Ab Initio for Intermediate Level

Introduction – Part2

LEVEL – ARCHITECTURE

About the Author

Created By:	Sankaran Krishnan, Raffic Abdullah, Sinoy Xavier, Simi John
Credential Information:	
Version and Date:	1.0 19/01/2012

Cognizant Certified Official Curriculum

Icons Used

Questions

Tools

Hands on Exercise

Coding Standards

Test Your Understanding

Reference

Demonstration

A Welcome Break

Contacts

Context Setting: Overview

- Introduction:
  - In the forthcoming sections we will
    - Introduce the concepts like PSETs and Continuous flows
    - Explain latest trends in AbInitio
    - Introduce tuning concepts of AbInitio
    - Introduce EME and BRE

Objectives

- After this course you will be able to:
  - Understand how to approach a problem solution using Conduct It and PSETs
  - Understand the tuning techniques of AbInitio
  - Know more about EME and BRE

Introduction to parameters

- A parameter can be used to specify a value controlling some detail of an object such as a graph or component. Every parameter has two basic parts:
  - Declaration: To establish its name. In addition, declaration attributes specify various things like what type of value a parameter can hold; whether the parameter is Input or Local.
  - Definition: To define its value if any. The definition has attributes of its own, which specify the value itself; how the value should be interpreted; and whether the value is embedded in the object itself, or held in a file.
- The collection of all of the parameters defined on an object (e.g. graph) is called its parameter set. By changing the parameter set's values, the object’s behavior can be controlled. A graph's parameter set is made up only of parameters you create; there are no default graph parameters.

Parameter definitions

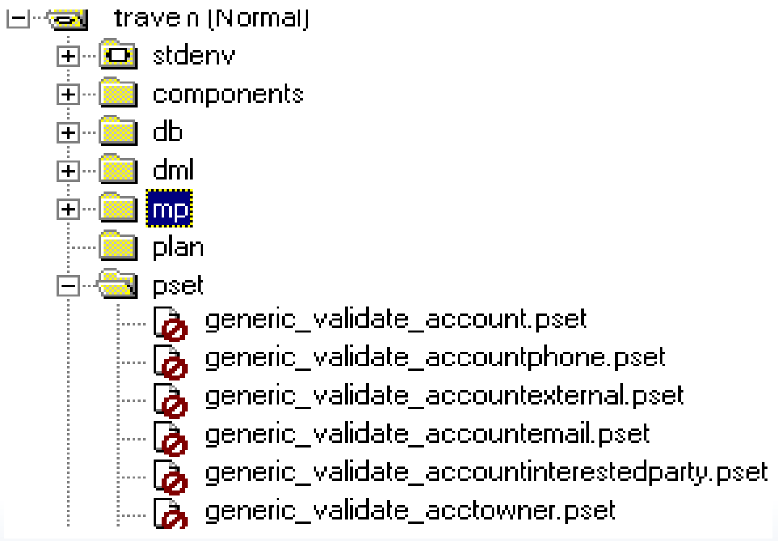
- Every parameter has to have a declaration. But an Input parameter does not need a definition until it's time to use it at runtime. (Local parameters are different: they have to have definitions from the start, because their values can't be specified by external users)
- Input parameters can also have more than one definition. For example, an Input parameter can have a default definition directly associated with its declaration as well as additional definitions in one or more input value files. The input value file to use would be specified by the user at runtime.
- Input file and input DML are Input parameters; that means the GDE will prompt to supply values for them every time the graph is run, regardless of whether they already have a default Value defined.
- For the parameter created as Local parameter, the value can be changed in the Parameters Editor only and is not accessible to change during the graph is run.
- The collection of all the Input parameters for a graph is called its Input parameter set. Values for an Input parameter set can be saved in parameter set files and reused when needed.

Parameter Set files

- Upon running the graph which has graph parameters set in, a parameter setting window pops up showing the input parameters and local parameters declared and defined for that particular graph.
- These things can be saved as a pset which then becomes a reusable file and the source of this pset is the absolute graph name on which it is defined.
- This enhances the reusability of a graph and if it has to be reused but with different parameters setting, multiple pset's on that graph can be created and there is no need to create the graphs multiple times.
- One graph can have multiple pset's defined over it with same parameter name or declaration but with different definition.

Parameter Set files– contd.

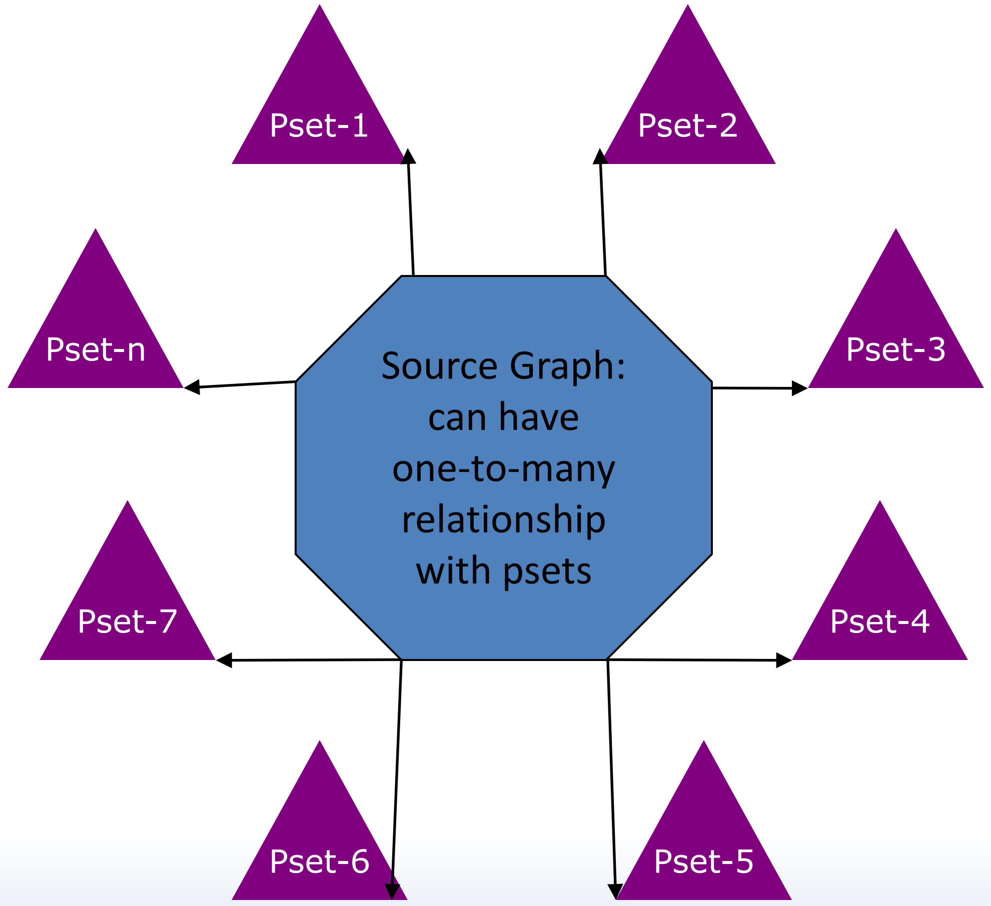
- To run the graph using pset, just trigger the pset you want to use and it will automatically trigger and open the source graph associated with it and upon running the graph, it will execute based on the parameter setting of that pset.
- For e.g. as shown in the snapshot , there are many generic validate graphs present under the pset folder of a sandbox which are having the same source graph but with different parameters .



```
trave n [Normal]
  stdenv
  components
  db
  dml
  mp
  plan
  pset
    generic_validate_account.pset
    generic_validate_accountphone.pset
    generic_validate_accountexternal.pset
    generic_validate_accountemail.pset
    generic_validate_accountinterestedparty.pset
    generic_validate_acctowner.pset
```

Parameter Set files – Cont.

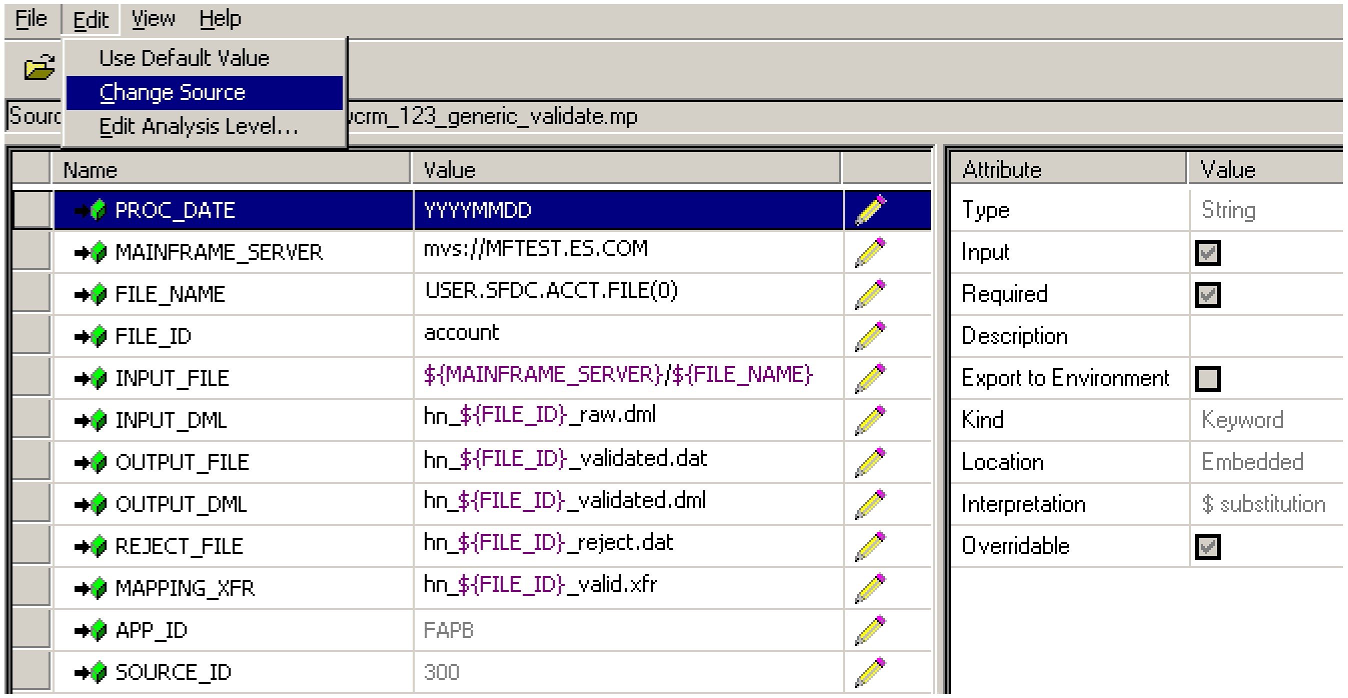
- The below shows the one-to-many relations that a graph can have with multiple psets.



Parameter Set files – Cont.

- The source graph associated with pset can be changed to another graph if required. Upon opening the pset, change the source graph by going to

  Edit (tab) → change source → new source graph name



cm_123_generic_validate.mp

Name	Value
PROC_DATE	YYYYMMDD
MAINFRAME_SERVER	mvs://MFTEST.ES.COM
FILE_NAME	USER.SFDC.ACCT.FILE(0)
FILE_ID	account
INPUT_FILE	${MAINFRAME_SERVER}/${FILE_NAME}
INPUT_DML	hn_${FILE_ID}_raw.dml
OUTPUT_FILE	hn_${FILE_ID}_validated.dat
OUTPUT_DML	hn_${FILE_ID}_validated.dml
REJECT_FILE	hn_${FILE_ID}_reject.dat
MAPPING_XFR	hn_${FILE_ID}_valid.xfr
APP_ID	FAPB
SOURCE_ID	300

Attribute	Value
Type	String
Input	✓
Required	✓
Description	
Export to Environment	☐
Kind	Keyword
Location	Embedded
Interpretation	$ substitution
Overridable	✓

Parameter Set files – Cont.

- It will prompt a window and new source path can be added.
- The point to be taken care is that the original source graph should not be deleted before changing the source path in the pset. The pset created earlier is linked to source graph path and if the original source graph is deleted it will throw an error while opening the pset itself.

Continuous graphs – Introduction

- There are two types of Ab Initio Graphs
  - Batch Graphs
  - Continuous Graphs

- Batch Graphs
  - A batch graph stops after it process all the input data that was available when it started.
  - If new data arrives while the graph is running, the graph must be restarted to process it

Continuous graphs – Introduction Cont.

- Continuous Graphs
  - These are graphs which read input continuously and produce usable output as it runs.
  - These are graphs which might or might not go forever
  - An emblematic example for continuous job processing is Web Logs. Data is continuously arriving and up-to-date reports are generated frequently.

Continuous graphs – Introduction Cont.

- Continuous Graphs can be designed to do any of the following
  - Run forever.
  - Exit when no more data is available.
  - Exit based on Content of data.
  - Exit in response to external event.
- A continuous graph includes
  - One or more subscribers. A subscriber is the only allowed data source.
  - A publisher at end of each data flow.
  - In between the subscribe and publish component there could be a number of continuous or continuity enabled components depending upon the requirements

Subscribers

- Data enters the continuous graph through a subscriber component.
- In other words, Subscriber is a component which is used to write the data from various sources into a continuous flow graph
- Originates computepoints and checkpoints.
- There are various components which are included as Subscribers like Batch Subscribe, Generate Records, JMS Subscribe, MQ Subscribe, Subscribe and Universal Subscribe.

Publishers

- Data leaves the continuous graph through a publisher component.
- In other words, A publisher is a component which is used to write the data to various destinations
- Consumes computepoints and checkpoints.
- There are various components which are included as Publishers like Multipublish, MQ Publish, JMS Publish, Publish, Trash, Continuous Multi Update table.

Compute-points And Checkpoints

- Continuous graphs are made possible by the continuous components and by the compute-points and checkpoints.
- Computepoints and checkpoints are the extra packets of information sent between records on the data flow.
- Subscribers creates these packets and the publishers consumes them.
- Both checkpoints and compute-points cause processing components to do the pending computations, and both cause publishers to commit data.
- Whenever publisher receives a compute-point or checkpoint, it sends the output to the queue or file specified as destination.

Computepoints And Checkpoints Cont.

- Computepoints
  - They mark the block of records that wanted to be processed as a group.
  - In a multiple input flows continuous graph, Computepoints is used to indicate which block of data on one flow corresponds to which blocks of data on other flow
  - When a program component receives a compute-point on all input flows, it completes the processing of all the records it has received since its last compute-point.
  - It is as if the component reached the end of data but does not terminate.
  - Many compute points can be active at the same time.
  - A compute-point after each record can be generated, for minimal latency requirement.

Computepoints And Checkpoints Cont.

- Checkpoints
  - These graphs periodically save intermediate states at special markers in data flows called checkpoints.
  - A check point is a special form of compute-point.
  - All checkpoints are compute-points whereas all compute-points are not checkpoints.
  - If the processing is interrupted for any reason it allows to restart from last check point. For this to happen
    - Each component saves its state to the disk and then passes the checkpoint to its output flows.
    - All publishers must finish processing the checkpoint.

Computepoints Vs Checkpoints

Computepoints	Checkpoints
Minimal latency	More latency into the process
Doesn’t incur the time or I/O cost associated with writing checkpoint files	Writing checkpoint files cost time and IO
Computepoint can occur so frequently	Checkpoints should not occur more often.
Less Involvement of Co>Operating System when compared to checkpoint	Involves much more of Co>Operating System
Usage is less in the way of processing resources	Uses much more in the way of processing resources
Use computepoints for latency control	Use checkpoints for graph recovery control

Restrictions on Continuous Graphs

- All components in the graph must be continuous component or they must be continuity enabled.
- There must be at least one Publisher in the graph, to determine when the checkpoint can be committed
- All subscribers must issue checkpoint and compute point in the same sequence
- The graph must run in a single phase. More than one phase is not allowed.
- All data in continuous flow graphs must come from a subscribe.

Subscribe Component

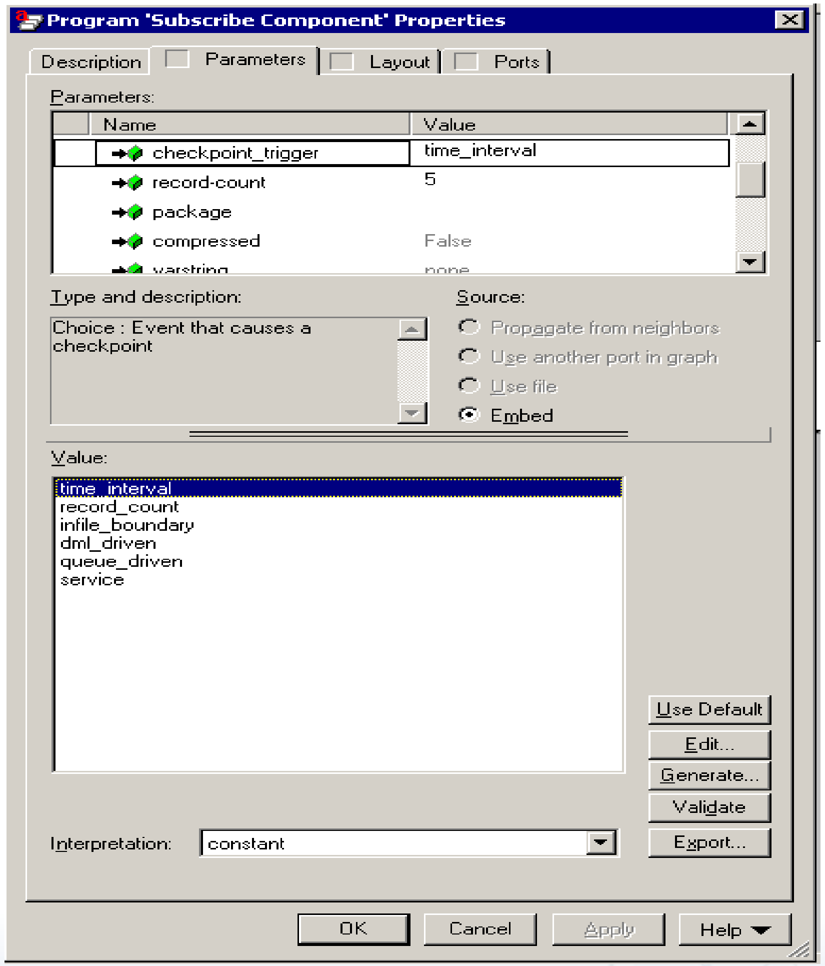
- Location – Component Organizer -> Continuous Folder
- Some of the important parameters of Subscribe components are explained below.
  - Infile: The data the component should read. The value can be one of the following:
    - An Ab Initio queue directory
    - The name of an input file
    - The basename for a sequence of input files
  - ID: It specifies the subscriber Id for queue. The subscriber Id identifies the subscriber directory you want the component to read from
  - Record Count: It specifies how many records the component should read before issuing a checkpoint. It will be available only when the checkpoint-trigger parameter is set to record_count or time_interval.

