Ab-Initio Practice Tips

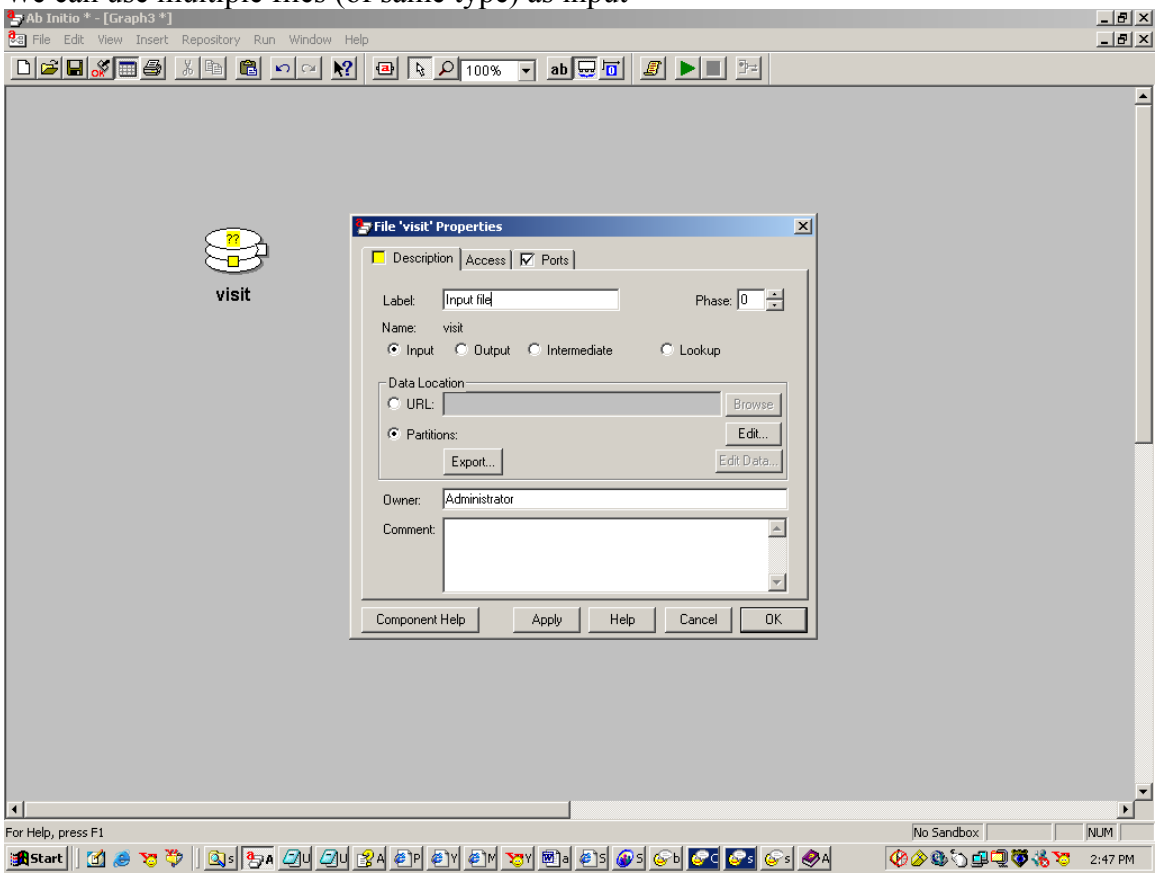
Dataset Components

Input file

Input File represents data records read as input to a graph from one or multiple serial files or from a multifile.

In the URL part of input file it is recommended to use variable ($ variable like $INPUT_FILEs)

We can use multiple files (of same type) as input



To do click on partition radio and the click the edit button. In the edit box mention the variable name which points the files

And the variable has to be defined in a fnx file like

export INPUT_FILES=`ls -1 $AI_TEMP/MML/CCE*`

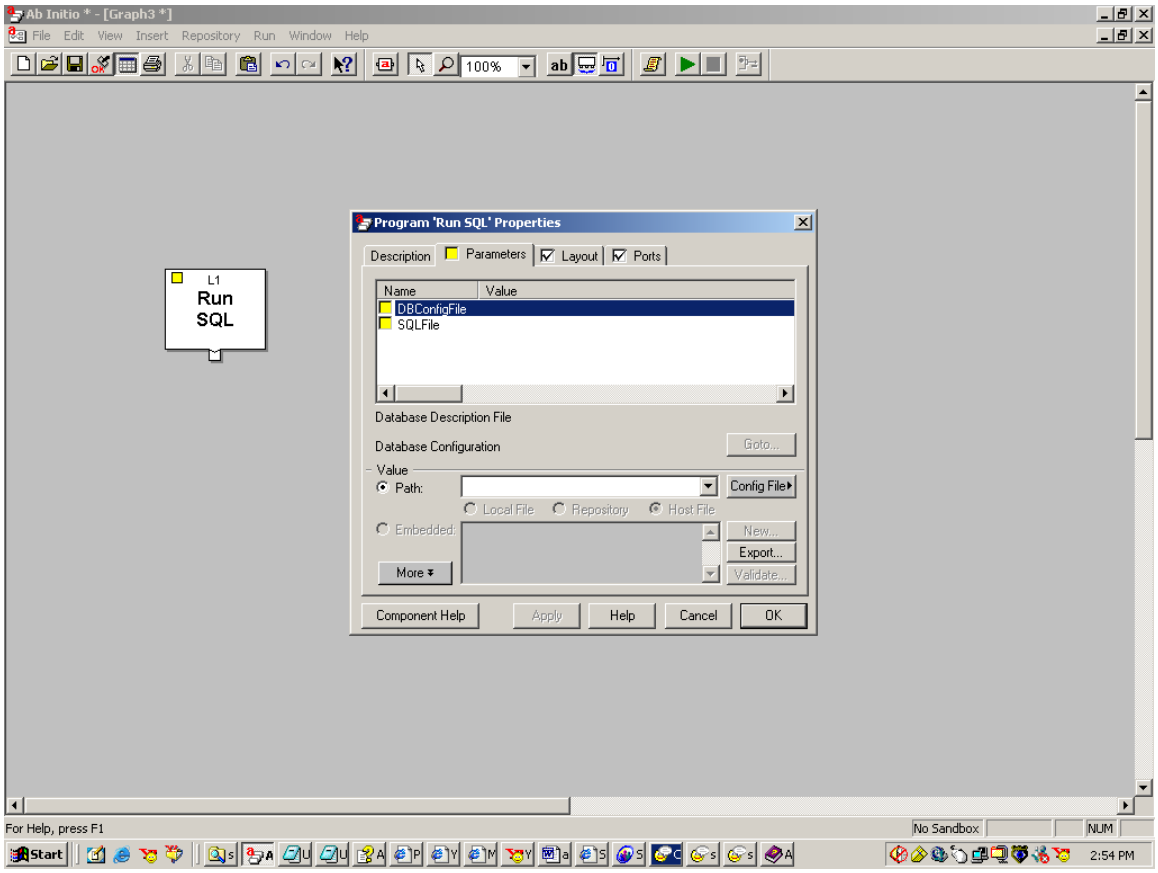
This INPUT_FILES points all the files under $AI_TEMP/MML directory which are stated with CCE.

