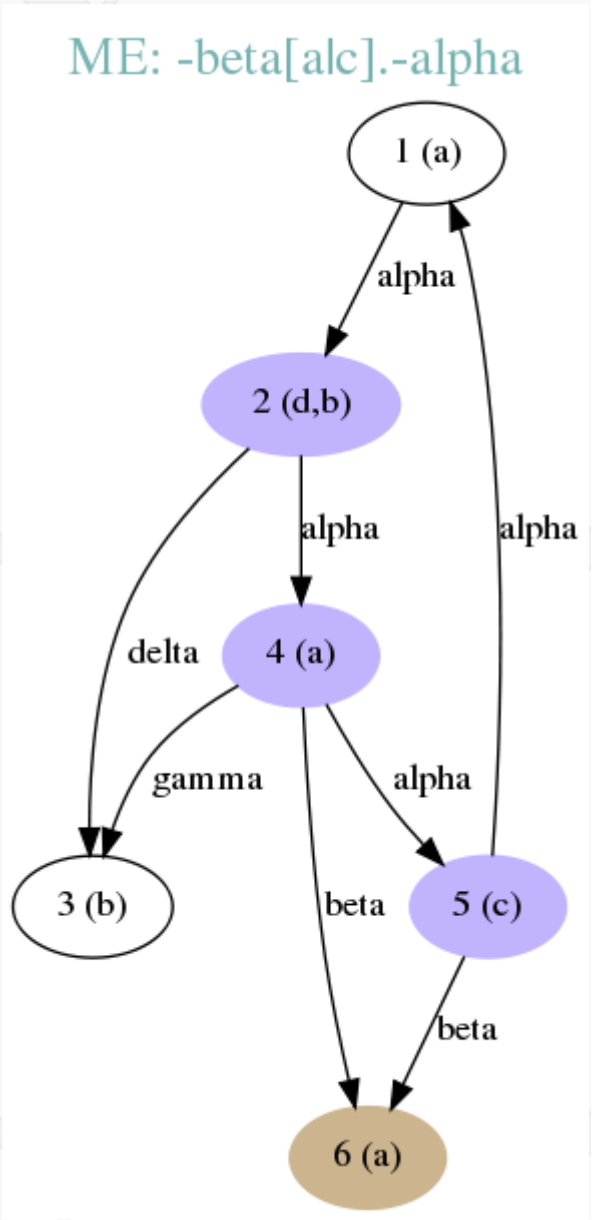
Case with a Closure Operator



Resulting paths:

- 4 -> 2
- 4 -> 2 -> 1
- 4 -> 2 -> 1 -> 5

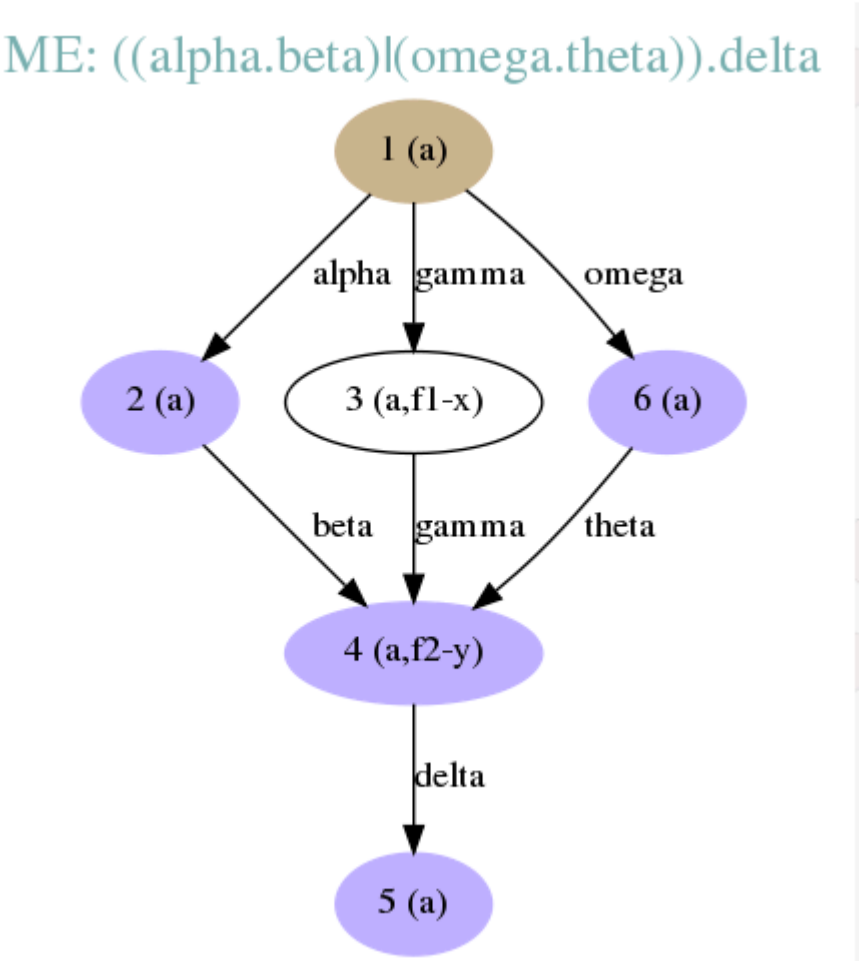
Case with an OR Operator for Node Type



Resulting paths:

- 4 -> 2
- 5 -> 4

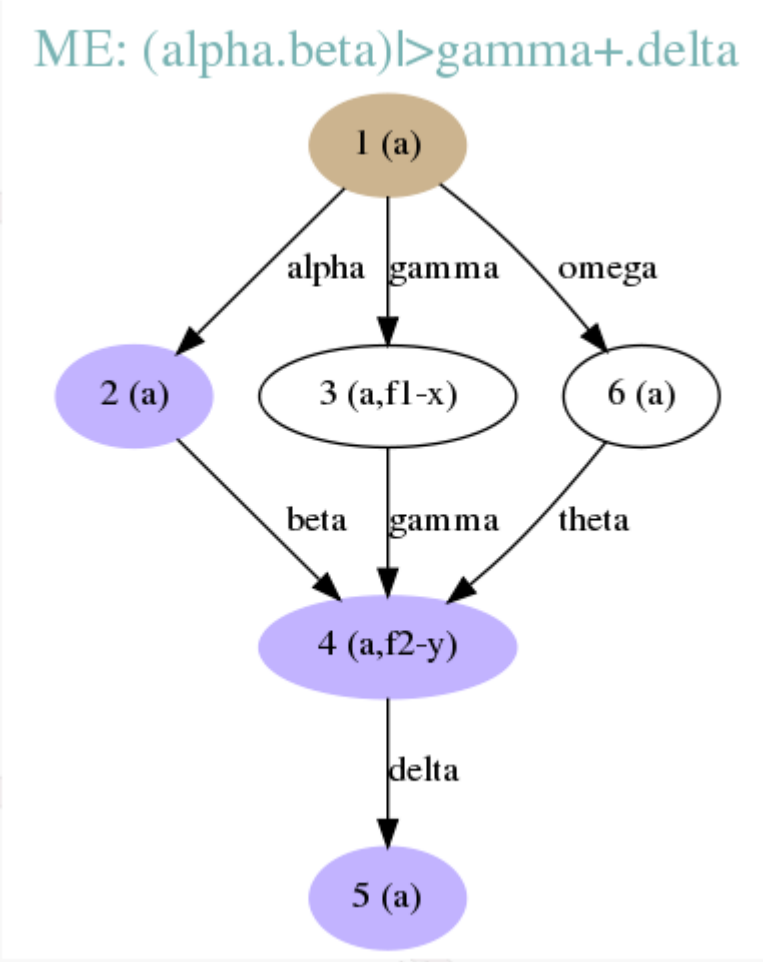
Case with an OR Operator on Path



Resulting paths:

- 2 -> 4 -> 5
- 6 -> 4 -> 5

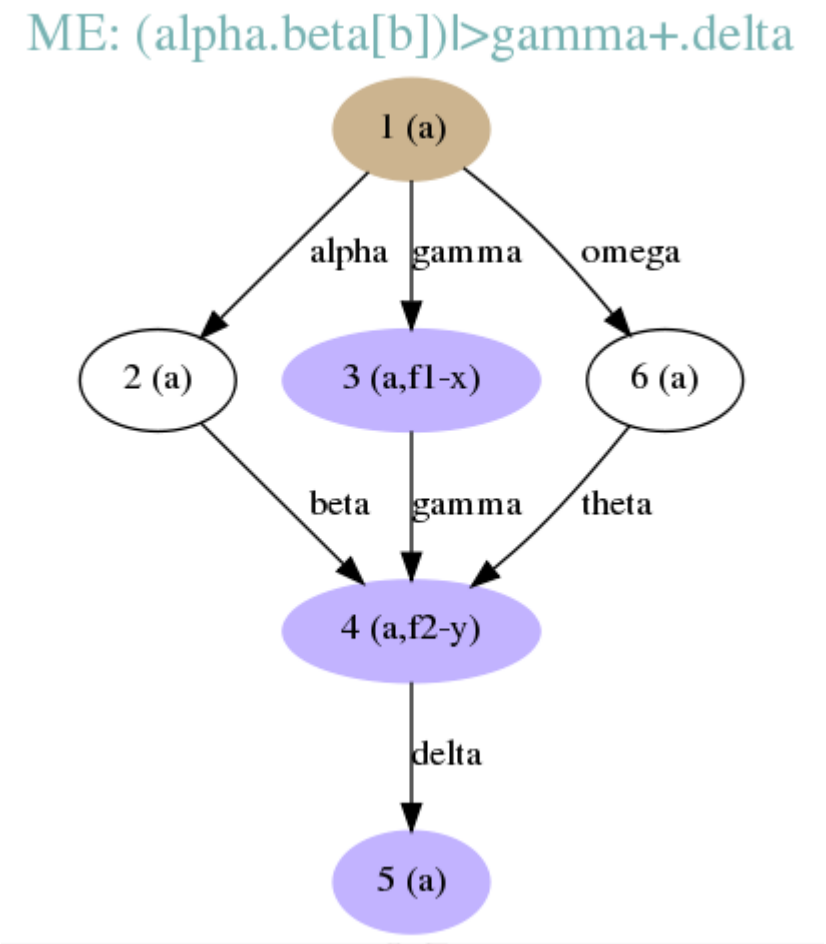
Case with Fallback Operator If the First Path Is Selected



Resulting path:

- 2 -> 4 -> 5

Case with Fallback Operator If the second Path Is Selected

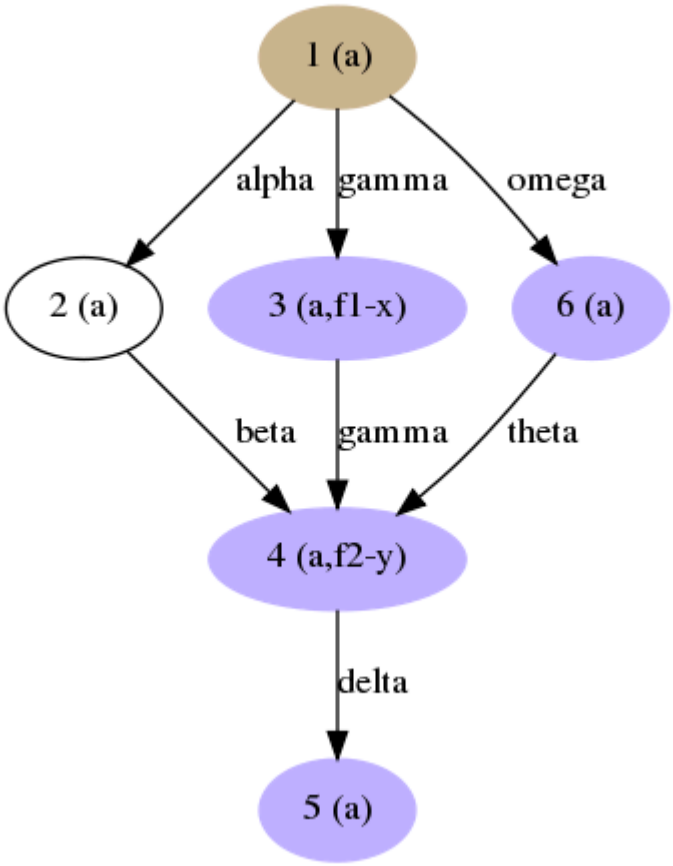


Resulting path:

- 3 -> 4 -> 5

Case with Fallback and OR Operators Together

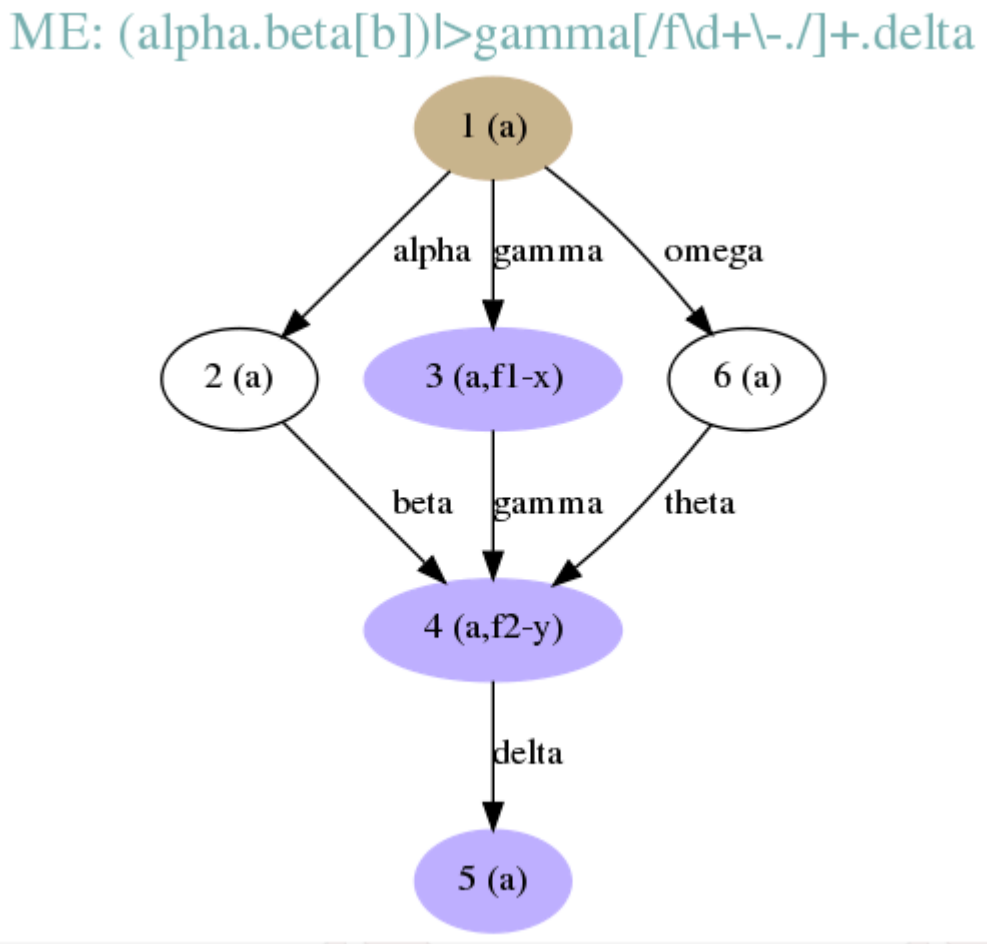
ME: (((alpha.beta[b])|>gamma+)|(omega.theta)).delta



Resulting paths:

- 3 -> 4 -> 5
- 6 -> 4 -> 5

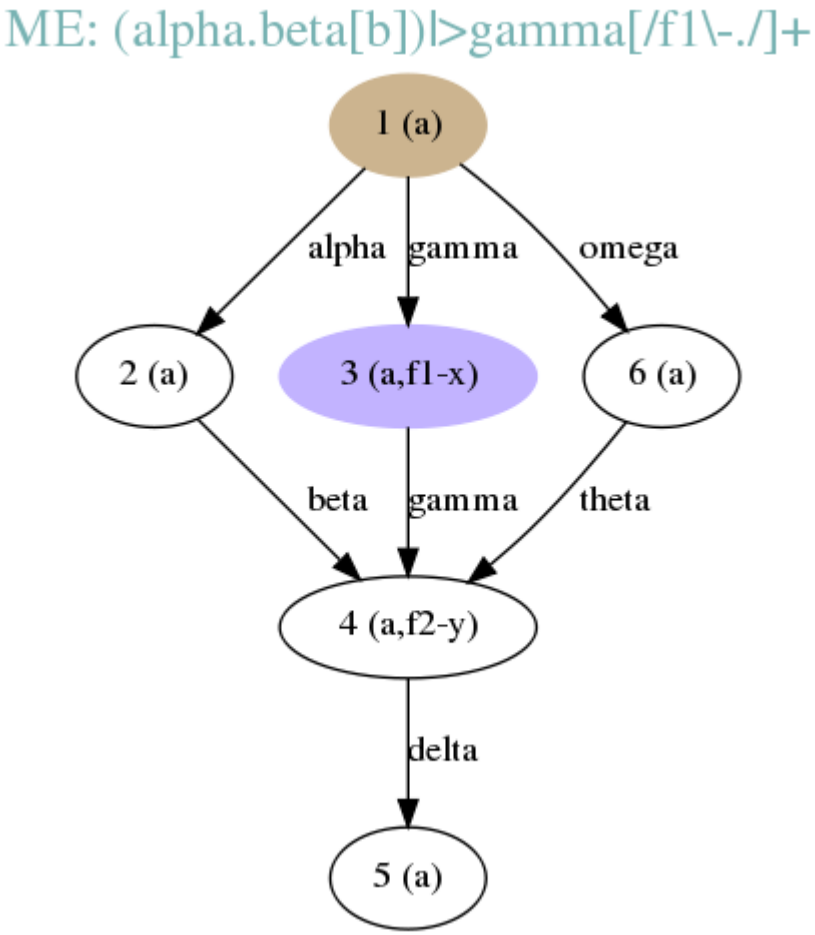
Case with Fallback Operator Using regexp in Node Type



Resulting path:

- 3 -> 4 -> 5

Another similar example



Resulting path:

- 3

Appendix - Old DSL Functions

This appendix lists the main old Domain-Specific Language (DSL) functions that you could use with Structured Data Consolidation (SDC), the Consolidation Server's ancestor. For each, you can find the Groovy and Java equivalent functions.

DSL	`delete()`
Groovy	`deleteDocument(it, false /* shouldBeRecursive */)`
Java	`handler.deleteDocument(document, false /* shouldBeRecursive */)`
DSL	`delete(uri)`
Groovy	`deleteDocument(uri, false /* shouldBeRecursive */)`
Java	`handler.deleteDocument(uri, false /* shouldBeRecursive */)`
DSL	`addCustomDirective(name, value)`
Groovy	`it.withDirective(name, value)`
Java	`handler.withDirective(name, value)`
DSL	`clearCustomDirective(name)`
Groovy	`it.deleteDirective(name)`
Java	`handler.deleteDirective(name)`
DSL	`vertexGet(path("path.to.nodes"))`
Groovy	`match(it, "path.to.nodes")*.last().flatten()`
Java	`GraphMatchHelpers.getPathsEnd(handler.match(document, "path.to.nodes"))`
DSL	`deleteParts(metaName)`
Groovy	`it.deleteParts(metaName)`
Java	`document.deleteParts(metaName)`
DSL	`distinct(["Foo", "Bar", "Foo"])`
Groovy	`["Foo", "Bar", "Foo"].unique()`
Java	`ImmutableSet.of("Foo", "Bar", "Foo")`
DSL	`hasMeta(name)`