Subscribe Component – Cont.

- Package: It specifies user-written DML functions that define the conditions under which the component issues checkpoints and computepoints.
- Wait: It specifies what the component does when it runs out of records to process.
    - True: The component waits for additional input data.
    - False: The component issues a checkpoint and exits.

Subscribe Component – Cont.

- Checkpoint_trigger: Specifies the method the subscriber uses to generate checkpoints and compute-points.
  - Time Interval : The component issues checkpoints at wall-clock intervals. Set this variable to the number of seconds you want between checkpoints.
  - Record Count: The component issue the checkpoint after reading the number of records mentioned in the Record Count parameter.



Parameters:

Name	Value
checkpoint_trigger	time_interval
record-count	5
package	
compressed	False
varstring	none

Type and description:
Choice : Event that causes a checkpoint

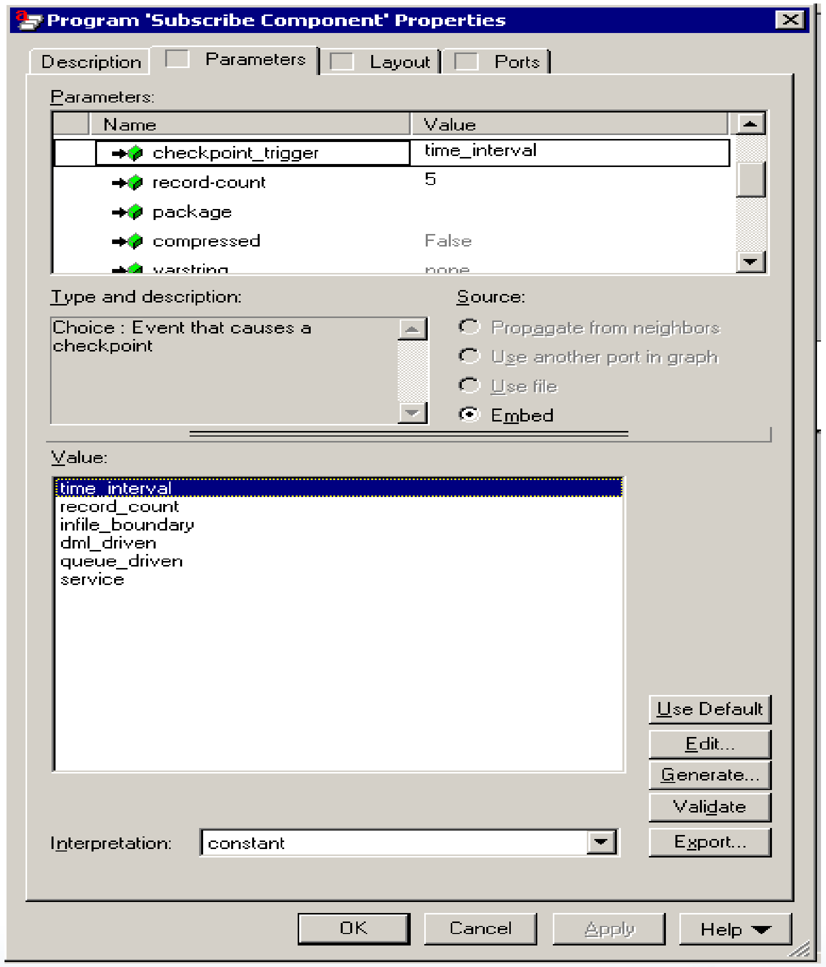
Source:
( ) Propagate from neighbors
( ) Use another port in graph
( ) Use file
(●) Embed

Value:
time_interval
record_count
infile_boundary
dml_driven
queue_driven
service

Interpretation: constant

Subscribe Component – Cont.

- Infile Boundary: The component issues checkpoint at the end of each data file.
- DML Driven: The component uses user-written DML functions to determine when to issue checkpoints and compute-point
- Queue Driven: The component generates checkpoints and compute-points in the order in which they were originally published to the queue by an upstream graph.
- Service: The component generates a computepoint after each record.



Parameters:

Name	Value
checkpoint_trigger	time_interval
record-count	5
package	
compressed	False
varstring	none

Type and description:
Choice : Event that causes a checkpoint

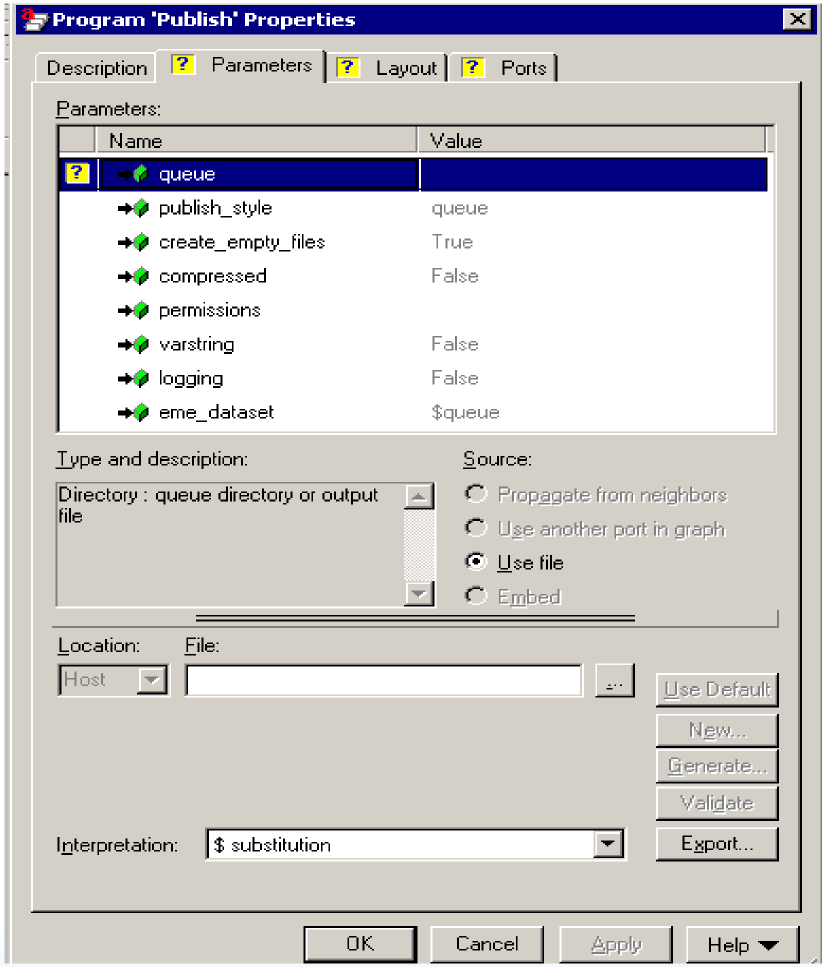
Source:
- Propagate from neighbors
- Use another port in graph
- Use file
- [✓ Embed]

Value:
time_interval
record_count
infile_boundary
dml_driven
queue_driven
service

Interpretation: constant

Publish Component

- Location – Component Organizer -> Continuous Folder
- Some important parameters:
- Queue – Specify the name of the output file, basename for a sequence of file or Ab Initio queue.
- Publish_style – The destination of the output, whether it is file, sequence of file or Ab Initio
  - Appended – It specifies the output data needs to be appended to the file mentioned in the queue parameter.



Parameters:

Name	Value
queue	
publish_style	queue
create_empty_files	True
compressed	False
permissions	
varstring	False
logging	False
eme_dataset	$queue

Type and description:
Directory : queue directory or output file

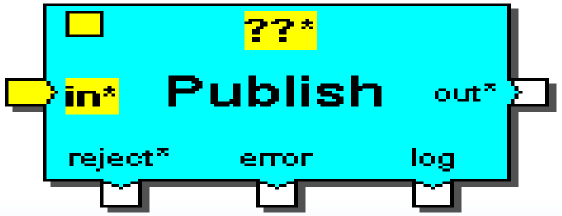
Source:
Propagate from neighbors
Use another port in graph
[✓ Use file]
Embed

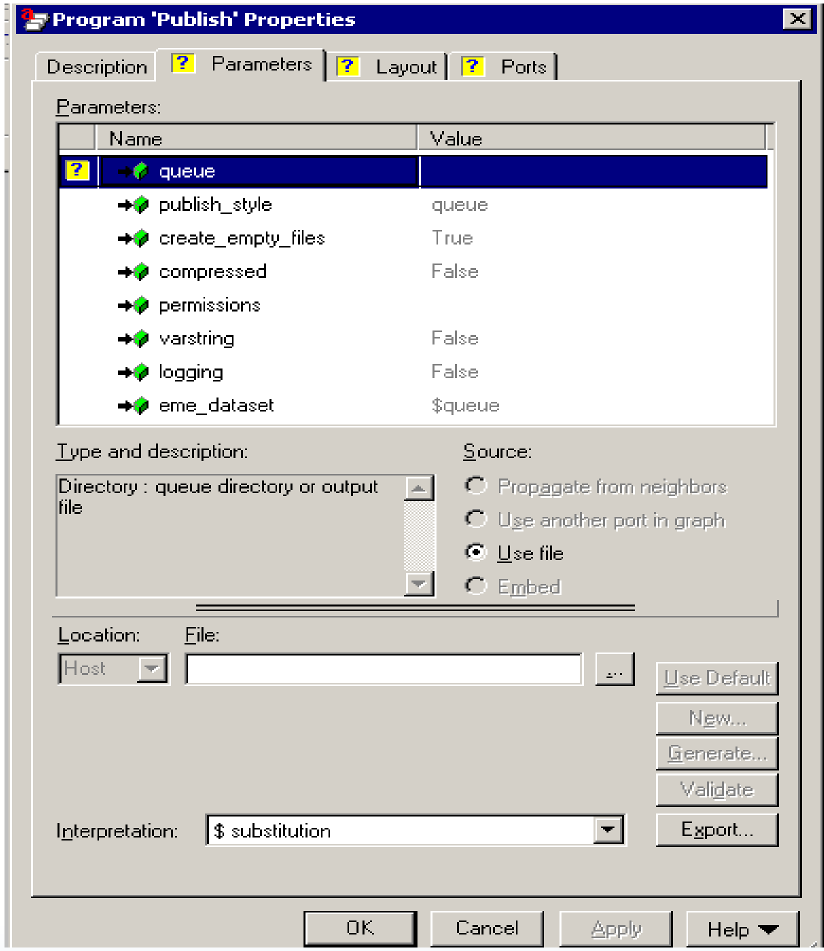
Location: Host
File:

Interpretation: $ substitution

Publish Component – Cont

- Publish_style –
  - files_after – This would write into a sequence of files. The basename of the sequence is the filename in the specified by the queue parameter to which adds a dot(.) followed by a zero padded decimal.
  - Queue – This sends to an Ab Initio queue.
  - None – This sends the data to the trash.




Program 'Publish' Properties

Parameters:

Name	Value
queue	
publish_style	queue
create_empty_files	True
compressed	False
permissions	
varstring	False
logging	False
eme_dataset	$queue

Type and description: Directory : queue directory or output file

Source: Propagate from neighbors / Use another port in graph / [✓ Use file] / Embed

Location: Host

File:

Interpretation: $ substitution

Queues in Continuous graphs

- AbInitio queue is a Data Structure that works on FIFO (First in first out) principle.
- Queues are the most reliable method for storing continuous flow data.
- In a way, AbInitio queues are analogous to multi-files in ordinary AbInitio graphs. They provide a method for storing records in an ordered sequence of files.
- They support the following
    - Record based persistence.
    - Publishers write data to queue.
    - Transactions
    - The ports of one or more subscribers read data from the queue.
    - Subscribers to the queue read the data in the order it was written

Publishers and Subscribers Interface with queue

- When a publisher receives a checkpoint or compute-point, it makes available all data it received since last point by writing a file into a queue. This file make the data available to subscribers in that queue.
- In each subscribers subdirectory, the publishers makes a hard link to the new file in the queue directory and then deletes the file from main queue directory. This allows the data to store once no matter how many subscribers reads.
- When the graph commits the data the file system removes it from the queue.
- The main queue directory has a write cursor. The publisher uses the write cursor to determine the sequence of files to write the data.
- The subscriber uses the write cursor to determine what data is available for read.
- Each subscribers subdirectory has a read cursor that enable it to avoid reading records more than once

Stopping a Continuous Graph

- Stopping a Continuous graph
  - The continuous graph doesn’t need to run forever. It can be stopped forcefully or gracefully.
- Forcefully stopping a Continuous graph
  - When a graph is stopped forcefully it stops immediately regardless of the current state of the queues, database and components involved.
  - The graph can be stopped forcefully in the below ways
    - Click the stop button.
    - Choose Stop>Stop.
    - Execute the command m_kill

Stopping a Continuous Graph Cont.

- Gracefully Stopping a Continuous graph
  - When the graph is stopped gracefully, it deletes all temporary checkpoint files and Clean up all graph internal and global state. Two ways of doing this are
    - By Design
      - The graceful stoppage by design can be done by setting subscriber component the wait parameter to false so that the graph stops when it runs out of data.
      - By using the types and methods of dml_driven of checkpoint_trigger parameter of subscriber component.
    - By Executing a command
      - Choose Stop >Shutdown or execute m_shutdown, to stop the graph in the next checkpoint.
      - Choose Stop >Force Shutdown or execute m_shutdown –f which force subscriber to generate checkpoint as soon as possible.

Restarting a Continuous Graph

- The options available for restarting a batch graph are the same for continuous graph, but the following factors are critical in proper recovery
  - Processing each record exactly once.
  - Maintain the record order.
- The Continuous components do the following in recovery
  - Subscriber “replay” all records since the last committed checkpoint to ensure proper order of record.
  - Processing components process the “replayed” records.
  - Publishers ignore the already committed records before the graph fails as they commit on compute-points apart from checkpoints.

Restarting a Continuous Graph – Cont.

- The graph can be restarted through GDE/Command line
- Restarting Through GDE can be done by clicking Run button (F5) and by selecting the needed option in the Recovery File Found dialog
  - Continue the job from last checkpoint,
  - Restart the job from beginning (rollback and delete recovery file)
- For restarting the continuous graph from command line.
  - To run in recovery mode Just do a normal run of the graph.
  - To clean and avoid restarting in recovery, run the command

    m_rollback –d job_name

Plan>It – Introduction

- Plan>It helps to create, manage and run large-scale data processing systems by sequencing the ETL jobs.
- Plan>It provides a framework to create complete production systems consisting of AbInitio graphs, custom scripts and third party programs.
- It is based on the scalable and reliable AbInitio technology

Plan>It – Introduction Cont.

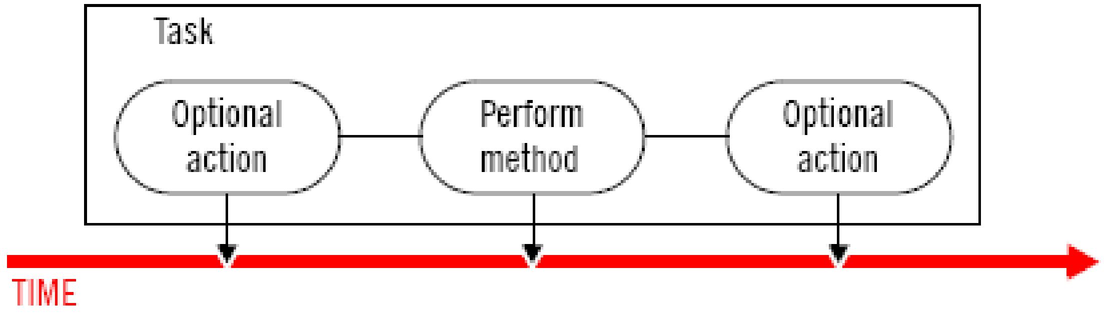
- With Plan>It its easy to connect different graphs i.e. creating graphs of graphs which is known as Plan.
- Plan can also contain the scripts , sub-plan, third party programs all working together as a large scale data processing system.
- Systems created by Plan>It are robust in design and the graphs, scripts and conditional logic of the systems are encapsulated.

Plan Elements

- Plan consists of the following elements.
  - Tasks – which can be the graphs, scripts and other programs.
  - Methods – which performs the actions of the tasks.
  - Parameters – which passes information between tasks.
  - Relationships – which determines the order in which the tasks to be executed.

Plan Elements-Tasks

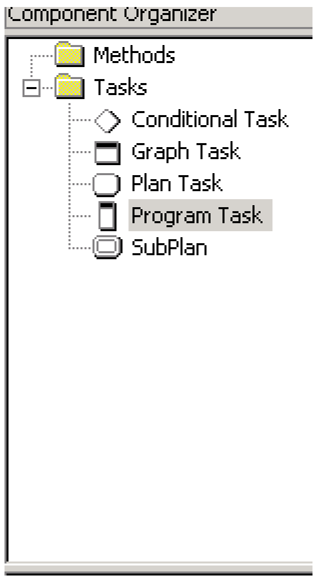
- Tasks are the building blocks of a plan. Each tasks can be thought of an ordered series of actions.
- The main action of the task is called its Perform method. Apart from the perform method there can be Optional actions also.



- The perform method gives the task its identity.

Plan Elements-Tasks Cont.