Output file_: Output File stores data records from a graph to one or multiple serial files or to a multifile.

Database Components

Run Sql

Run sql is a database component. It executes SQL statements in an Oracle or SQL Server database.



In the run sql component two parameter files are required to be mentioned (as they are marked yellow)

1) DBConfigFile: Name of the database table configuration which is required to be used has to be mentioned here.

2) SQL file : Name of the file containing SQL statements. The DBMS execution utility executes the SQL statements contained in the file specified in the SqlFile parameter. If the DBMS execution utility successfully executes the statements, it exits with status 0.
For Oracle Databases

The DBMS execution utility is **sqlplus**. The first line of the SQL file should usually be:

WHENEVER SQLERROR exit failure;

This causes sqlplus to exit with a non-0 status on any SQL failure. The last line of the SQL file should always be:

exit;

The following is an example:

WHENEVER SQLERROR exit failure;
delete from table_foo where name='FRED';
commit;
exit;

For SQL Server Databases

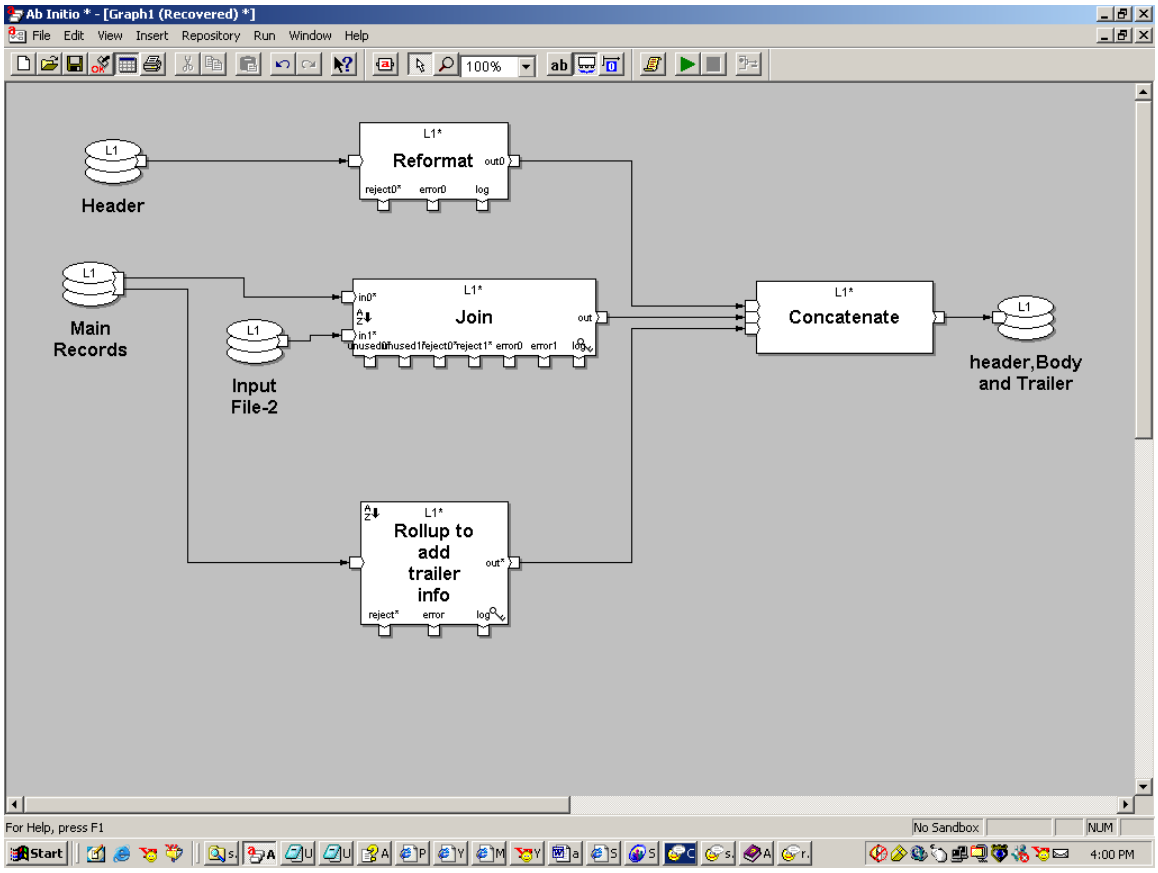
The DBMS execution utility **is osql**. The following is an example:

delete from table_foo where name='FRED';
GO

Departition Components

Concatenate: Concatenate appends multiple flow partitions of data records one after another.