Groovy	`it.hasMeta(name)`
Java	`document.hasMeta(name)`
DSL	`skipIf(docType, expression)`
Groovy	`process(docType) { if (expression) { discard() } ... }`
Java	`public String get[Transformation|Aggregation]Type() { return docType; } public void process(... handler, ... document) { if (expression) { handler.discard(); } }`
DSL	`metaDel(metaName)`
Groovy	`it.deleteMeta(metaName)`
Java	`document.deleteMeta(metaName)`
DSL	`metaGet(pathExpression, metaName)`
Groovy	`match(it, pathExpression)*.last().flatten().collect { it.getMetas(metaName) }.flatten()`
Java	`final List<String> result = new ArrayList<>(); or (final IAggregationDocument doc : GraphMatchHelpers.getPathsEnd(handler.match(document, pathExpression))) { result.addAll(doc.getMetas(metaName)); } return result;`
DSL	`metaGet(paths list, metaName, metaDefaultValue)`
Groovy	`paths list*.last().flatten().collect { value = it.getMeta(metaName) (value) ? value : metaDefaultValue }`
Java	`final List<String> result = new ArrayList<>(); for (final IAggregationDocument doc : GraphMatchHelpers.getPathsEnd(paths list)) { final String value = doc.getMeta(metaName)); result.add((value == null) ? metaDefaultValue : value); } return result;`
DSL	`metaSet(metaName, metaValue)`
Groovy	`it.withMeta(metaName, metaValue)`
Java	`document.withMeta(metaName, metaValue);`
DSL	`metaSet(metaName, metaValues)`
Groovy	`it.withMeta(metaName, metaValues)`

Java	`document.withMeta(metaName, metaValues);`
DSL	`metaSet(pathExpression)`
Groovy	`for (doc in match(it, pathExpression)*.last().flatten()) { it.withMetas(doc.getAllMetas()) }`
Java	`for (final IAggregationDocument doc : GraphMatchHelpers.getPathsEnd(handler.match(document, pathExpression))) { document.withMetas(doc.getAllMetas())); }`
DSL	`metaSet(targetMetaName, pathExpression, sourceMetaName)`
Groovy	`metas = match(it, pathExpression)*.last().flatten().collect { it.getMetas(sourceMetaName) } for (m in metas) { it.withMeta(targetMetaName, m) }`
Java	`final List<List<String>> selection = new ArrayList<>(); for (final IAggregationDocument doc : GraphMatchHelpers.getPathsEnd(handler.match(document, pathExpression))) { selection.add(doc.getMetas(metaName)); } for (final List<String> metas : selection) { document.withMeta(targetMetaName, metas); }`
DSL	`metaSet(pathExpression, metasList)`
Groovy	`docs = match(it, pathExpression)*.last().flatten() for (doc in docs) { for (metaName in metasList) { values = doc.getMetas(metaName) if (values) { it.withMeta(metaName, values) } } }`
Java	`for (final IAggregationDocument doc : GraphMatchHelpers.getPathsEnd(handler.match(document, pathExpression))) { for (final metaName : metasList) { final List<String> values = doc.getMetas(metaName); if (values != null) { document.withMeta(metaName, values); } } }`
DSL	`metaSet(targetMeta, pathExpression, sourceMeta, allowedTypes)`
Groovy	`docs = match(it, pathExpression)*.last().flatten().collect { if (allowedTypes.contains(it.getType()) { it } } for (doc in docs.flatten().minus(null)) { values = doc.getMetas(sourceMeta) if (values) { it.withMeta(targetMeta, values) } }`
Java	`final List<IAggregationDocument> selection = new ArrayList<>(); for (final IAggregationDocument doc :`

	`GraphMatchHelpers.getPathsEnd(handler.match(document, pathExpression))) { if (allowedTypes.contains(doc.getType())) { selection.add(doc); } }for (final IAggregationDocument doc : selection) { final List<String> metas = doc.getMetas(sourceMeta); if ((metas != null) && ! metas.isEmpty()) { document.withMeta(targetMeta, metas); } }`
DSL	`metaSet(targetMeta, pathExpression, sourceMeta, conditionMeta, allowedValues)`
Groovy	`docs = match(it, pathExpression)*.last().flatten().collect { if (allowedValues.contains(it.getMetas(conditionMeta)) { it } } for (doc in docs.flatten().minus(null)) { values = doc.getMetas(sourceMeta) if (values) { it.withMeta(targetMeta, values) } }`
Java	`final List<IAggregationDocument> selection = new ArrayList<>(); for (final IAggregationDocument doc : GraphMatchHelpers.getPathsEnd(handler.match(document, pathExpression))) { if (allowedValues.contains(doc.getMetas(conditionMeta)) { selection.add(doc); } } for (final IAggregationDocument doc : selection) { final List<String> metas = doc.getMetas(sourceMeta); if ((metas != null) && ! metas.isEmpty()) { document.withMeta(targetMeta, metas); } }`