- There are different types of tasks
  - Graphs Tasks - Perform method is a Graph
  - Program Tasks - Perform method is a script or program
  - Plan Tasks - Perform method is a plan
  - Conditional Tasks - Perform method consists of evaluating an expression or an external program determining whether it is true or false

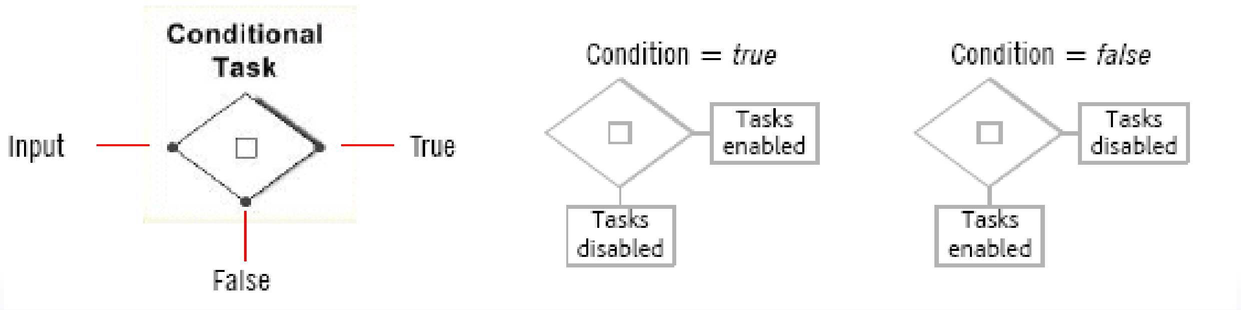


Plan Elements-Tasks Cont.

- Graph Tasks
  - Used to run the AbInitio graphs with in the plan
  - The perform method parameters - plan, argument and working directory determine the plan name, the arguments and the directory in which to execute the plan.
- Program Tasks
  - Used to run the executables and scripts.
  - The same can be determined by the task properties parameter mode i.e. Mode=Executable (External Program) or Mode=script(Embedded script)

Plan Elements-Tasks Cont.

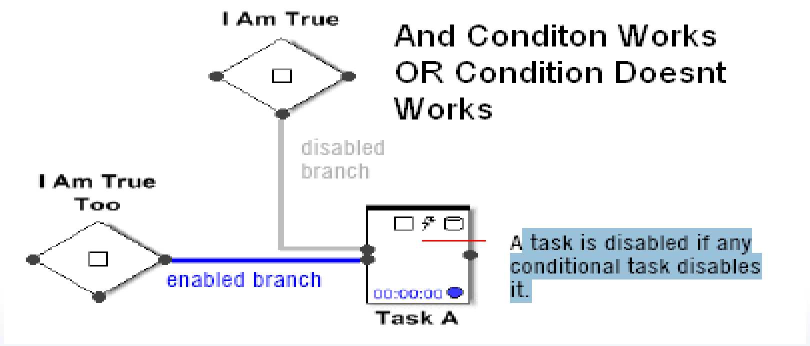
- Plan Tasks
  - Used to run different plans
  - The perform method parameters - plan, argument and working directory determine the plan name, the arguments and the directory in which to execute the plan.
- Conditional Tasks
  - Used to implement conditional logic within a plan.
  - The perform method can be specified as an expression or a program.



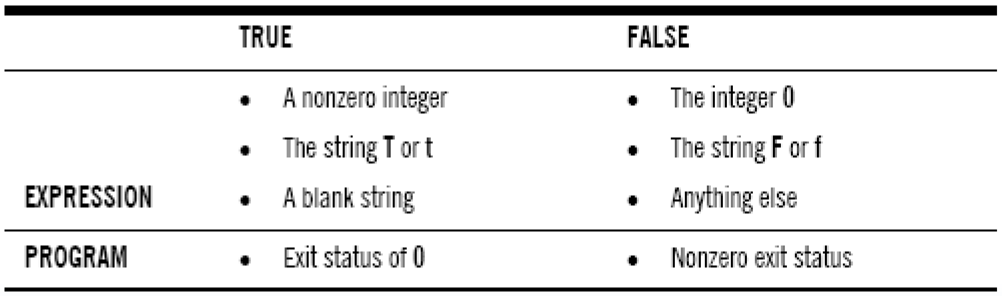
Conditional task Rules

- Conditional Tasks
  - In general Conditional tasks allows only the tasks of the enabled branch to run.
  - The primary rule is that Plan>It disables any task downstream of a disabled conditional branch unless that task is also downstream of the enabled conditional branch
  - When a task is connected to multiple conditional tasks, the task must be enabled by all its upstream conditional tasks, or Plan>It disables it.

Multiple Conditionals



Conditional Tasks Logic



	TRUE	FALSE
EXPRESSION	• A nonzero integer • The string T or t • A blank string	• The integer 0 • The string F or f • Anything else
PROGRAM	• Exit status of 0	• Nonzero exit status

Plan Elements-Method

- At particular stages of a task processing, assigning of action can be done. Such actions are known as Methods
- The method can contain a running program, evaluating a condition etc.
- Methods are the most important executable of the plan
- The perform method is mandatory for all tasks.

Plan Elements-Method Cont.

- There are different kinds of methods
  - Trigger
  - At Start
  - Perform
  - At Success
  - At Failure
  - At Warning
  - At Rollback
  - At Shutdown
  - At Period

Plan Elements-Method Cont.

- Trigger – This method assess specified condition and run the task if the returned condition is true. If condition is false the trigger method fails or continue to wait. E.g. Trigger a plan only if a file is present
- At Start – This method runs before the perform method and after the trigger method returns true(if present). E.g. Logging of Information before start up
- Perform – This method is mandatory and it performs the tasks. Its parameters should be defined for all tasks before the plan is run .

Plan Elements-Method Cont.

- At Success – This method runs when the perform method finishes successfully(i.e. Returns an exit status of 0) E.g. Send a mail to admin notifying the success
- At Failure– This method runs when the task fails(i.e. Returns an exit status of non zero) E.g. Send a mail to admin notifying the failure
- At Warning– This method runs when the task's perform method exceeds a specified execution time threshold E.g. Generate warning message and write into a file

Plan Elements-Method Cont.

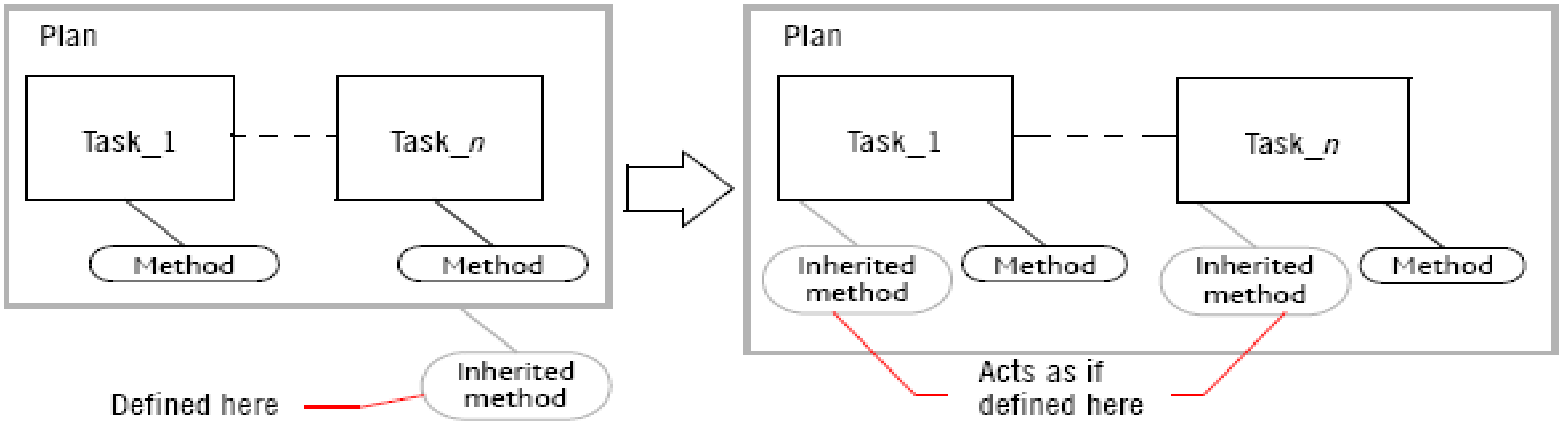
- At Rollback – This method runs as a part of Plan>It recovery mechanism. Runs after the task's At Failure method
- At Shutdown– Contains the executable to shutdown the perform method when necessary. This method is run manually.
- At Period– This method runs repeatedly at some time interval till the parent tasks is active.

Plan Elements-Method Cont.

- Further classification of the methods
  - Synchronous Method
    - There are methods which runs in synch. with relation to each other
    - Trigger, At Start, Perform, At Success and At failure methods run in synch. with relation to plan events and conditions.
  - Asynchronous Method
    - There are methods which runs Asynchronously to other methods and not dependent on the plan events.
    - At Shutdown, At Warning and At Period are Asynchronous method.
  - At Rollback is an exception it may either run in relation to or independent of plan events

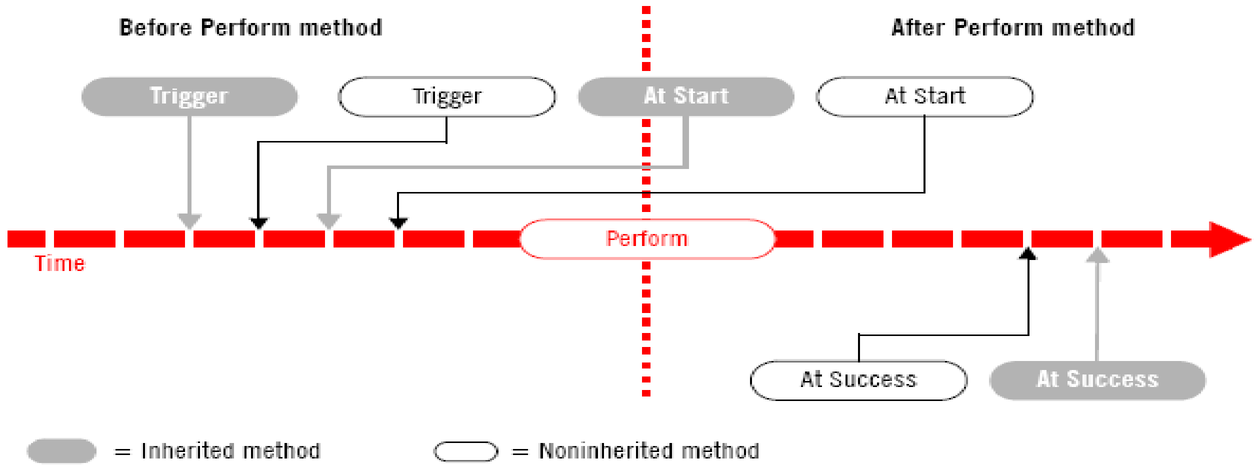
Plan Elements-Method Cont.

- Inherited Method
  - These are methods which is defined at plan level which applies to all task found below that level.
  - They allow as a easy way to apply the same method to all tasks in the plan.
  - They apply only to the tasks within the plan and not to the plan.
  - In the below diagram, Inherited method is defined on plan, But they executes in every task inside



Plan Elements-Method Cont.

- Inherited Method(order of execution)
  - When a task has both Inherited and non-inherited method, Plan>It executes methods as below



  - Before task's perform method, Plan>It executes the inherited method first, followed by analogous non-inherited method.
  - After task's perform method, Plan>It executes the non-inherited method first, followed by analogous inherited method

Plan Elements-Parameters

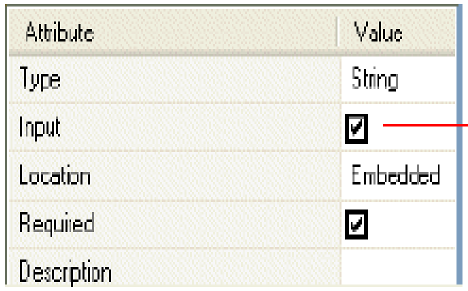
- Parameters are principal mechanism for passing information between applications in AbInitio software.
- Parameters can be assigned to any method, graph, task, plan, project, and common project and those project can be shared throughout the object hierarchy
- When a graph has parameters, they appear in the Input values tab of task properties dialog.

Plan Elements-Parameters Cont.

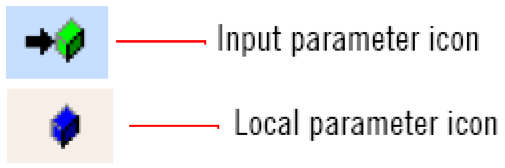
- Plan>It modifies the parameter values as plan runs. Multiple plans, graphs and tasks can be made to read and modify the same parameters. Steps for the same is below and explained in the upcoming slides:
  - Expose the parameter by declaring its input parameter while creating a plan.
  - While creating a task that is to override a parameter, declare the parameter dynamic by including it on the dynamic parameter tab of the task.
  - While creating a method to override the parameter value , use dtm parameter override command

Plan Elements-Parameters Cont.

- Exposing a Parameter: Input Attribute
  - The input attribute box determines whether the parameter is an input parameter or a local parameter



Attribute	Value
Type	String
Input	✓
Location	Embedded
Required	✓
Description	



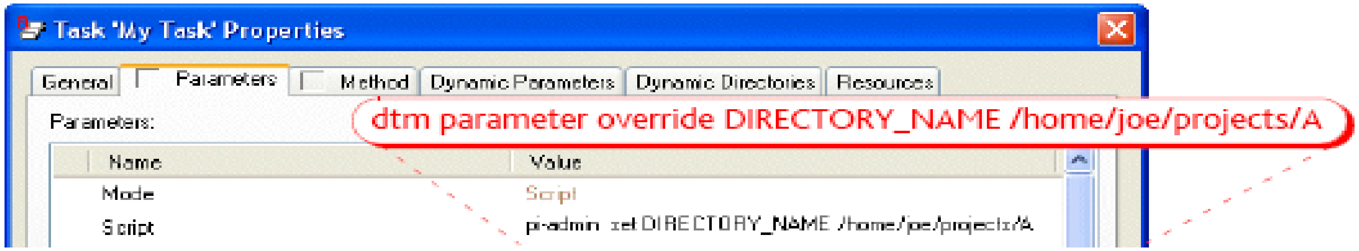
  - The input parameter can have their values modified by plans, tasks, and graphs outside projects. The input value editor dialogue appears at runtime to supply values for input parameter.
  - External graphs can neither see the local parameters nor edit their values. The value can be provided by editing the value field in Edit parameters dialog.

Plan Elements-Parameters Cont.

- Declaring a Parameter: Dynamic parameter tab
  - The task properties and Plan properties dialog includes a Dynamic parameters tab where the input parameters can be listed which are intend to modify using one of the task's method.
  - The parameter should be declared dynamic before any method tries to modify else the plan will fail.

Plan Elements-Parameters Cont.

- Modifying a Parameter: dtm parameter override
  - To Modify the value of input parameter the dtm parameter override command can be used. The syntax of the command is
    - ***dtm parameter override*** *paramname newvalue*
  - Suppose for example there is an input parameter containing the DIRECTORY_NAME which has some default value /tmp/home/ at the plan level. This could be override at the tasks level e.g. My Task using the above command. The command could be declared at one of the tasks method using a script.



dtm parameter override DIRECTORY_NAME /home/joe/projects/A

Name	Value
Mode	Script
Script	pi-admin set DIRECTORY_NAME /home/joe/projects/A

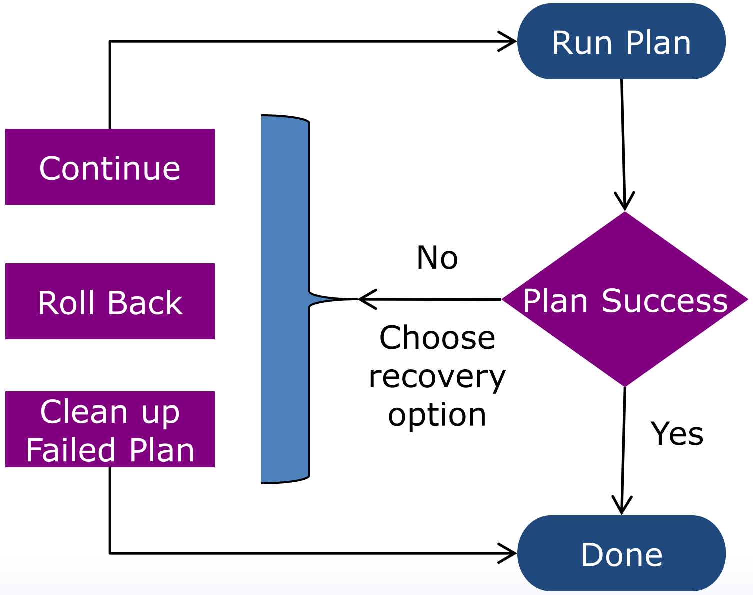
  - The modified value is visible to the downstream tasks of My Task.

Plan Elements - Relationship

- When the tasks icons are dragged into the plan the way the tasks are connected determines the sequence of their run.
- When the tasks are connected the left task must complete before the right one begins.
- A task that has no dependencies can start running as soon as the plan starts.
- When the task are connected, the relationship and dependencies should be created between the tasks.

Plan – Life Cycle

- When a plan is run, Plan>It creates an executable copy in the memory while it runs and recovers if it fails.
- Schematic representation of a plan is as below



Plan>It Perseverance and Recovery

- When Plan>It encounters a problem, it runs as many independent tasks as possible and only then it stops the process and provide options for recovery.
- Plan>It and GDE provides several options for recovery. Some of the options are listed below.
  - Continue from the point of recovery
  - Rollback the job and then restart it
  - Roll back without rerunning
  - Remove the job and restart the plan
- When the plan fails the job output dialog informs of the failure and the job recovery dialog appears and provides the options for handling the failure.

Plan>It Recovery Options

- Continue from the point of recovery
  - Plan>It runs the "At rollback" method of the tasks which caused the failure and then continues forward from that point.
  - Command-line Interface(CLI) equivalent
    - ***dtm run planname –continue-from-failure***
- Rollback the job and then restart it
  - Plan>It runs the "At rollback" method of all tasks in reverse execution order, starting at the point of failure. After rolling back, it runs the plan again.
  - Command-line Interface(CLI) equivalent
    - ***dtm run planname –rollback-and-restart***

Plan>It Recovery Options Cont.