Example : If the requirement is to generate and output file containing header, body and trailer part (all parts are from different flow)
[please find the picture as example]



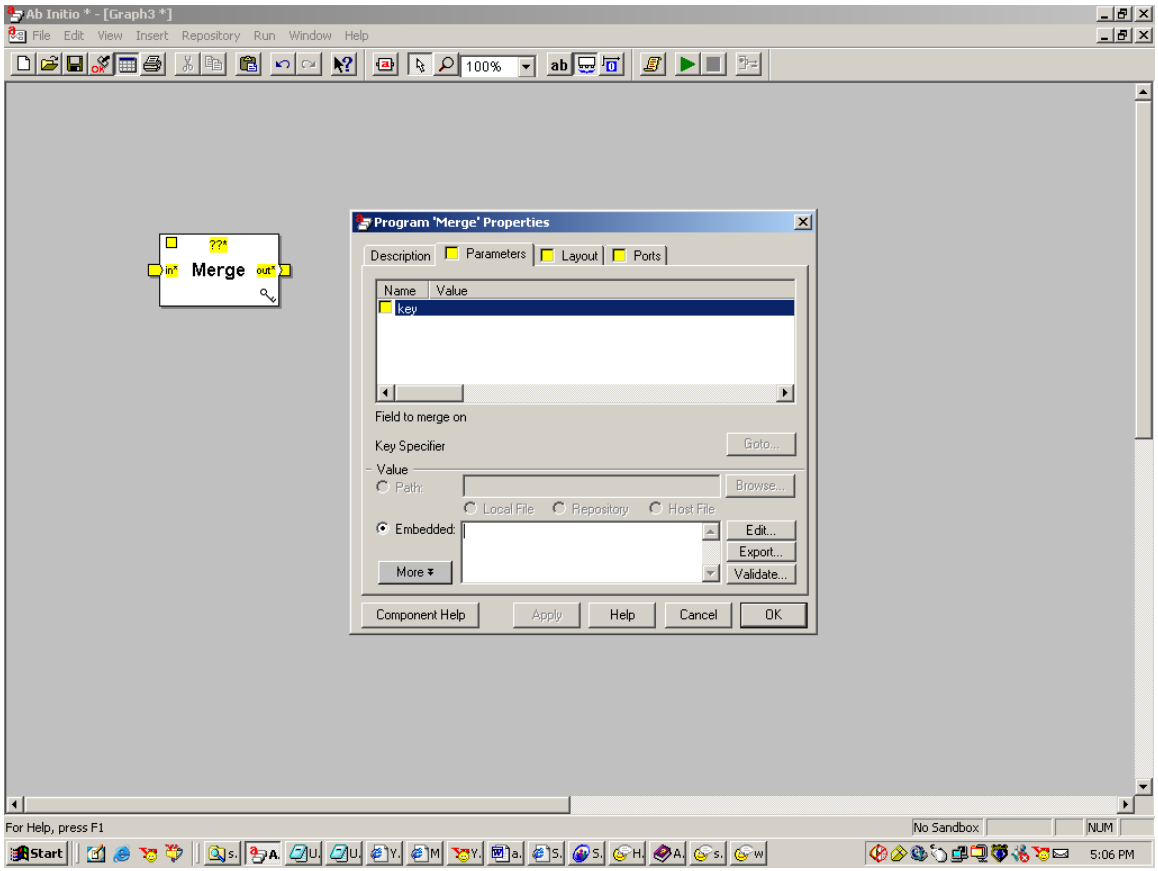
Gather:

Gather combines data records from multiple flow partitions (mfs) arbitrarily and make the flow serial and collect from different serial flow of same type (of same dml) to make it single flow.

Merge:

Merge combines data records from multiple flow partitions that have been sorted according to the same key specifier (for reference see yellow mark in parameter box), and maintains the sort order.

CAUTION: While using merge component use flow buffering.



[

There are some basic difference of concatenate, gather and merge which are mentioned as below

1. Concatenate: Append different flows of same types (same dml) in order in a single flow.
   In the example, concatenate will always take header record, then detail and then trailer.
2. Gather: Collect different flow arbitrarily.

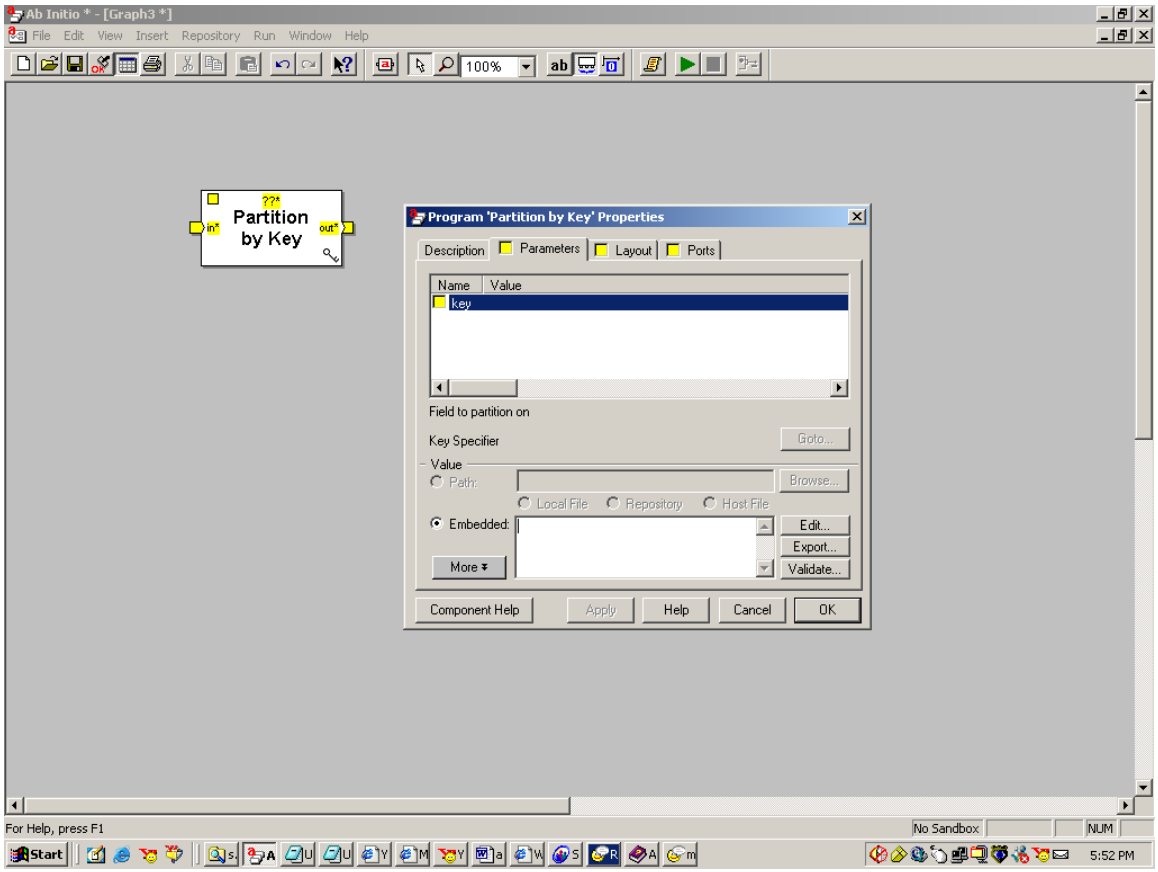
3. Merge: Collect different flows and maintain the sorted order.
   But the gather will do this arbitrarily

]

Partitioning Component

Partition by Key

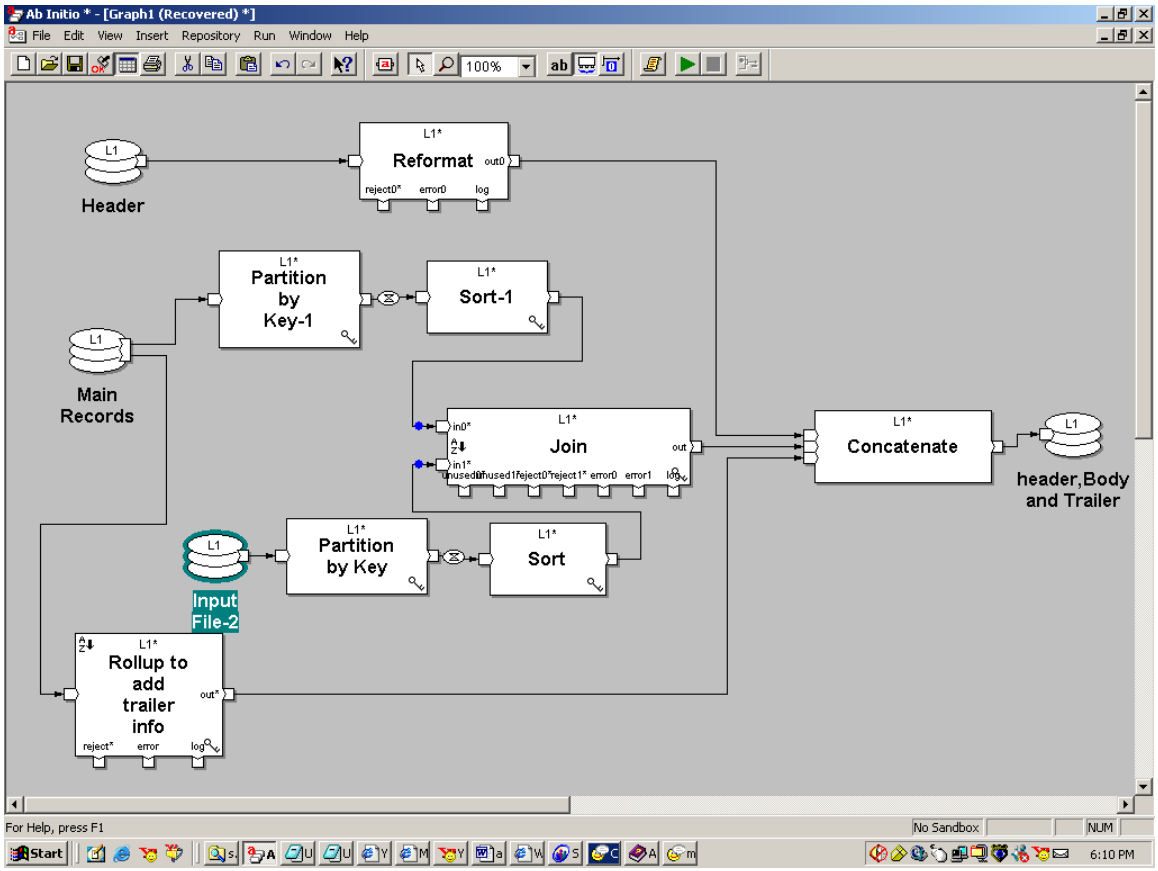
Partition by Key reads records from the in port and distributes data records to its output flow partitions according to key values.