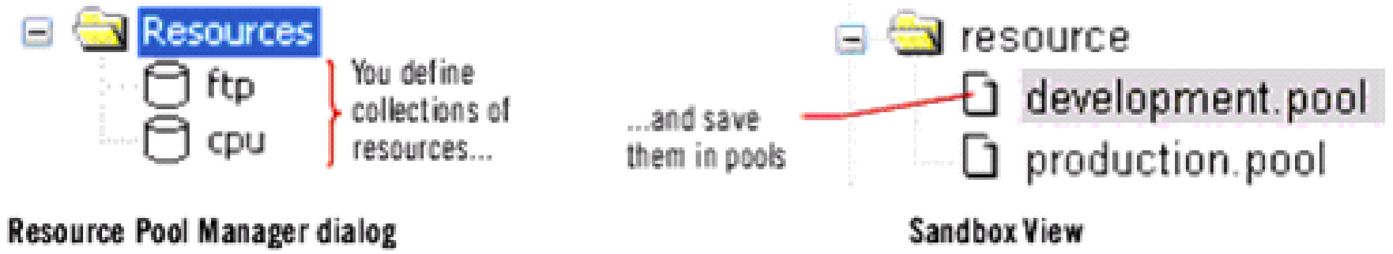
- Rollback without rerunning
  - Plan>It runs the “At rollback” method of all tasks in reverse execution order, starting at the point of failure. After rolling back, it does not rerun the plan.
  - This doesn’t appear on the Job recovery dialog. It is invoked from Run > Rollback options
  - Command-line Interface(CLI) equivalent
    - ***dtm run planname –rollback-and-stop***
- Remove the job and then restart the plan
  - Plan>It discards any processing and deletes the original recover file and runs the plan again.
  - Command-line Interface(CLI) equivalent
    - ***dtm run planname –remove-and-restart***

Resource Management

- Resource Pool is used to restrict the:
  - number of Plans and Graphs that run concurrently on the system
  - number of CPUs used for running the application
  - number of concurrent file transfers
  - concurrent connections to the database
- Command to view the resource pool settings
  - dtm resource view <complete path to resource pool file> | more

Resource and Resource Pools

- To scale resources used to meet the needs of the computing environment, you define one or more Conduct>It resources and then save those collections of resources in your sandbox as resource pools.
- A resource pool contains purely logical entities that you create and assign arbitrarily


Resources
ftp
cpu
You define collections of resources...
...and save them in pools
resource
development.pool
production.pool
Resource Pool Manager dialog
Sandbox View

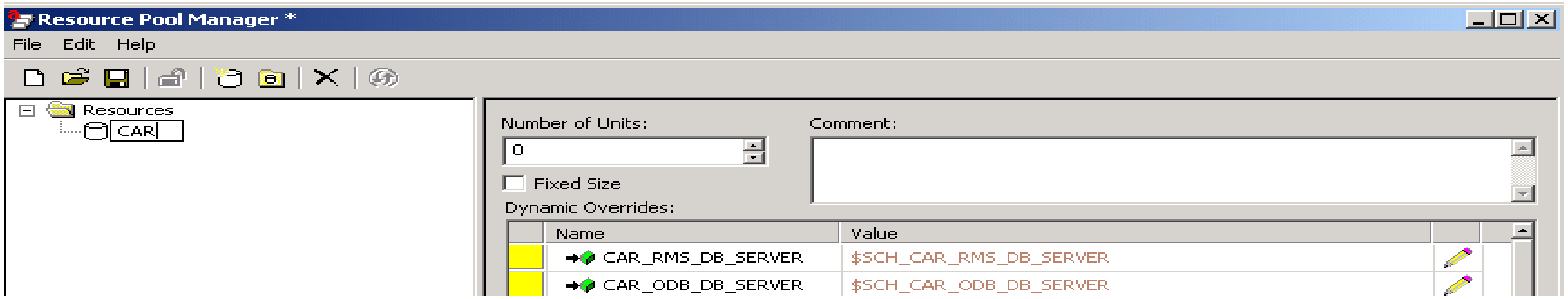
- The resource "ftp" is used to limit concurrent file transfers and "cpu" is used to limit concurrent cpu usage

Resource and Resource Pools Cont

- The resources can be assigned to the same pool or different pools.
- Having set up the resource pool, individual tasks in the plan can be assigned to use the resources in the pools
- Each such task runs only when all the resources it requires are available
- If any required resource is not available, the task will wait for it.

Resource Units

- The unit is a number that represents the capacity of the resource.



Resource Pool Manager *

File Edit Help

Resources
CAR

Number of Units: 0

Comment:

☐ Fixed Size

Dynamic Overrides:

	Name	Value
🟨	CAR_RMS_DB_SERVER	$SCH_CAR_RMS_DB_SERVER
🟨	CAR_ODB_DB_SERVER	$SCH_CAR_ODB_DB_SERVER

- Resizable resources are used for flexibility purpose. Suppose after normal work hours, when demand is reduced and additional resources become available, in order to run the plan more quickly resizable resources are used. Whereas the fixed size doesn’t have this flexibility as the units are fixed. Normally the Resource Units are distributed in the Organizational level.

Load Balancing

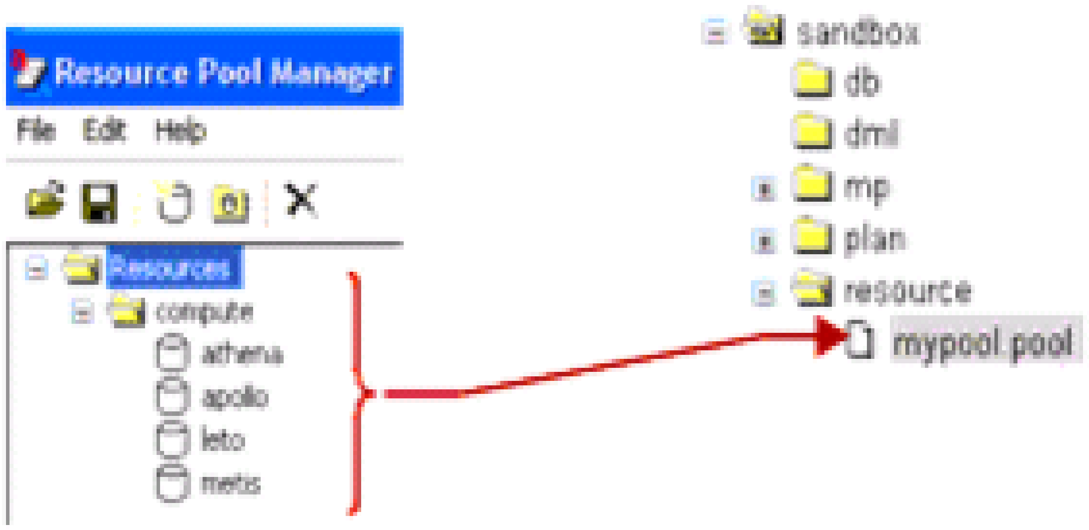
- In order to balance the load, the resources are defined and saved in a pool. If there are four machines in your environment, you might construct the pool as follows:

Group	Name	Units
compute	Athena	4
	Apollo	1
	Metis	1
	Leto	2

- For this example, assume that Athena is a four-processor machine, that Leto is a dual-processor machine, that Apollo and Metis are uniprocessor machines, and that you want to distribute tasks by processor.

Load Balancing Cont

- Having constructed this group of resources, you save it in your sandbox as mypool.pool, where it appears in the resource folder:



Load Balancing Cont

- Configure the resources to override parameter defaults as needed. On the Dynamic Overrides tab of the Resource Pool Manager dialog, override the default sandbox variables, allowing each resource to point to a separate run host and run directory:

Group	Name	Units	Variable overrides

compute	Athena	4	RUN_HOST=//athena
			RUN_DIR=tmp

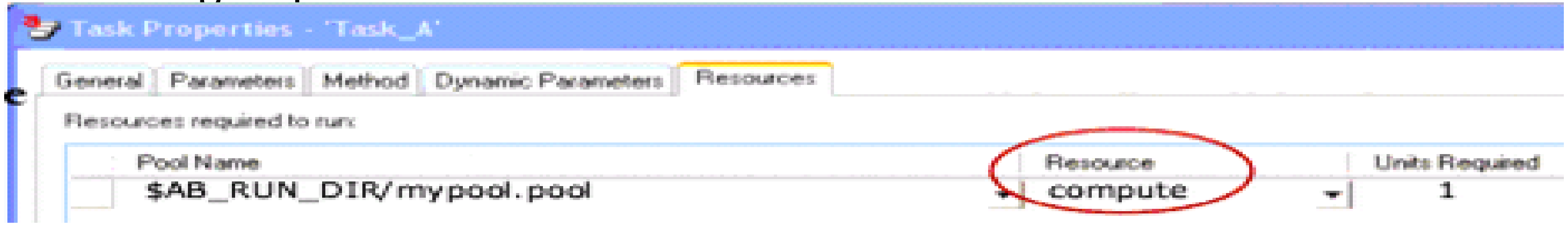
	Apollo	1	RUN_HOST=//apollo
			RUN_DIR=disk1

	Metis	1	RUN_HOST=//metis
			RUN_DIR=disk3

	Leto	2	RUN_HOST=//leto
			RUN_DIR=disk1

Load Balancing Cont

- Assign resources to tasks and plans. On the Resources tab of the Task/Plan Properties dialog, you can specify either a resource name or a resource group.

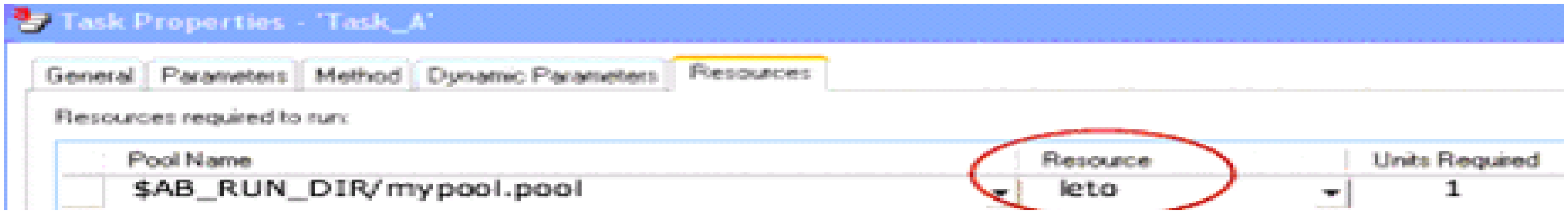

Task Properties - 'Task_A'

General | Parameters | Method | Dynamic Parameters | Resources

Resources required to run:

Pool Name	Resource	Units Required
$AB_RUN_DIR/mypool.pool	compute	1

- At runtime, Conduct>It assigns your task in round-robin fashion to the next available resource in the compute group. On the other hand, if you want a task to run specifically on, say, the server Leto, you can specify the resource Leto by name:


Task Properties - 'Task_A'

General | Parameters | Method | Dynamic Parameters | Resources

Resources required to run:

Pool Name	Resource	Units Required
$AB_RUN_DIR/mypool.pool	leto	1

Database Connectivity

- In AbInitio, if the source/destination is a database table, then to extract/load the data from or to the database tables it is necessary to first establish a connection between the database server on which the tables are located and AbInitio Co>op. This connection is established using a .dbc (database configuration) file in AbInitio.
- DBC files can be created through GDE or edited directly using an editor like vi.
- Command to create the dbc file:
  - **m_db gencfg -**<*database_name*> **>** <*dbc_filename*>.dbc

Database Connectivity Cont

- To create dbc file through GDE please follow below steps:
  - Drag a Input/Output table component→Go to Properties→Click on Config File→New

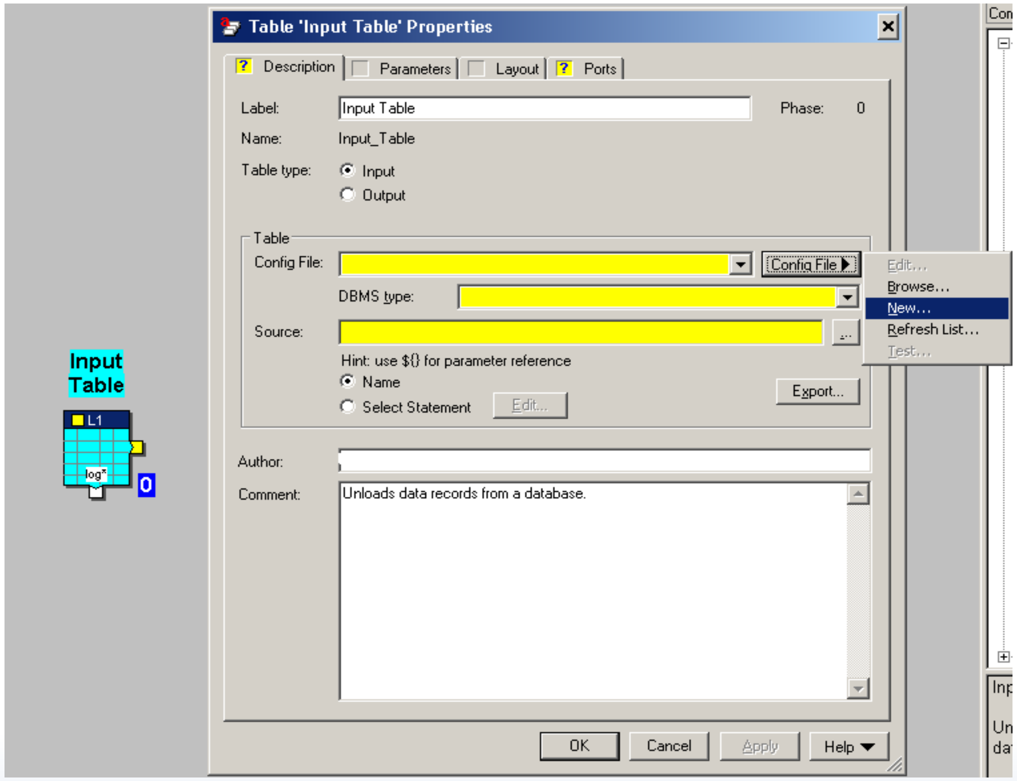


Table 'Input Table' Properties

Description | Parameters | Layout | Ports

Label: Input Table Phase: 0

Name: Input_Table

Table type: [✓ Input] Output

Table

Config File: Config File ▶

DBMS type:

Source:

Hint: use ${} for parameter reference

[✓ Name] Select Statement Edit... Export...

Author:

Comment: Unloads data records from a database.

OK Cancel Apply Help

Database Connectivity Cont

- Select a database from the list of supported databases.

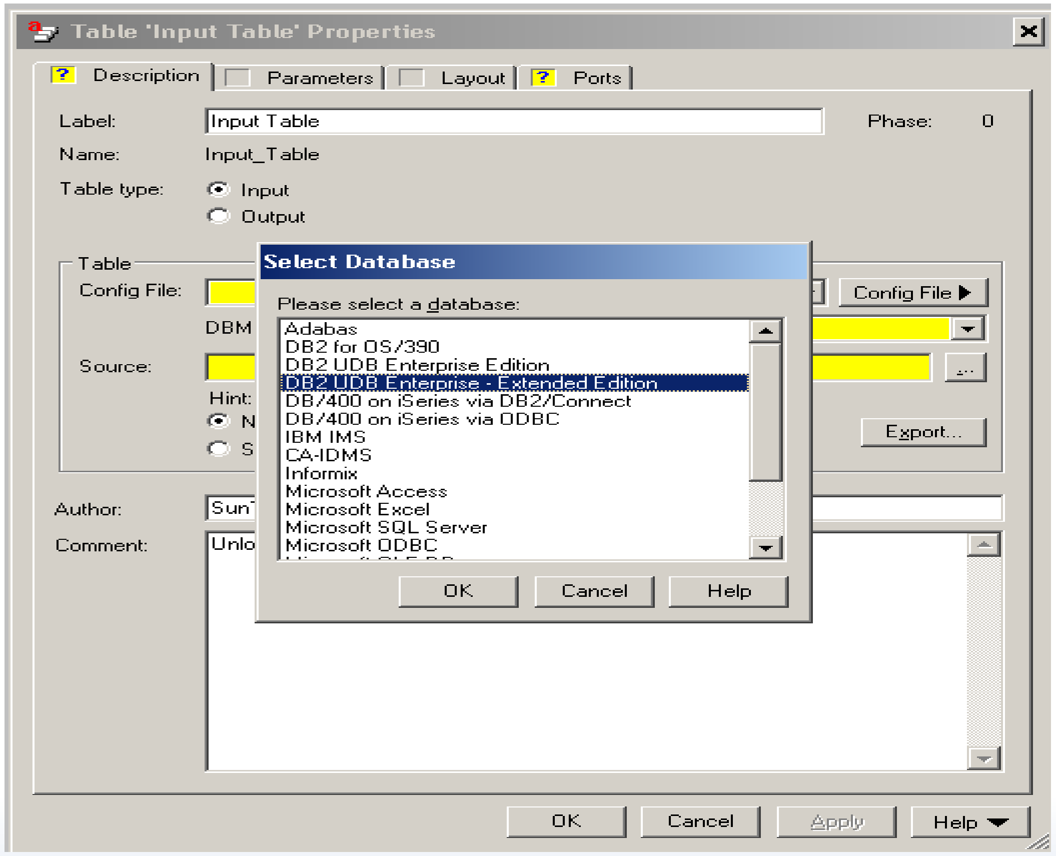


Table 'Input Table' Properties

Description | Parameters | Layout | Ports

Label: Input Table

Phase: 0

Name: Input_Table

Table type: [✓ Input] Output

Table

Config File:

DBM

Source:

Hint:

Author: Sun

Comment: Unlo

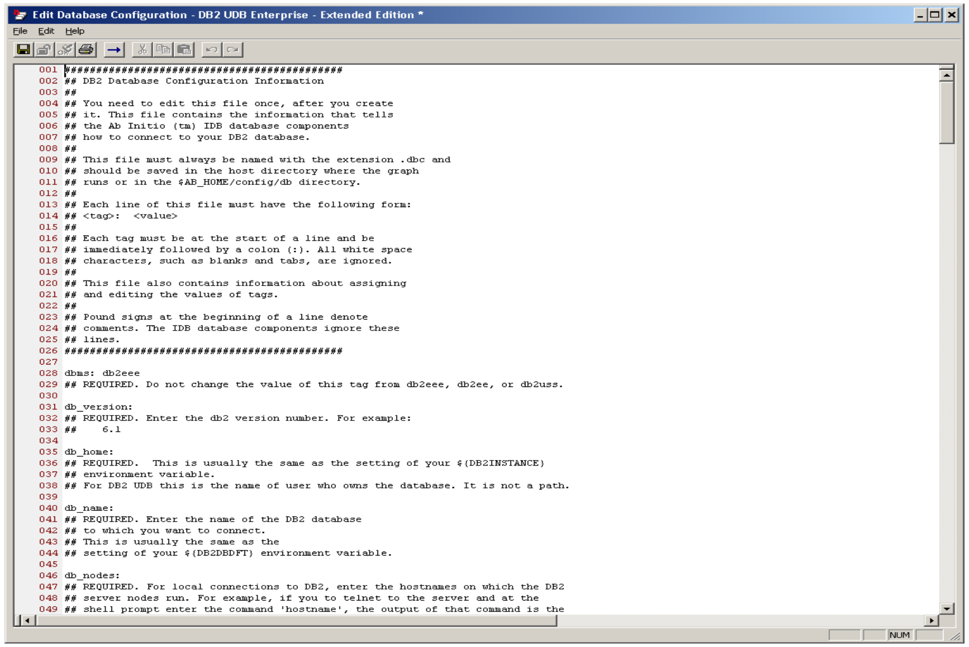
Select Database

Please select a database:

Adabas
DB2 for OS/390
DB2 UDB Enterprise Edition
[✓ DB2 UDB Enterprise - Extended Edition]
DB/400 on iSeries via DB2/Connect
DB/400 on iSeries via ODBC
IBM IMS
CA-IDMS
Informix
Microsoft Access
Microsoft Excel
Microsoft SQL Server
Microsoft ODBC

Database Connectivity Cont

- Click OK. The Edit Database Configuration editor appears containing database-specific configuration information.