In the parameter field key has to be mentioned

A ***partition by key*** component is generally followed by a ***sort*** component

See the example below



[In the above example in Join component sort parameter is used as *input must be sorted*]

Partition by Round Robin:

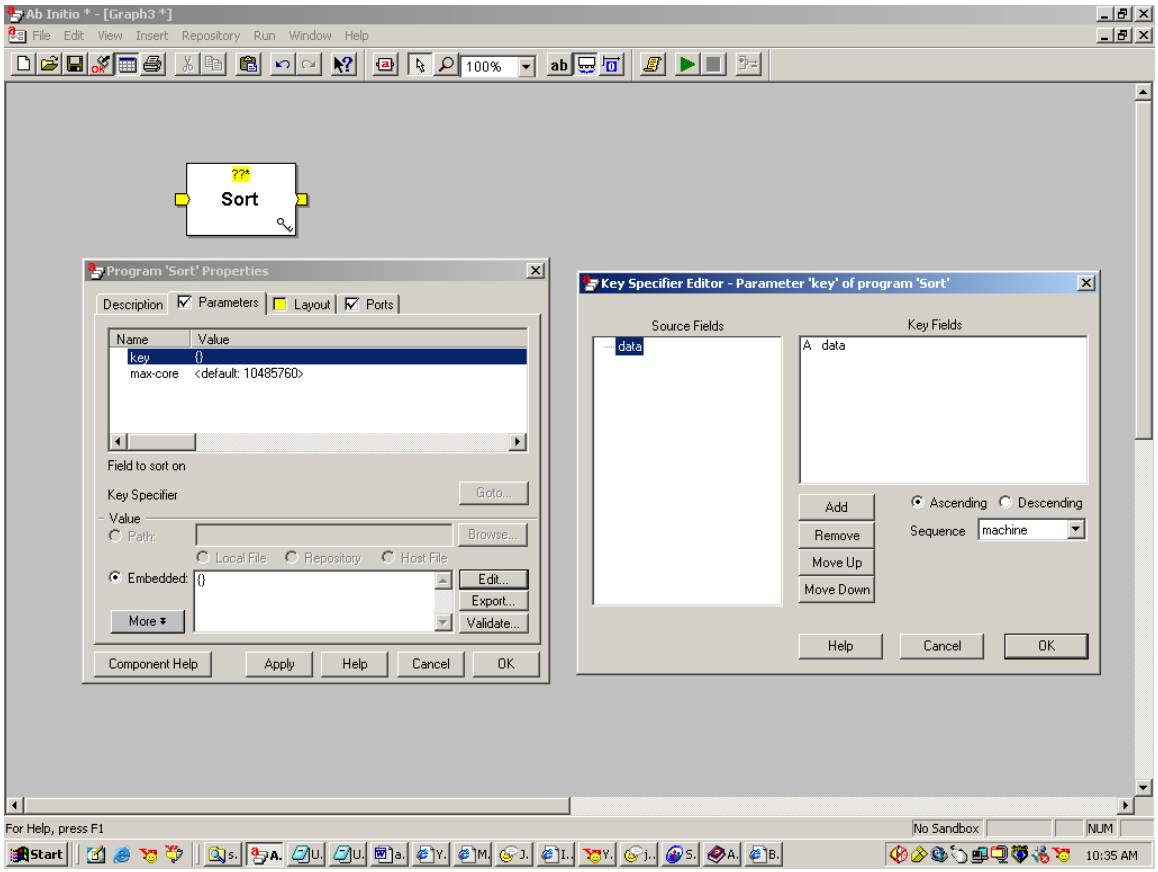
Partition by Round-robin distributes blocks of data records evenly to each output flow in round-robin fashion.

The difference between Partition by Key and Partition by Round Robin is the 1st one may not distribute data uniformly across the all partition in a multi file system but the latter does.

Sorting Component

Sort

Sort component sort the data in ascending or descending order according to the key specified.

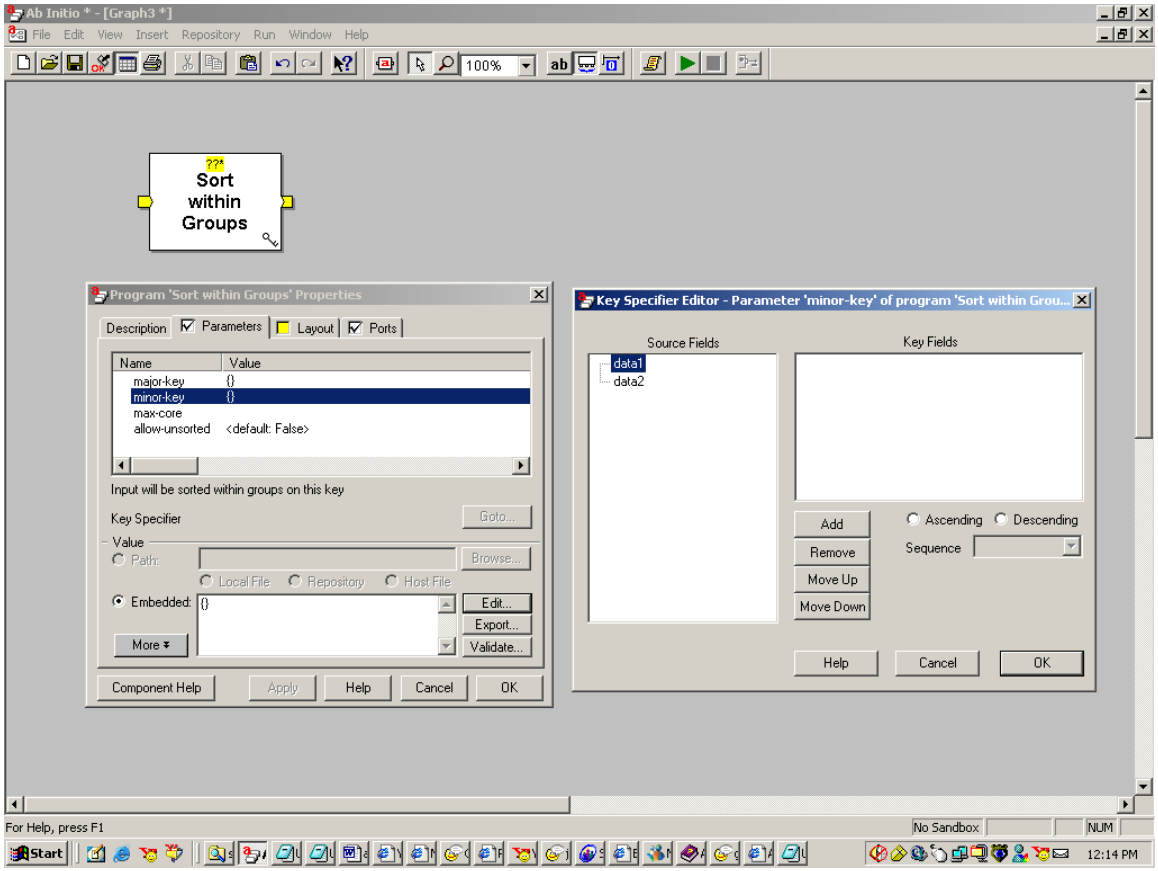


By default sorting is done in ascending order. To make the flow in descending order the descending radio button has to be clicked.
In the parameter max-core value is required to be specified. Though there is a default value, it recommended to use $ variable which is defined in the system [$MAX_CORE, $MAX_CORE_HALF etc].
It is strongly recommended not use the default or hard coded max-core value .

Sort within groups

Sort within Groups refines the sorting of data records already sorted according to one key specifier: it sorts the records within the groups formed by the first sort according to a second key specifier.
In parameter part there are two sort keys

1) major key: it is the main key on which records are already sorted.
2) minor key : If the records are already sorted according to major key, according to minor key records are resorted within major key group.



For example consider the record set

key1	*key2*	*data*
D	10	aaaaa
A	11	vvvvv
D	9	666ad
B	1	fadds
A	2	fsdgfd
A	1	nnn98
B	6	bbnna

After the records are sorted on major key which is key1 in this case. The records are expected by sort within group as

key1	*key2*	*data*
A	11	vvvv
A	2	fsdgfd
A	1	nnn98
B	1	fadds
B	6	bbnna

D	10	aaaaa
D	9	666ad

The output after sort within group will be if the minor key is key2

key1	*key2*	*data*
A	1	nnn98
A	2	fsdgfd
A	11	vvvv
B	1	fadds
B	6	bbnna
D	9	666ad
D	10	aaaaa

In this component max-core value should be set as per mentioned in case of sort component.

Partition by key and sort

Previously it was mentioned a partition by key component is generally followed by a sort component. If the partitioning key and sorting key is the same instead to using those two components *partition by key and sort* component should be used
In this component also key and max-core value has be mentioned as per same rule of sort component

Transform Component

Dedup Sorted

Dedup Sorted separates one specified data record in each group of data records from the rest of the records in the group i.e. removes duplicate records from the flow according to key specified.

The duplicate records from a flow can be removed by three ways by this component by mentioning the keep parameter.

1) first: This default the value. This implies the first record of the duplicates (i.e. same key value) will be kept
2) last: This implies the last record of the duplicates (i.e. same key value) will be kept
3) unique-only: In this case all the duplicate records will be removed

Example
Consider the following records

key1	*data*
A1	11
A1	12
A1	13
A2	14
B1	15
B2	16
B2	17
D1	18
D2	19
D2	20

The ou1put for case 1

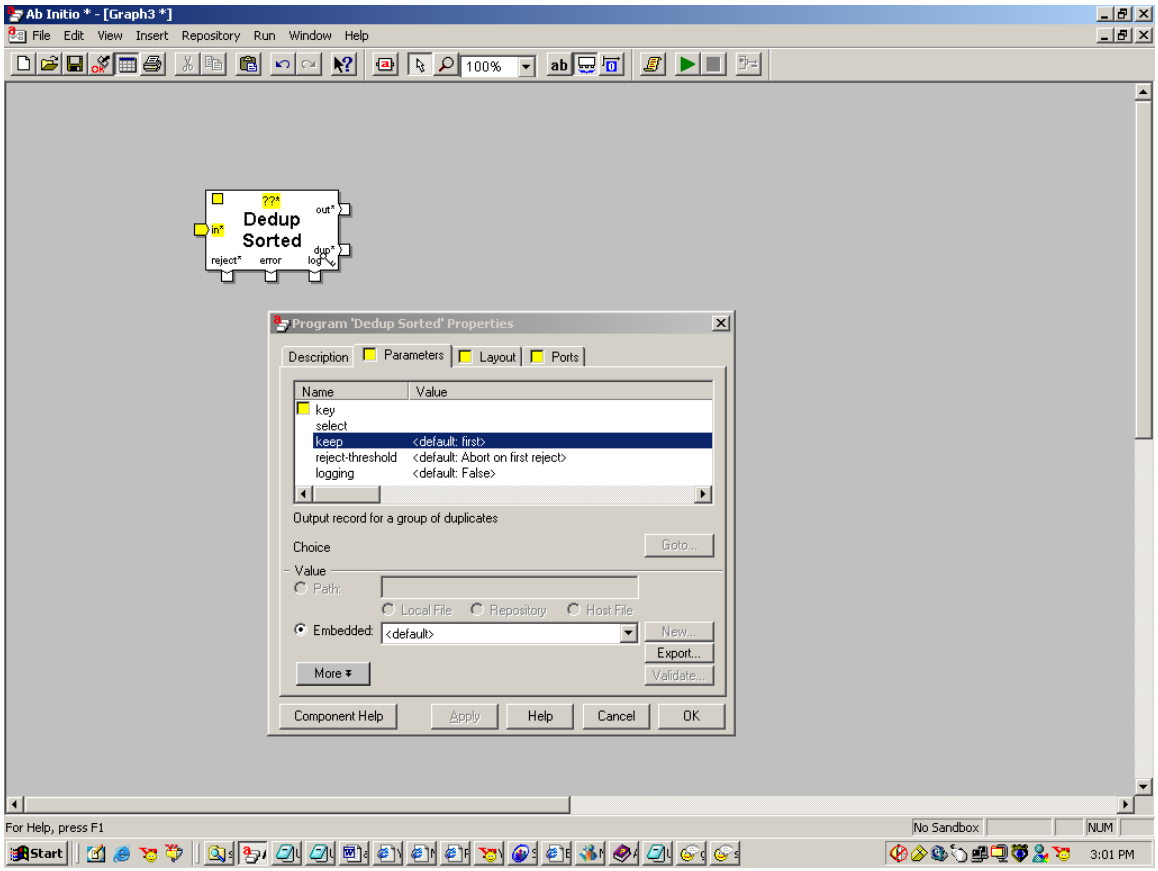
key1	*data*
A1	11
A2	14
B1	15
B2	16
D1	18
D2	19