Edit Database Configuration - DB2 UDB Enterprise - Extended Edition *

```
#############################################
## DB2 Database Configuration Information
##
## You need to edit this file once, after you create
## it. This file contains the information that tells
## the Ab Initio (tm) IDB database components
## how to connect to your DB2 database.
##
## This file must always be named with the extension .dbc and
## should be saved in the host directory where the graph
## runs or in the $AB_HOME/config/db directory.
##
## Each line of this file must have the following form:
## <tag>:  <value>
##
## Each tag must be at the start of a line and be
## immediately followed by a colon (:). All white space
## characters, such as blanks and tabs, are ignored.
##
## This file also contains information about assigning
## and editing the values of tags.
##
## Pound signs at the beginning of a line denote
## comments. The IDB database components ignore these
## lines.
#############################################

dbms: db2eee
## REQUIRED. Do not change the value of this tag from db2eee, db2ee, or db2uss.

db_version:
## REQUIRED. Enter the db2 version number. For example:
##    6.1

db_home:
## REQUIRED.  This is usually the same as the setting of your ${DB2INSTANCE}
## environment variable.
## For DB2 UDB this is the name of user who owns the database. It is not a path.

db_name:
## REQUIRED. Enter the name of the DB2 database
## to which you want to connect.
## This is usually the same as the
## setting of your ${DB2DBDFT} environment variable.

db_nodes:
## REQUIRED. For local connections to DB2, enter the hostnames on which the DB2
## server nodes run. For example, if you to telnet to the server and at the
## shell prompt enter the command 'hostname', the output of that command is the
```

- Follow the comments in the configuration file to fill in required or other fields as necessary
- Close the editor and save the file in the following location: **Sandbox→db folder→dbc file**

Database Connectivity Test

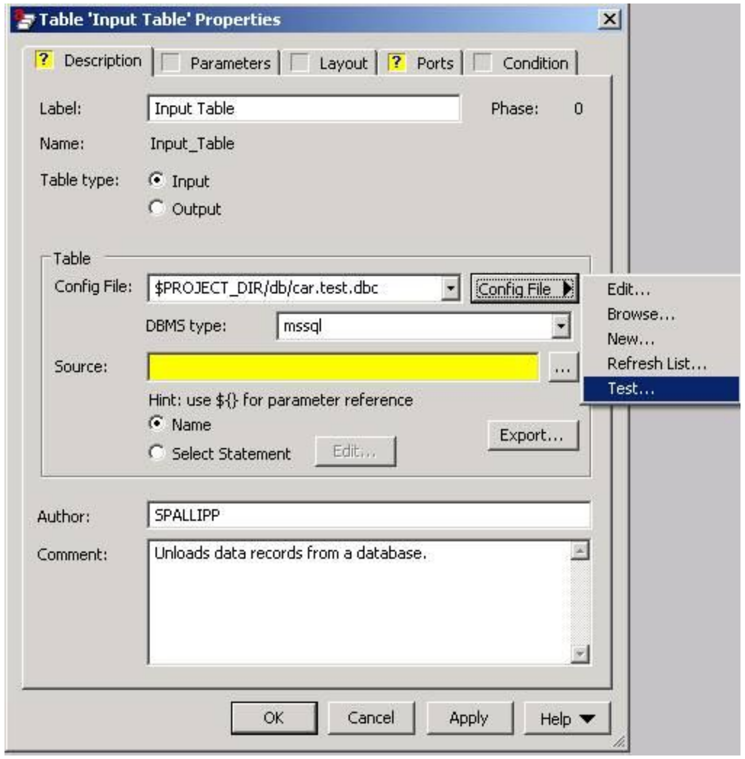
- The database connectivity (using dbc file) can be tested by two ways:
  - Using "**m_db test**" utility command in command prompt
  - Using Database Components (Input/Output table)

- Using "**m_db test**" utility command in command prompt
  - It is possible to test any database connectivity (using dbc file) without having to create a graph or using GDE using utility command
  - Go to the location where dbc file is saved and type in following command

    m_db test <dbc_filename>

Database Connectivity Test Cont

- Using Database Components (Input/Output table)
  - The database connectivity can be tested using the Database Components in GDE as follows:



Label: Input Table
Phase: 0
Name: Input_Table
Table type: (● Input) ○ Output
Table
Config File: $PROJECT_DIR/db/car.test.dbc
DBMS type: mssql
Source:
Hint: use ${} for parameter reference
(● Name) ○ Select Statement
Author: SPALLIPP
Comment: Unloads data records from a database.

Custom Components

- A custom component can be termed as:
  - A component we build from scratch to execute our own existing program or shell script
  - An Ab Initio built-in component customized and configured in a particular way and saved for reuse
  - A subgraph constructed with built-in components and saved for reuse
- A custom component consists of two elements
  - Program or shell script
  - A program specification file describing the program's command-line arguments, ports, parameters, and other attributes

Custom Components Cont

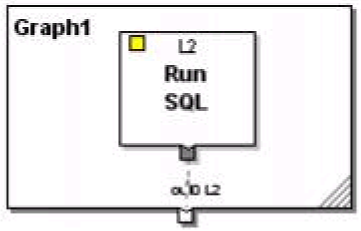
- When the GDE generates the script for a graph, a custom component appears in that script as a line beginning with mp custom
- In a shell script, you write an mp custom line to call a custom component.
- To see examples of mp lines generated by the GDE for Ab Initio built-in components:
  - In the GDE, place a component in the workspace.
  - On the GDE menu bar, choose **Edit > Script > Generated Script.**
  - If the **Errors were detected during compilation** message box appears, click **No**. **The Edit MP script** window opens.
  - Find the line in the script that begins with **mp** component name.

Creating Custom Components from Subgraph

- To create a custom component using an outer frame:
  - In the GDE, open the common sandbox where the component will be saved. (**Choose Project > Open Sandbox**.)
  - Create a **components folder** in this sandbox, and a sandbox parameter with which to reference the folder location
  - From the GDE menu bar, choose **File > New** to open a new graph
  - From the GDE menu bar, choose **View > Outer Frame**
  - A frame appears in your graph workspace. Each component you drag in gets dropped inside this frame. When you save the graph, you are actually saving a subgraph that will become your custom component.

Creating Custom Components from Subgraph

- From the Component Organizer, drag in the components on which you want to base your custom component. For example, drag in the Run SQL component.
- Attach any ports to the edge of the frame inside the graph canvas For example, attach the Run SQL log port to one of the edges of the outer frame:



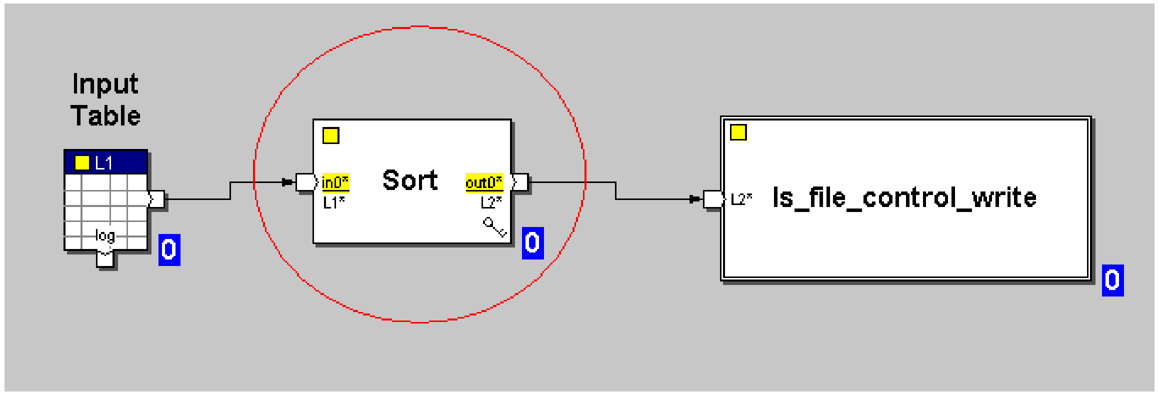
- Configure the components as needed, and then save the graph in the components folder, using a file extension of **.mp**. Check in the file to the EME Technical Repository.
- To use the new component, drag it from the sandbox onto the graph canvas.

Performance Tuning

- Very often, poor performance is the result of bad design decisions. Once the application is in production it is usually too late to hope for drastic performance improvements without some redesign work. So the best time to work on optimizing the performance of a graph is while it is still in design and development.

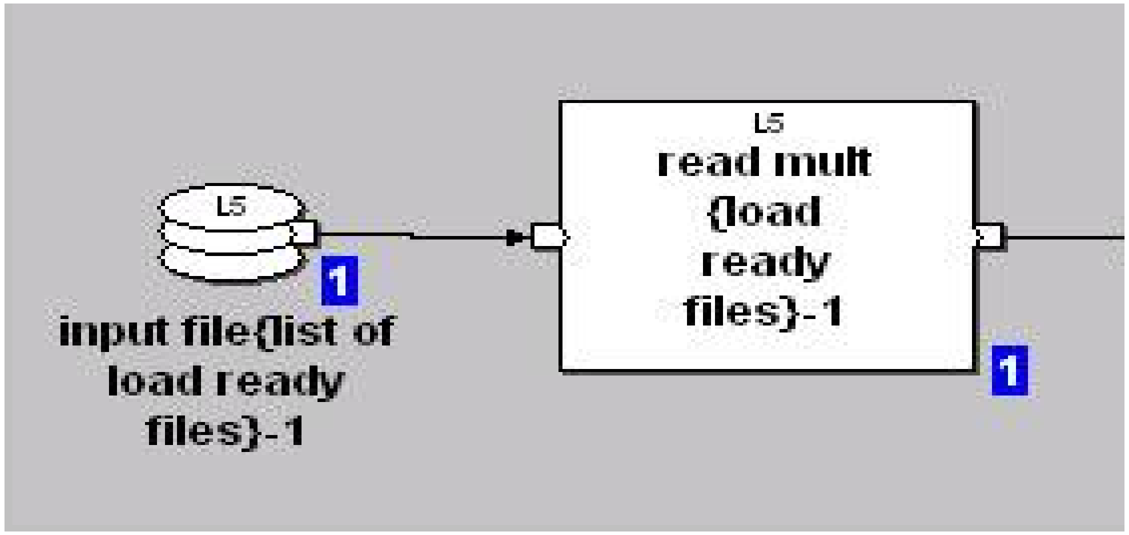
Performance Tuning During Development Cont

- **Less computation on databases.**
- There are many things that you should do outside the database. Operations involving heavy computation are usually better done with components, in the graph, rather than in a database. For example, Sorting will almost always be faster when you use the Sort component rather than sorting in the database.



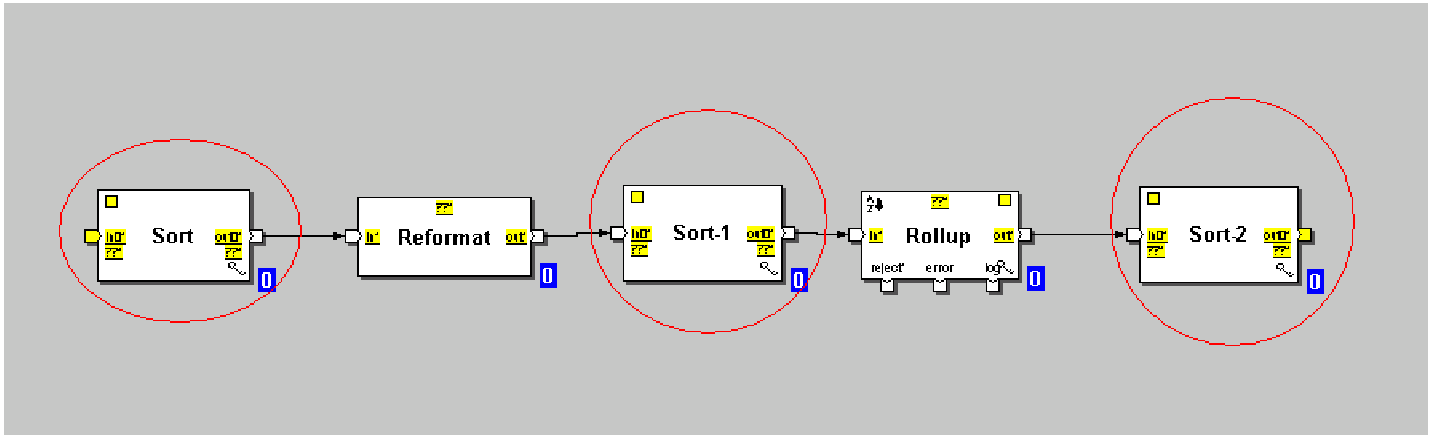
Performance Tuning During Development Cont

- **Having lesser data per run**.
- If we have lesser data per run, the graph's startup time will be large in relation to the actual run time. Thus we cannot have optimized performance.
- For example: Instead of reading many small files many times, we can directly use “Read Multiple Files” one time.


L5
input file{list of load ready files}-1
1
L5
read mult {load ready files}-1
1

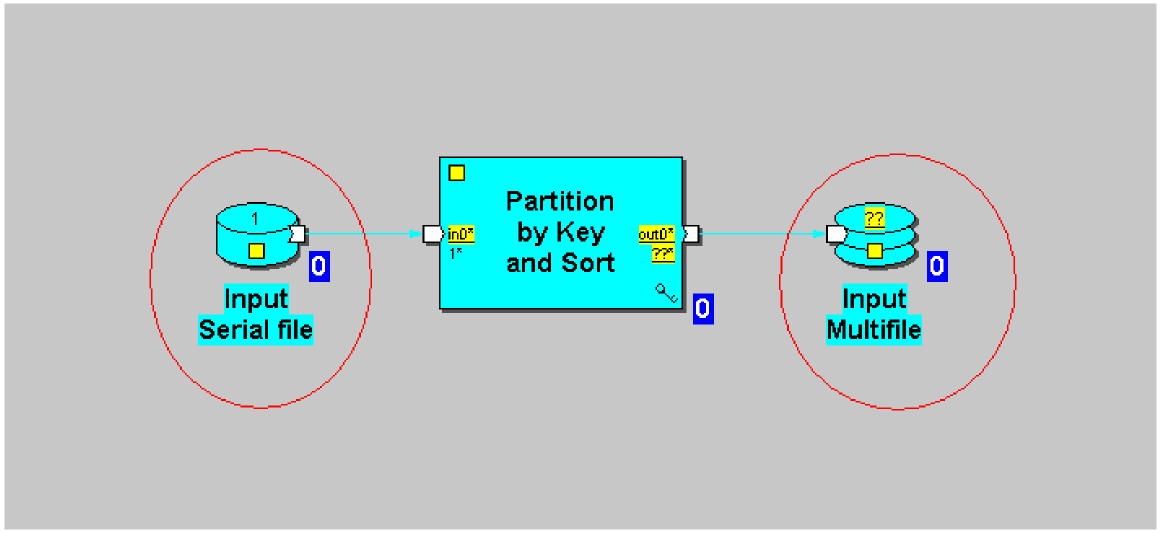
Performance Tuning During Development Cont

- **Too many sorts.**
- The “SORT” component breaks pipeline parallelism and causes additional disk I/O to happen. So we should avoid using this component multiple times



Performance Tuning During Development Cont

- **Using Multifiles.**
- We can enhance our graph performance by using a multifile instead of using large single file. We can partition our large file into smaller files among several disks so that we can use them in parallel for reading and writing purposes.