Output for case 2

key1	*data*
A1	13
A2	14
B1	15
B2	17
D1	18
D2	20

Output for case 3

key1	*data*
A2	14
B1	15
D1	18

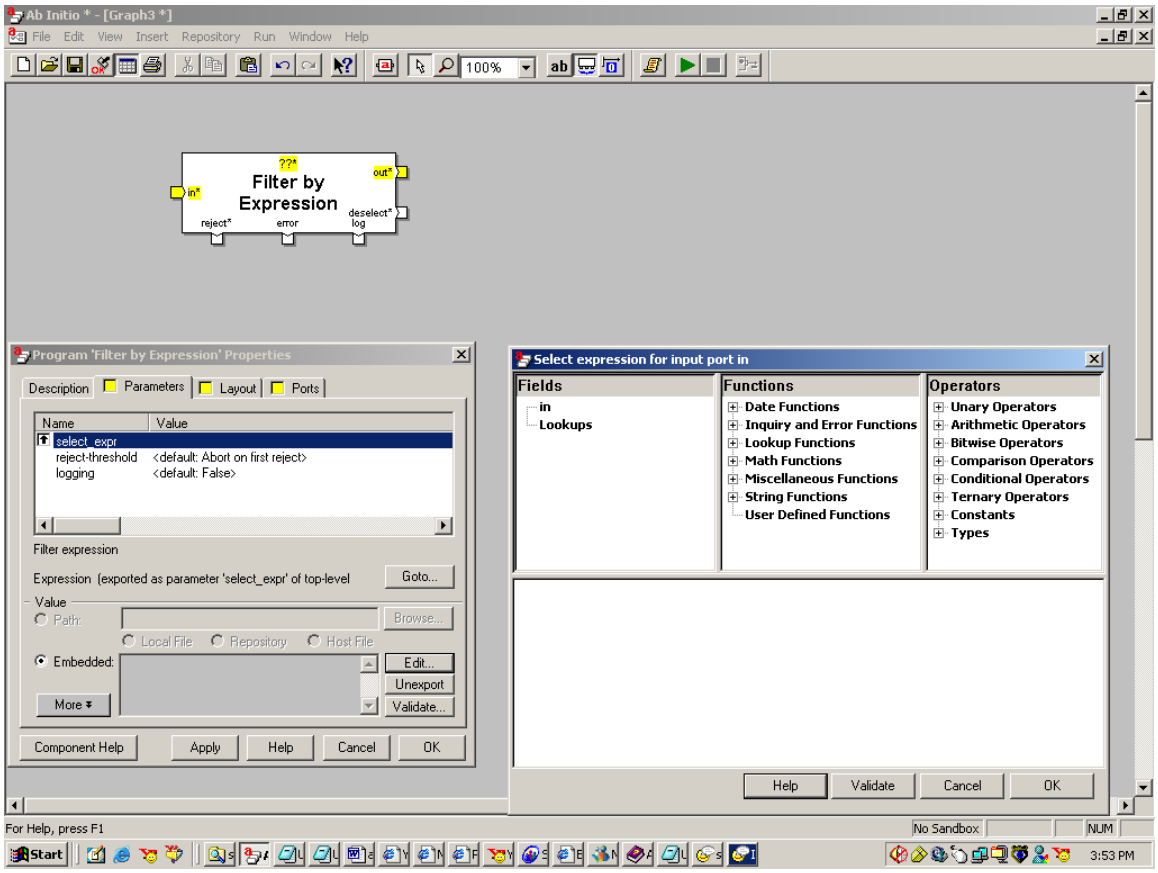


The above picture suggest where to fix different parameters for *dedup sorted* component

Transform Components

Filter by expression

Filter by Expression filters data records according to a specified DML expression.
Basically it can be compared with the where clause of sql select statement.
Different functions can be used in the select expression of the filter by expression component even lookup can also be used.



In this filter by expression there is **reject-threshold** parameter

The value of this parameter specifies the component's tolerance for reject events. Choose one of the following:

- **Abort on first reject** — Write Multiple Files stops the execution of the graph at the first reject event it generates.
- **Never abort** — the component does not stop the execution of the graph, no matter how many reject events it generates.
- **Use ramp/limit** — the component uses the settings in the ramp and limit parameters to determine how many reject events to allow before it stops the execution of the graph.