Performance Tuning During Development Cont

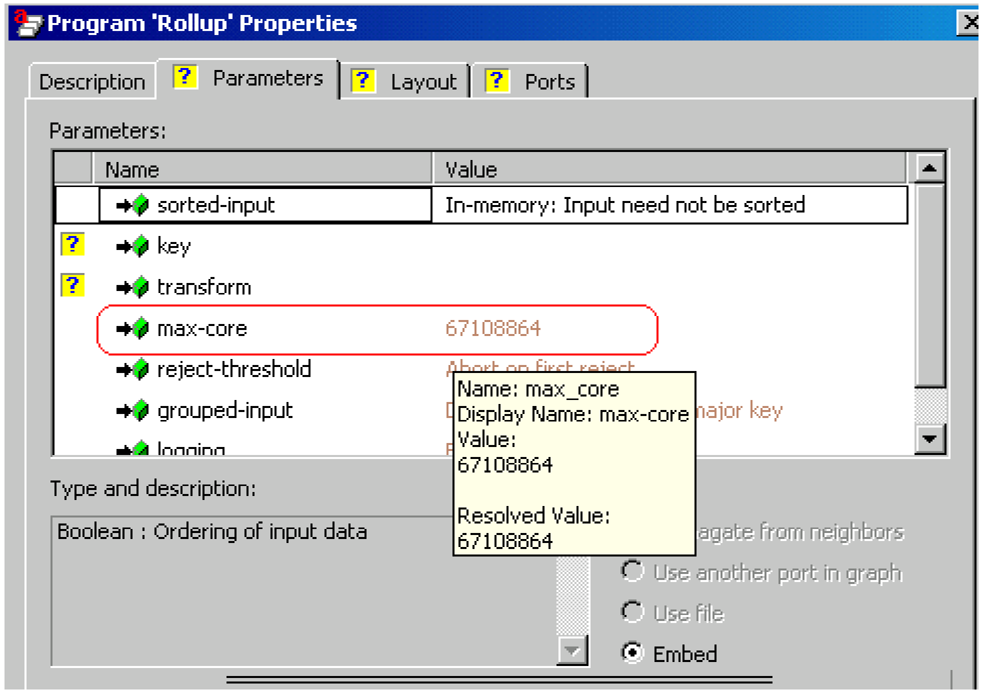
- **Wrong placement of phase breaks.**
- Wherever a phase break occurs in a graph, the data in the flow is written to disk; it is then read back into memory at the beginning of the next phase. So we should always keep into consideration that we should place the phase break at appropriate flow where we have comparatively lesser data to be written.
- For example, putting a phase break just before a “Filter By Expression” is a bad idea since after using this component the size of data may get reduced.
- **Large Phases.**
- The highest memory consumption of a graph is determined by the memory consumption of its “largest” phase. So making the phase smaller (by reducing the number of components in it) will reduce the amount of memory it requires.

Performance Tuning During Development Cont

- **Data parallelism.**
- Each additional partition of a component requires space for the additional component processes in memory, reducing the number of partitions reduces the number of component processes. So we should always try not to have any unnecessary partitions.
- For Example: If you have x number of processors available, reducing data parallelism to less than x will also reduce CPU demand by the number of partitions you have eliminated

Performance Tuning During Development Cont

- **Max-Core Values.**
- The max-core parameter allows you to specify the maximum amount of memory that can be allocated for the component. We should use the MAX_CORE value efficiently


Program 'Rollup' Properties

Parameters:

Name	Value
sorted-input	In-memory: Input need not be sorted
key	
transform	
max-core	67108864
reject-threshold	Abort on first reject
grouped-input	major key
logging	

Name: max_core
Display Name: max-core
Value:
67108864

Resolved Value:
67108864

Type and description:

Boolean : Ordering of input data

Performance Tuning During Development Cont

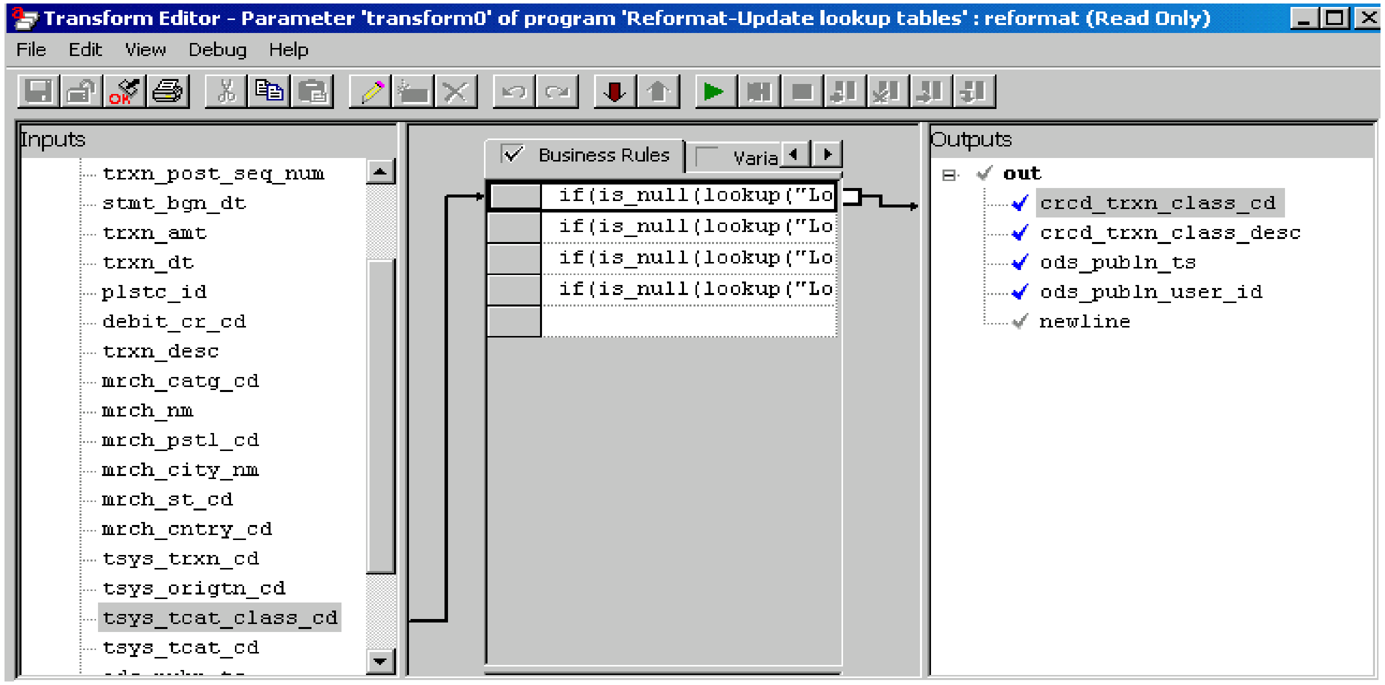
- Giving Low max-core: Giving max-core, a value less than what the component needs to get the job done completely in memory will result in the component writing more temporary data to disk at runtime. This can cause slow performance.
- Giving High max-core: If max-core is set high enough that your graph's working set no longer fits in physical memory, the computer will have to start paging simply to run the graph. This will certainly have an adverse effect on the graph's performance

- **Parallel Processing**

- Use data parallel processing wherever possible. Make a graph parallel as early as possible and keep it parallel as long as possible

Performance Tuning Related to Data

- **Reduce data volumes early in the processing.**
- Drop unneeded rows early in the graph.
- Drop unneeded fields early in the graph


Transform Editor - Parameter 'transform0' of program 'Reformat-Update lookup tables' : reformat (Read Only)

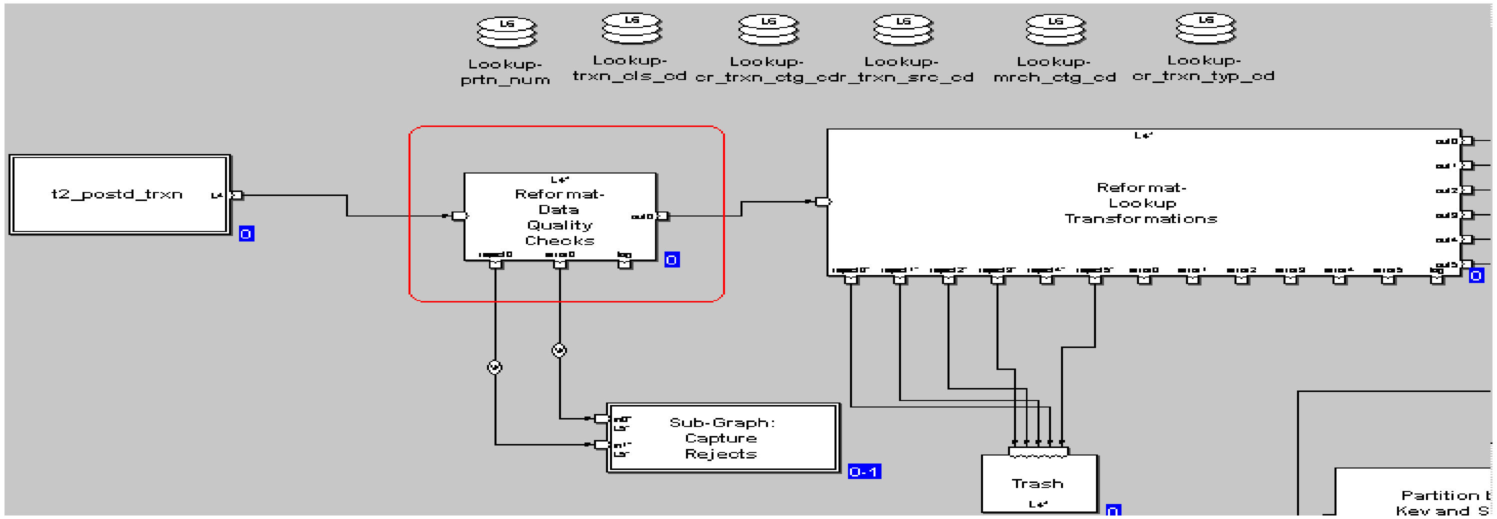
```
Inputs:
trxn_post_seq_num
stmt_bgn_dt
trxn_amt
trxn_dt
plstc_id
debit_cr_cd
trxn_desc
mrch_catg_cd
mrch_nm
mrch_pstl_cd
mrch_city_nm
mrch_st_cd
mrch_cntry_cd
tsys_trxn_cd
tsys_origtn_cd
tsys_tcat_class_cd
tsys_tcat_cd

Business Rules:
if(is_null(lookup("Lo
if(is_null(lookup("Lo
if(is_null(lookup("Lo
if(is_null(lookup("Lo

Outputs:
out
  crcd_trxn_class_cd
  crcd_trxn_class_desc
  ods_publn_ts
  ods_publn_user_id
  newline
```

Performance Tuning Related to Data

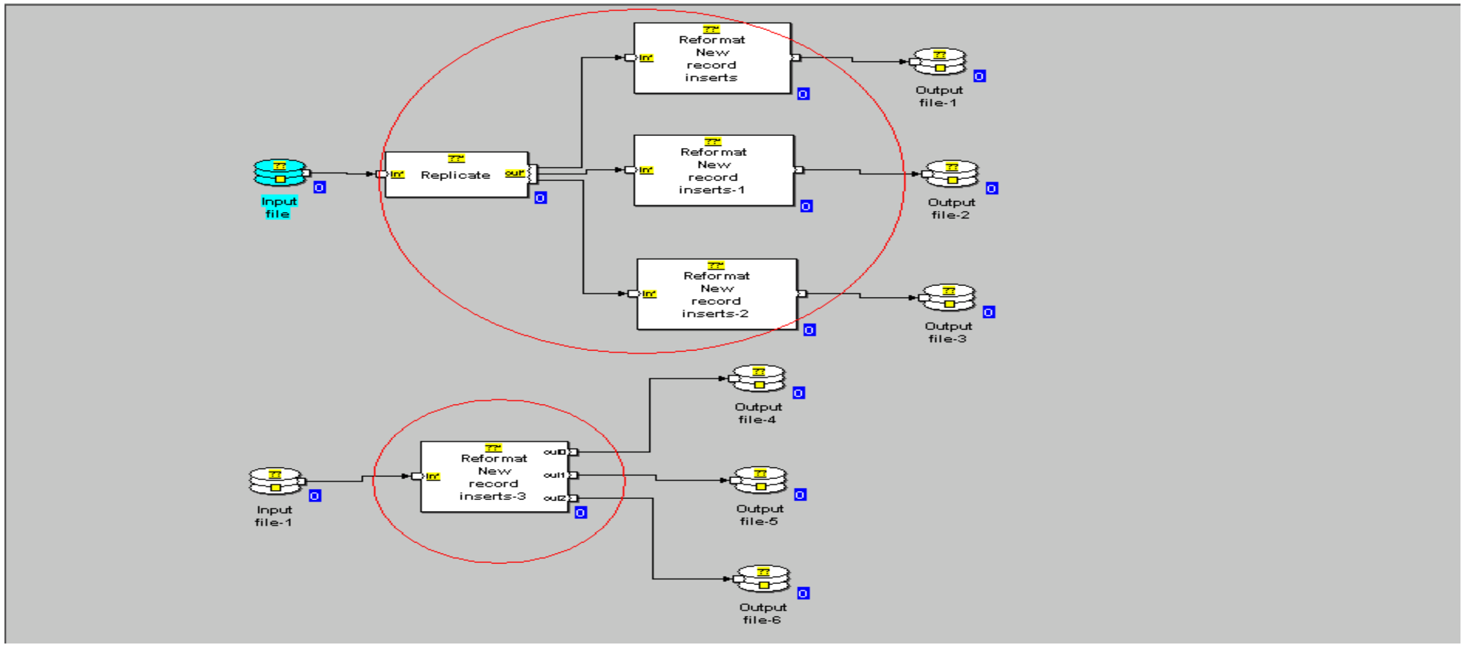
- **Handle data quality issues as soon as possible, since it will reduce the data which is unnecessary. Do not spread the data quality rules throughout the graph until necessary.**



- **Expand data as late as possible.**
- Use of components like Replicate, Normalize, and Join tend to increase the volume of data. So, these components should be used with the minimum volume of data, as far as possible

Performance Tuning Related to Components

- **Use reformat with multiple output ports instead of replicating and using multiple reformats. Instead of too many Reformat component consecutively one after the other use output indexes parameter in the first Reformat component and mention the Condition there.**



Performance Tuning Related to Components

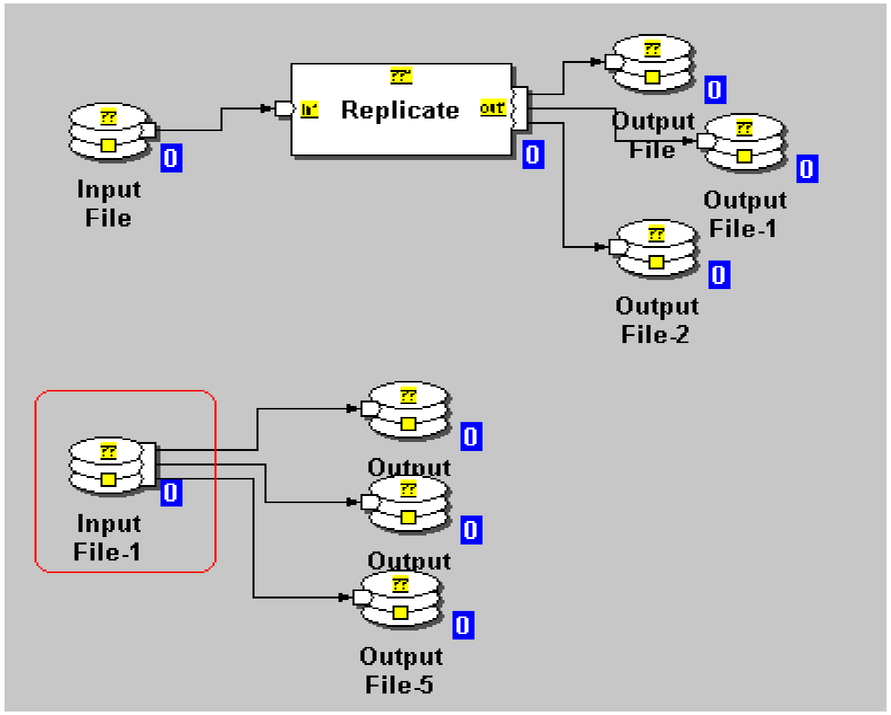
- **Use gather instead of concatenate until necessary, and also if any component which supports gather, then omit gather component**
- For joining records from 2 flows use Concatenate component only when there is a need to follow some specific order in joining records. If no order is required then it is preferable to use Gather component.
- **Try to sort the data in parallel by using Partition by Key and Sort, rather than sorting it serially**
- **Use Sort Within Groups, to avoid complete re sorting of data.**
- Use Sort Within Groups, if you have sorted your data by a key and later you wish to refine your results by sorting on some minor key. This gives a better performance, than using a second incremental sort.

Performance Tuning Related to Components

- **Never put a checkpoint or a phase after sort, instead use check pointed sort**
- Since after sorting the sorted data will be written to the disk and after putting phase or checkpoint the data again will be written to the disk. Check pointed sort separates the initial sort from the final merge, and puts a checkpoint between them. The result is that only one copy of the data will be stored on disk.
- **Never put a checkpoint or a phase after the Replicate component unless it is necessary.**
- Since, it will write the replicated data to disk. Rather we can put a phase or a checkpoint before Replicate.

Performance Tuning Related to Components

- **Connect multiple flows directly into an Input file**
- If we need to process a file in more than one way, we should connect multiple flows directly to the file instead of using replicate components



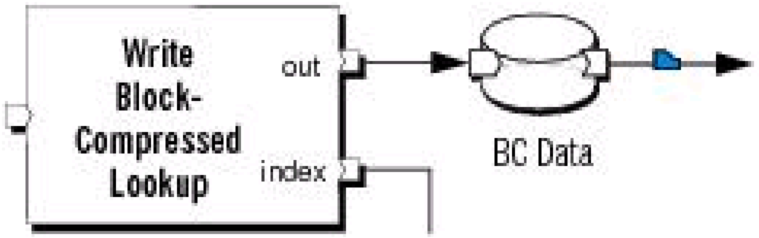
- **Try not to embed record formats, if they are to be reused.**

Index Compressed Flat Files(ICFF)

- **Advantages of ICFF are:**
- Disk requirements — Because ICFFs store compressed data in flat files without the overhead associated with a DBMS, they require much less disk storage capacity than databases on the order of 10 times less.
- Memory requirements — Because ICFFs organize data in discrete blocks, only a small portion of the data needs to be loaded in memory at any one time.
- Speed — ICFFs allow you to create successive generations of updated information without any pause in processing. This means the time between a transaction taking place and the results of that transaction being accessible can be a matter of seconds.
- Performance — Making large numbers of queries against database tables that are continually being updated can slow down a DBMS. In such applications, ICFFs outperform databases.
- Volume of data — ICFFs can easily accommodate very large amounts of data. In fact, that it can be feasible to take hundreds of terabytes of data from archive tapes, convert it into ICFFs, and make it available for online access and processing.

How to read ICFF data

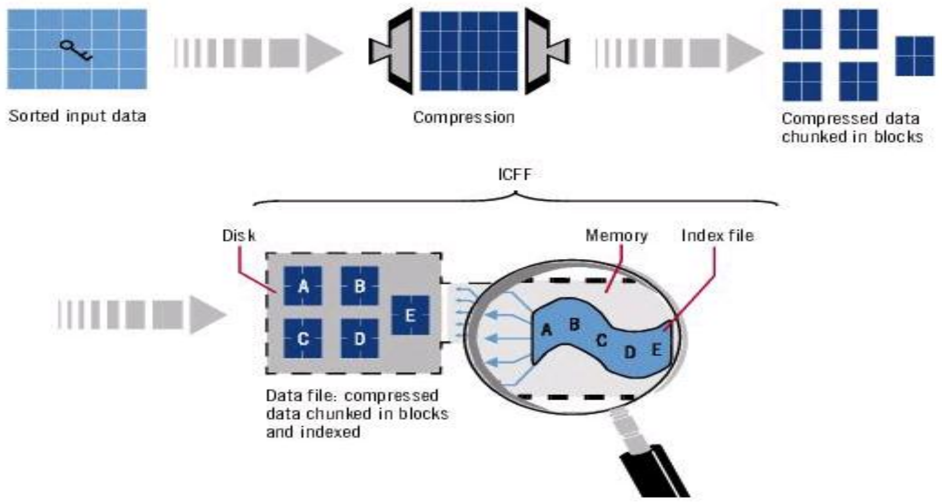
- We can “read” ICFF data in the sense of loading it from disk, or in the sense of uncompressing and directly examining an ICFF data file’s contents.
- **Loading ICFF data into a graph**
- Here we need to write a transform that includes one or more lookup functions.
- **Directly examining an ICFF data file**
- Attach an intermediate file to the out port of the Write Block-Compressed Lookup component:



- Define the intermediate file’s output (read) port to take its record format from the in port of Write Block-Compressed Lookup.

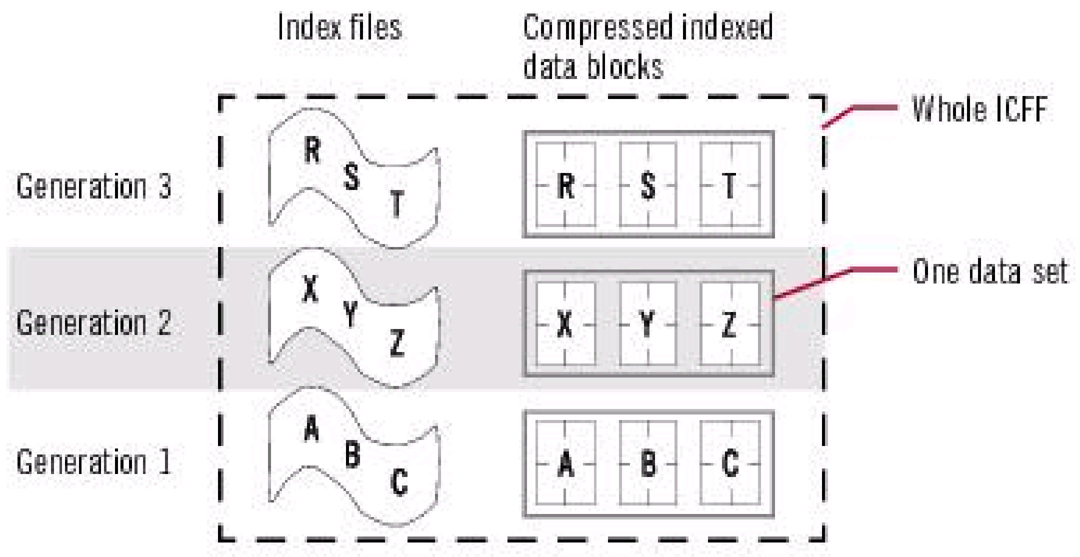
How indexed compressed flat files work

- To create an ICFF, we need presorted data. WRITE BLOCK-COMPRESSED LOOKUP component, compresses and chunks the data into blocks of more or less equal size. The graph then stores the set of compressed blocks in a data file, each file being associated with a separately stored index that contains pointers back to the individual data blocks. Together, the data file and its index form a single ICFF.
- A crucial feature is that, during a lookup operation, most of the compressed lookup data remains on disk — the graph loads only the relatively tiny index file into memory.


Sorted input data
Compression
Compressed data chunked in blocks
ICFF
Disk
Memory
Index file
A
B
C
D
E
Data file: compressed data chunked in blocks and indexed

Generations

- Addition of data to an ICFF is possible even while it is being used by a graph. Each chunk of added update data is called a generation. Each generation is compressed separately; it consists of blocks, just like the original data, and has its own index, which is simply concatenated with the original index.



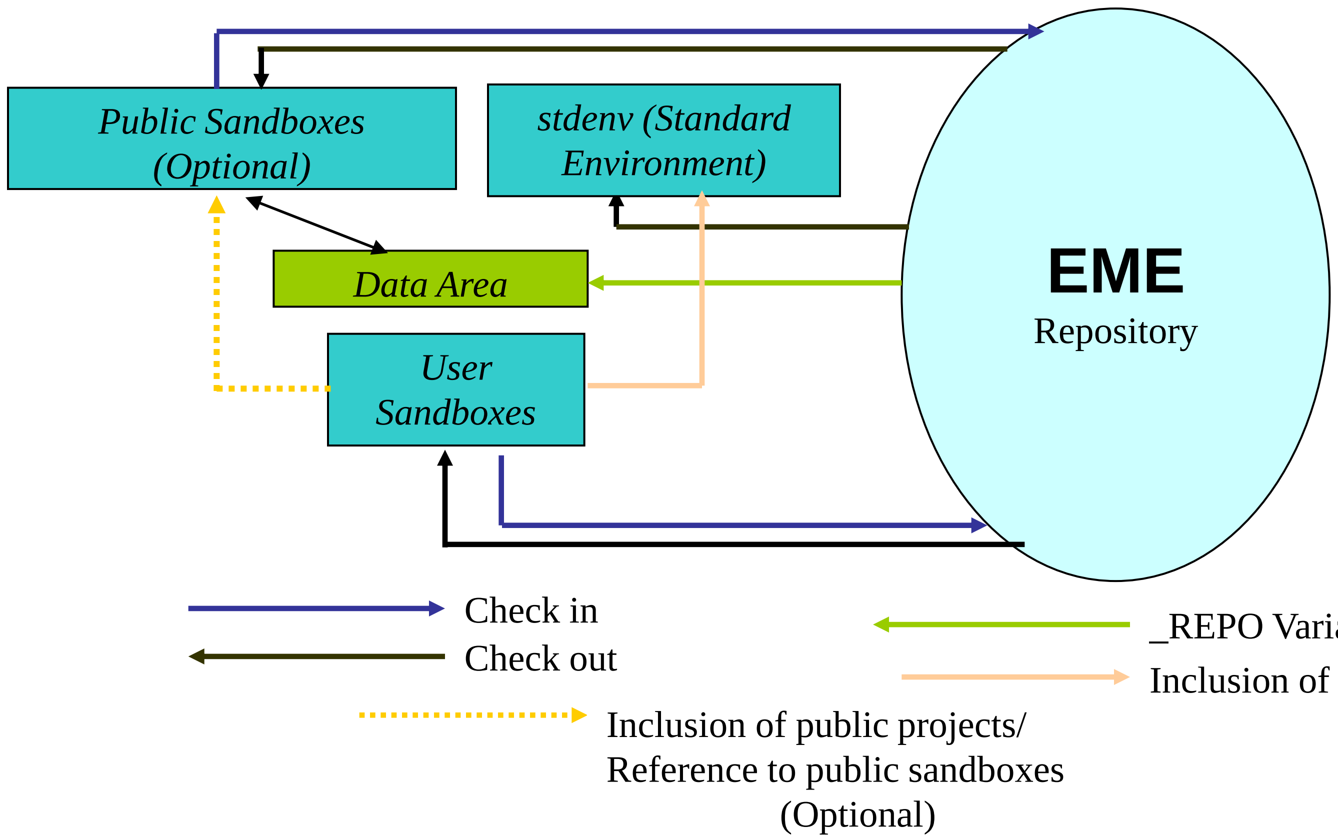
How Generations are created

- As an ICFF generation is being built, the ICFF building graph writes compressed data to disk as the blocks reach the appropriate size. Meanwhile, the graph continues to build an index in memory.
- In a batch graph, an ICFF generation ends when the graph or graph phase ends. In a continuous graph, an ICFF generation ends at a checkpoint boundary.
- Once the generation ends, the ICFF building graph writes the completed index to disk.

EME(Enterprise Meta Environment)

- EME is the version control system for the AbInitio graphs and meta data
- Integrated with GDE for easy developer access
- Can be interacted through Management Console, UNIX shell, Web interface and GDE
- EME is an object oriented data storage system that version controls and manages various kinds of information associated with Ab Initio applications, which may range from design information to operational data. In simple terms, it is a repository, which contains data about data – metadata.

Environment Structure



Public Sandboxes (Optional)
stdenv (Standard Environment)
Data Area
User Sandboxes
EME
Repository
Check in
Check out
_REPO Variables
Inclusion of stdenv
Inclusion of public projects/ Reference to public sandboxes (Optional)

EME Administration

- Use an administrative login for this purpose (emeadmin)
- Creating the EME
  - export AB_AIR_ROOT= /<mount>/<EME location>
  - air repository create /<mount>/<EME location>
- Start/shutdown EME
  - air repository start
  - air repository shutdown
- Verify EME
  - air ls
  - http://<eme-server>/abinitio/ (requires EME web server configuration using install-aiw)

EME Administration Cont..

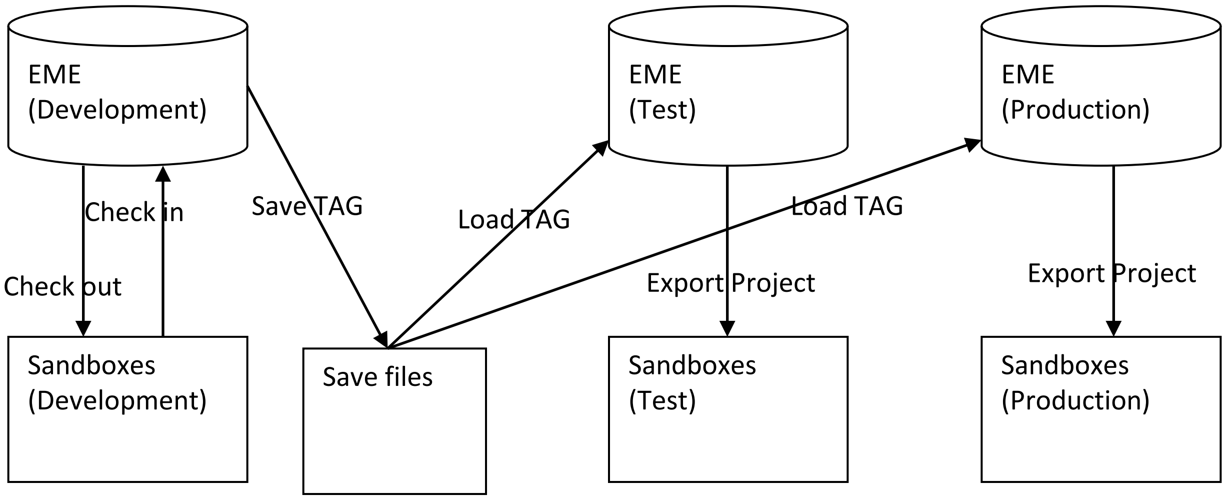
- Backing up and restoring the EME
- Option 1 – offline
  - Should be restored onto a machine with same hardware architecture
  - air repository backup tar cvf <eme-backup.tar> ‘{}’
  - tar –xvf <eme-backup.tar>
- Option 2 – offline
  - Can be restored onto any UNIX platform
  - air repository create-image <eme.img.gz> -compress
  - air repository load-from-image ${AB_AIR_ROOT} <eme.img.gz>
- Option 3 - online
  - Should be restored onto a machine with same hardware architecture
  - air repository online-backup start <file name>
  - air repository online-backup restore <file name> <new eme root>

Standard Environment

- Stdenv project
    - Builds the basic infrastructure and environment for running AbInitio applications.
    - Contains the SERIAL and MFS locations, error/tracing levels, narrow/medium/wide MFS paths etc.
    - Contains enterprise level parameters and values that will be used by private projects.
    - stdenv project is provided by AbInitio
    - Every project includes stdenv project and inherit the parameters defined

Tagging Process

Tagging and promotion process flow



Tagging Process Cont.

- Tagging using air tag create
  - air tag create -project-only TestProject.01.00.00 /Projects/bi/TestProject
- Tagging using import-configuration
  - air tag import-configuration /users/emeadmin/cfg/TestProject.01.00.00.config /Projects/bi/TestProject/cfg/TestProject.01.00.00.config
  - air tag tag-configuration TestProject.01.00.00 /Projects/bi/TestProject/cfg/TestProject.01.00.00.config
  - air object save /users/emeadmin/save/TestProject.01.00.00.save -exact-tag TestProject.01.00.00 -external local -external common -settings local -no-annotations
  - gzip -f /users/emeadmin/save/TestProject.01.00.00.save

Tagging Process Cont.

- Loading tag in to EME
    - air object load -table-of-contents /users/emeadmin/save/TestProject.01.00.00.save
    - air object load /users/emeadmin/save/TestProject.01.00.00.save

- Exporting a project
    - air project export /Projects/bi/TestProject –basedir

/users/emeadmin/sand/bi/TestProject.01.00.00 -from-tag

TestProject.01.00.00 -create –cofiles -common

/Projects/bi/stdenv

/users/emeadmin/sand/bi/stdenv.01.01.00

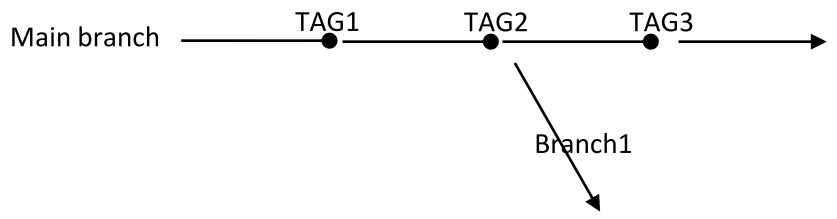
Tagging Process Cont.

- Useful commands

  - air tag list - lists the tags in EME
  - air tag list -p TestProject.01.00.00- list the primary objects
  - air tag list -e TestProject.01.00.00 - list all objects
  - air tag delete TestProject.01.00.00- delete the tag
  - air promote save ..
  - air promote load ..

Branching

- Branching
  - Create branches in EME for bug-fixes etc.



- Commands
  - export AB_AIR_ROOT=//<eme server>/<eme path>
  - air branch create <branch name> [–from-branch <parent branch>] [-from-version <branch version | tag>]
  - air branch list
  - air branch delete <branch-name>
  - export AB_AIR_BRANCH=<branch name>

Business Rules Environment(BRE)

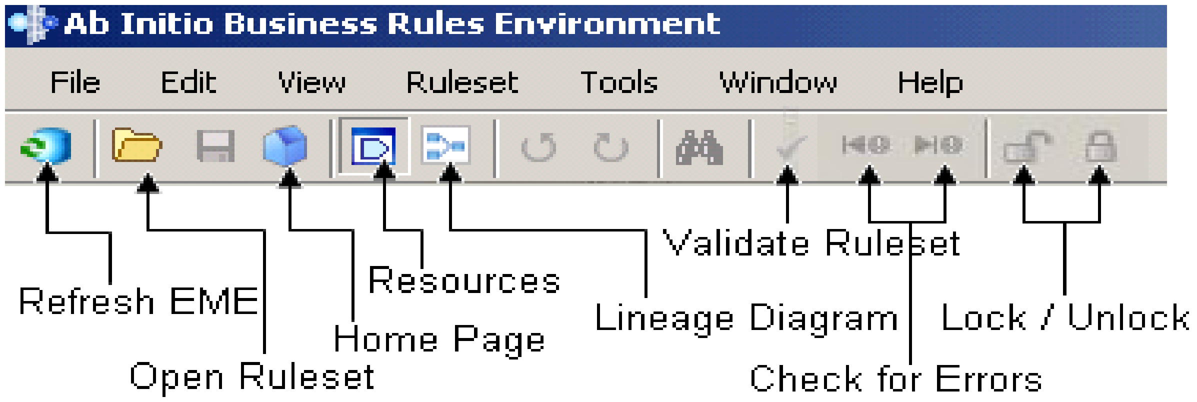
- BRE is the Business Rule Environment and this product was launched with GDE 1.15, but needs a separate license (not the same license as GDE).
- This is new way of creating XFR's with ease from mapping rules written in English
- Even business team also can understand these generic transformation rules
- Rules are written on spreadsheet kind of page and then converted to component by a click on button
- With BRE we can avoid writing of the same validation twice.

Advantages of BRE

- BRE is another console from AbInitio and which needs a separate license. This provides the Business users as well as the GDE developers to develop and implement the Business Rules
- The time taken to implement the rules is getting reduced
- This tool is more transparent for Business Users/Analysts as well as developers
- The traceability is also another benefit for BRE.

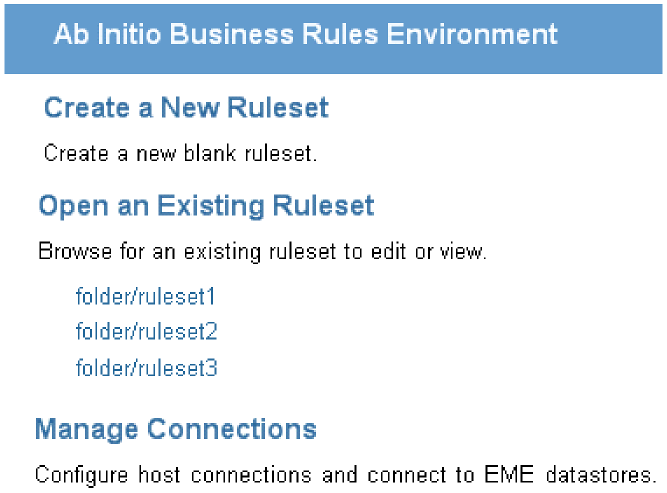
Starting With BRE

- To start BRE from the Windows Start Menu option under the AbInitio folder. Make a host and data store connection with EME similar to the GDE connection.



Starting With BRE Cont

- Once we logged into BRE, there are links to create a new Ruleset, Open an existing Ruleset as shown below



Ab Initio Business Rules Environment

Create a New Ruleset

Create a new blank ruleset.

Open an Existing Ruleset

Browse for an existing ruleset to edit or view.

folder/ruleset1
folder/ruleset2
folder/ruleset3

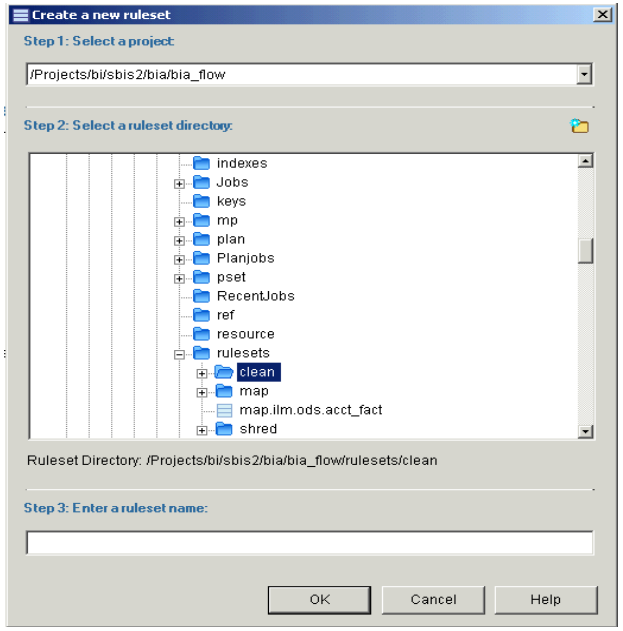
Manage Connections

Configure host connections and connect to EME datastores.

- Please refer to BRE help file for the icons and symbols used in Rulesets.

Starting With BRE Cont

- **Create a New Ruleset** : There are 3 steps involved for creating a new ruleset :
- Open the project path of EME
- Select the ruleset directory/subdirectory.
- Name the ruleset


Create a new ruleset

Step 1: Select a project:

/Projects/bi/sbis2/bia/bia_flow

Step 2: Select a ruleset directory:

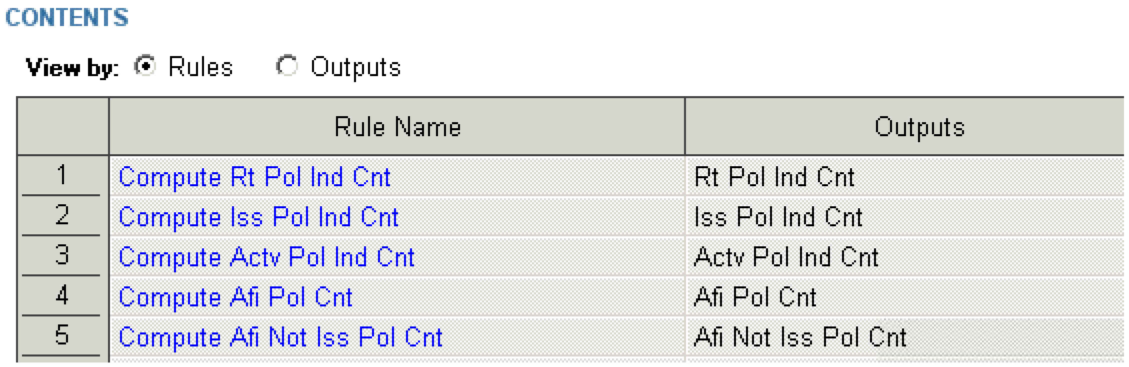
```
indexes
Jobs
keys
mp
plan
Planjobs
pset
RecentJobs
ref
resource
rulesets
    clean
    map
    map.ilm.ods.acct_fact
    shred
```

Ruleset Directory: /Projects/bi/sbis2/bia/bia_flow/rulesets/clean

Step 3: Enter a ruleset name:

Starting With BRE Cont

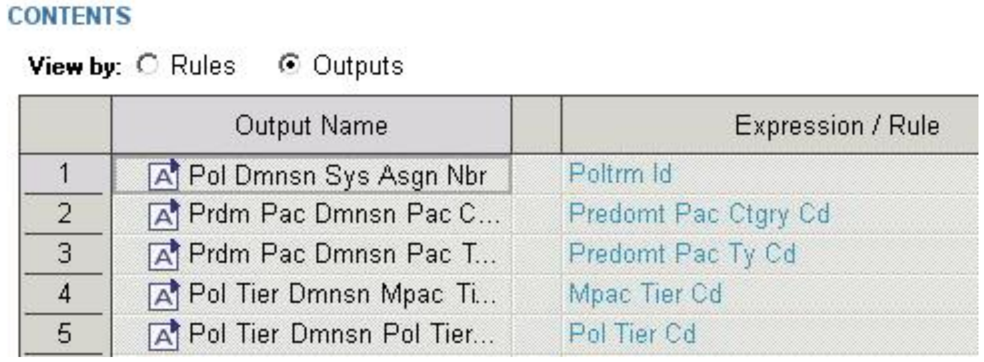
- **Open an Existing Ruleset**: Open the existing ruleset from the main Project directory. Once you open the rule set, you may view it in two ways, either by Rules or by Output.



CONTENTS

View by: [✓ Rules] Outputs

	Rule Name	Outputs
1	Compute Rt Pol Ind Cnt	Rt Pol Ind Cnt
2	Compute Iss Pol Ind Cnt	Iss Pol Ind Cnt
3	Compute Actv Pol Ind Cnt	Actv Pol Ind Cnt
4	Compute Afi Pol Cnt	Afi Pol Cnt
5	Compute Afi Not Iss Pol Cnt	Afi Not Iss Pol Cnt



CONTENTS

View by: Rules [✓ Outputs]

	Output Name		Expression / Rule
1	Pol Dmnsn Sys Asgn Nbr		Poltrm Id
2	Prdm Pac Dmnsn Pac C...		Predomt Pac Ctgry Cd
3	Prdm Pac Dmnsn Pac T...		Predomt Pac Ty Cd
4	Pol Tier Dmnsn Mpac Ti...		Mpac Tier Cd
5	Pol Tier Dmnsn Pol Tier...		Pol Tier Cd

View the Ruleset by Rules and Output.

BRE Rulesets

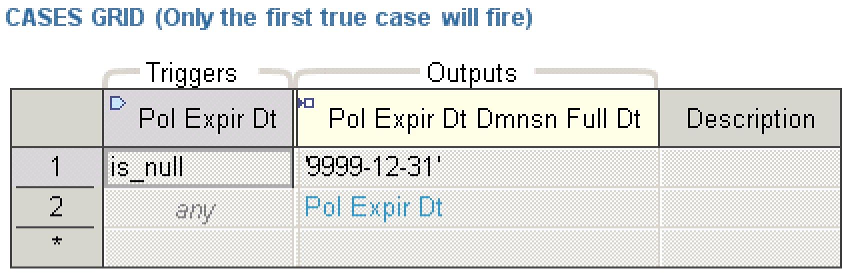
- The object which is created by BRE is ruleset, which consists of one or more closely related generic transformation rules. This computes a value for one field in the output record.
- The ruleset generally contains:

i. Input and Output datasets
ii. Lookup files
iii. Other set of Business rules with repetitive actions.
iv. Parameters whose values are determined upon running the graph
v. Special formulae and functions

- The rule consists of cases and each case is having a set of conditions which determines one of the possible values for the output field

BRE Rulesets Cont

- The case grid, as shown below contains set of conditions (in Triggers) and determines the value of output (in Outputs) based on the input value.



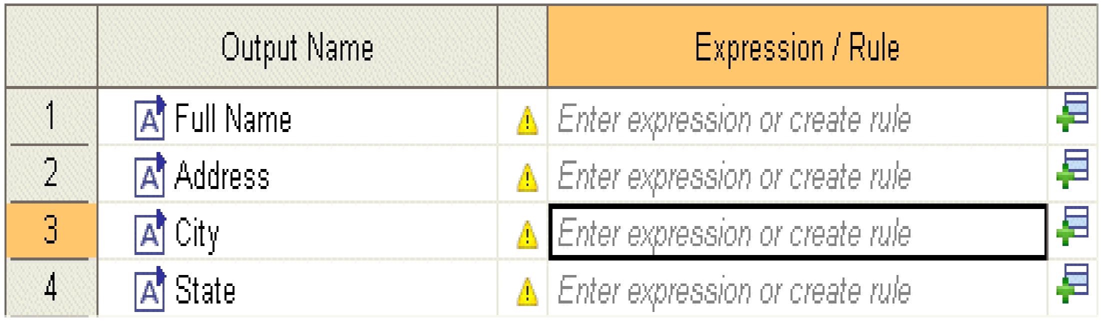
CASES GRID (Only the first true case will fire)

	Triggers	Outputs	
	Pol Expir Dt	Pol Expir Dt Dmnsn Full Dt	Description
1	is_null	'9999-12-31'	
2	any	Pol Expir Dt	
*			

- In BRE, there are three types of rulesets as explained below:
- **Reformat Rulesets**: This ruleset takes the input one by one and apply the transformation and produces the output.
- **Filter Rulesets**: This rulesets reads the input and based on the conditions specified, either keep or discard the record and given the value to the output variable
- **Join Rulesets**: Reads the inputs from multiple sources and combine them, then apply the transformation specified in ruleset and move the value to the output.

Creating and Validating Rules in Ruleset

- Upon loading the dataset, the mapping of technical names to business names is listed in the Input and output sections. Looking at the schema for the EME, relationship between physical name (i.e. within the DML) to logical and then to a business name can be seen.
- After opening the ruleset, click Outputs in the Content area on the Ruleset tab. As shown below

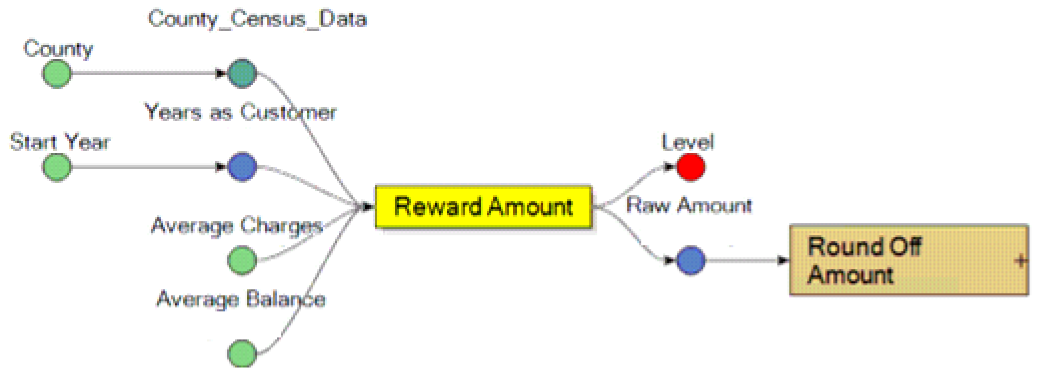


	Output Name		Expression / Rule	
1	Full Name	⚠	Enter expression or create rule	
2	Address	⚠	Enter expression or create rule	
3	City	⚠	Enter expression or create rule	
4	State	⚠	Enter expression or create rule	

- Click the icon to add a new rule/expression in the ruleset for the required columns. For example, to get the Full Name, we need to join the First Name and Last name columns with a space separator as First Name +” ”+ Last Name

Lineage Diagrams

- The lineage diagrams are the method to represent the data flows between the rulesets elements. It's possible to represent the lineage diagram as a diagnostic tool for a ruleset, a rule or lookup or for a I/O variable



- To create a lineage diagram from a ruleset, choose the ruleset > lineage diagram. From the lineage diagram, view menu choose the entire ruleset.
- To create a lineage diagram from a rule, choose the rule > click the icon on the toolbar, on the ruleset tab content area create

XML Processing

For common and custom format XML processing

The following components are available to help with the bulk of XML processing requirements:

Component	Description
READ XML	Reads a stream of characters, bytes, or records; then translates the data stream into DML records.
READ XML TRANSFORM	Reads a record containing a mixture of XML and non-XML data; transforms the data as needed, and translates the XML portions of the data into DML records.
WRITE XML	Reads records and translates them to XML, writing out an XML document as a string.
WRITE XML TRANSFORM	Translates records or partial records to a string containing XML.

XML Processing

For common XML processing only

You use the following components and utilities only in conjunction with the common XML processing approach:

Component or utility	Description
XML SPLIT	Reads, normalizes, and filters hierarchical XML data. This component is useful when you need to extract and process subsets of data from an XML document.
xml-to-dml	Derives the DML-record description of XML data. You access this utility primarily through the **Import from XML** dialog, though you can also run it directly from a shell.
Import from XML	Graphical interface for accessing the **xml-to-dml** utility from within XML-processing graph components.
Import for XML Split	Graphical interface for accessing the **xml-to-dml** utility from within the XML SPLIT component.

XML Processing

For function-based processing

The following component is available to help process XML data using the function-based approach:

Component	Description
XML REFORMAT	Parses or constructs XML data, operating in the same way as REFORMAT, but with additional predefined types and functions to support XML document processing.

XML Processing

For XML validation

The following component is available to help validate XML documents:

Component	Description
VALIDATE XML TRANSFORM	Separates records containing valid XML from records containing invalid XML. You must provide an XML Schema to validate against.

XML data processing approaches

Each data processing task comes with its own challenges and requires its own techniques. However, most XML processing tasks tend to yield to one of three general approaches.

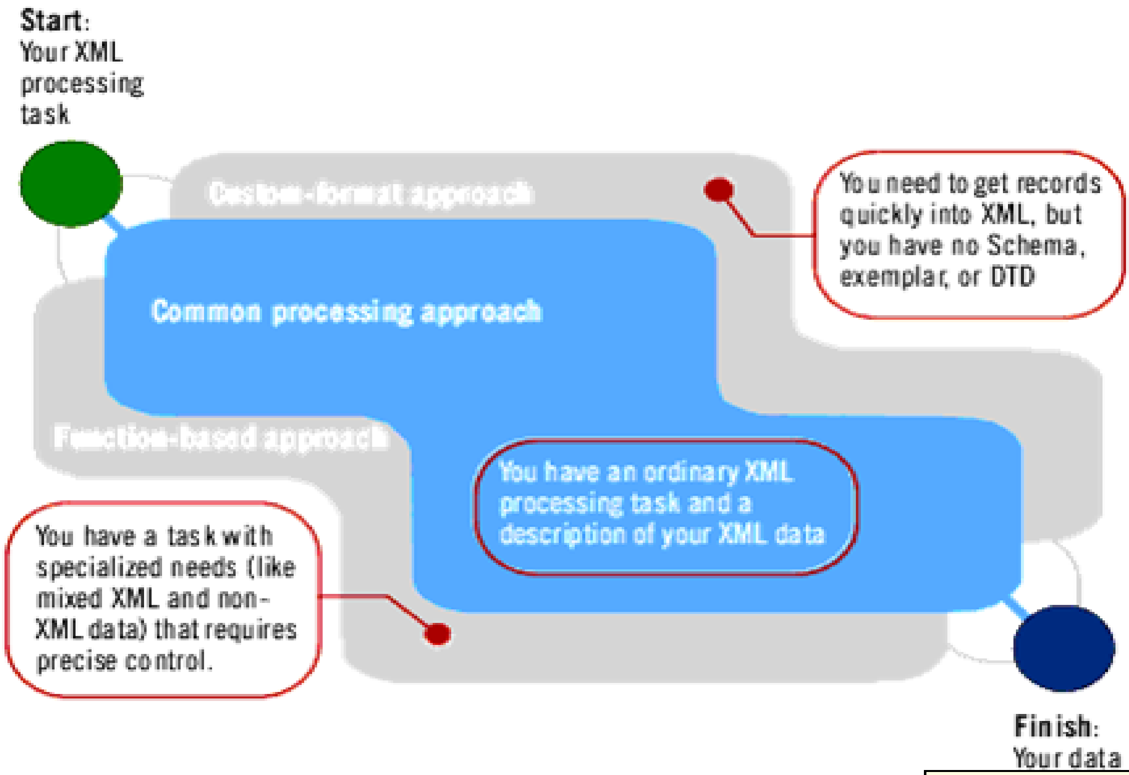

Start: Your XML processing task
Custom-format approach
You need to get records quickly into XML, but you have no Schema, exemplar, or DTD
Common processing approach
Function-based approach
You have an ordinary XML processing task and a description of your XML data
You have a task with specialized needs (like mixed XML and non-XML data) that requires precise control.
Finish: Your data

Diagram representing the basic approaches to processing XML data.

XML data processing approaches

The table below summarizes these XML processing approaches:

Approach	When to use	See
Common approach	98% of the time when you want to bring XML data into graphs, transform the data, and write it back out as XML.	"Common processing approach"
Function-based	In rare cases when the common approach does not work. For instance, this may happen when dealing with mixed XML and non-XML data.	"Function-based XML processing"
Custom format	When you lack a formal or informal XML description and need an expedient way to transform data into XML.	"Custom format XML processing"

Questions

Welcome Break

Test Your Understanding

- Instructions:
  - How to work on Conduct it
  - How to deal with PSETs
  - What is Resource Pool
  - What are Continuous flows
  - Different Performance tuning techniques in Abinitio
  - EME Tagging and Branching
  - Business Rules Environment

Summary

- Summarize important points here
  - PSETs
  - Continuous flows
  - Conduct it and Resource pool
  - DBC configuration and Custom components
  - Performance Tuning
  - ICFF
  - EME
  - BRE
  - XML Processing

Ab Initio for Intermediate Level

You have successfully completed – Part2