The default is **Abort on first reject**.

[ramp: Rate of toleration of **reject** events in the number of records processed.

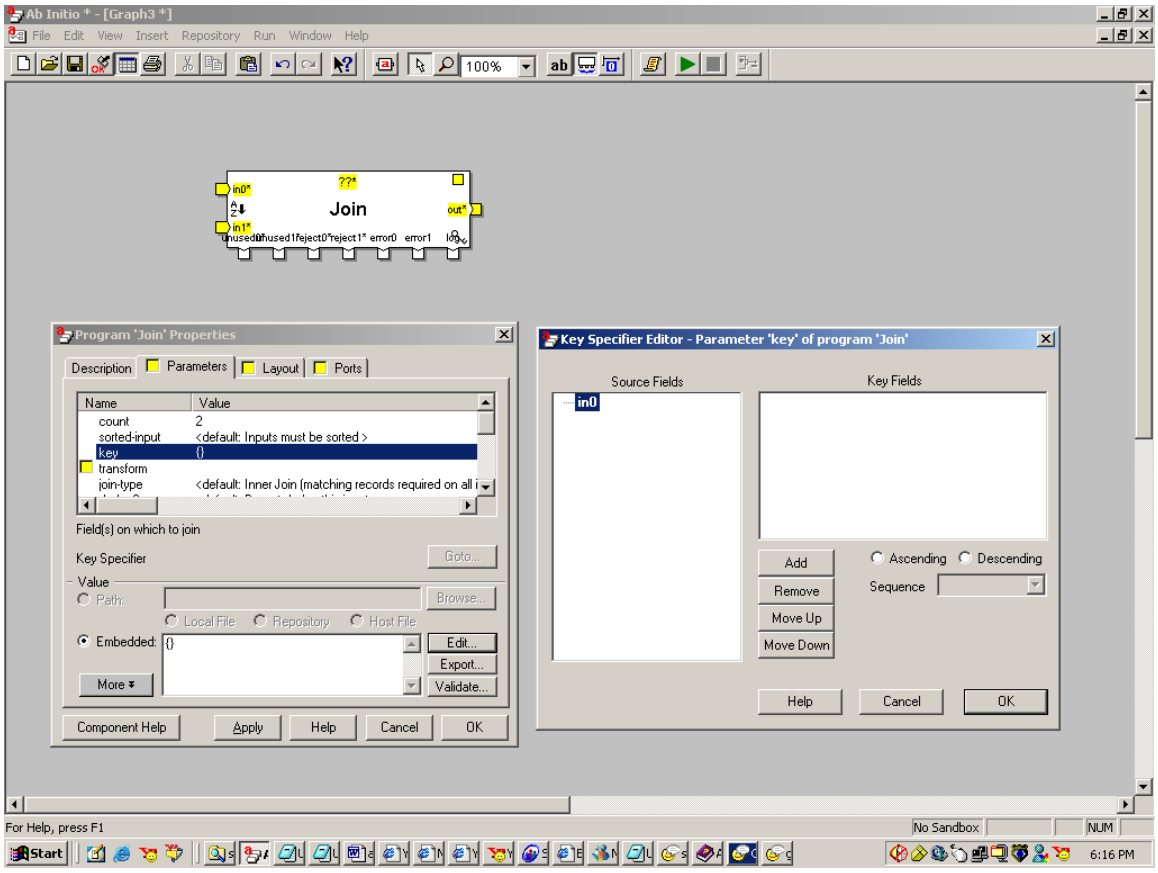
Default is **0.0**.

limit :A number representing **reject** events.

Default is **0**.]

Join

Join reads the records from multiple ports, operates on the records with matching keys using a multi input transform function and writes the result into output ports.



In join the key parameter has to be specified from input flow (either of the flow) **ascending or descending** order (please refer to picture above).

If all the input flows do not have any common field, **override-key** must be specified to map the key specified.
If any selection from input ports is required the select expression for the respective port instead of using extra ***filter_by_expression*** component. (But if the records which are not selected are also required to be processed it is recommended to use ***filter_by_expression*** component)
If the duplicate records are to be removed instead of using **dedup** component before join . **dedup** parameter for respective port can be used.
[above two cases can be example of how to minimize number of component. But]

reject-threshold also should be specified if default is not requied
[for reference see ***filter_by_expression*** component]

driving is Number of the port to which you want to connect the driving input. The driving input is the largest input. All other inputs are read into memory.

reject-threshold: for reference see ***filter_by_expression*** component

The **driving** parameter is only available when the **sorted-input** parameter is set to **In memory: Input need not be sorted**.

For example, suppose the largest input to be joined is on the **in1** port. Specify a port number of **1** as the value of the **driving** parameter. The Join component reads all other inputs to the join, for example, **in0**, and **in2**, into memory.

Default is **0**, which connects the driving input to port **in0**.

There are three types of join
Inner: sets the record-required parameters for all ports to false.

Standard inner join-
This example uses Join to join two input flows, connected to input ports in0 and in1. Both ports have record formats with two fields, and both are sorted on the first field, as follows:

In in0 flow

key1	*rec1*
1	A
1	B
2	B
3	C
5	E
6	F

In in1 flow

key1	*rec2*
1	A2
2	A3
4	D2
5	E2

If the output flow has got the record format key1, rec1 and rec2 and also joining key is key1 and the transform function is as follows

```
out :: joinit (in0, in1) =
begin
        out.key1 ::  in0. key1;
        out.rec1   ::  in0.rec1;
        out.rec2  ::  in1.rec2;
end;
```

If he dedup parameters to false, Join calls the transform function with the following arguments, and writes the results to the out port as

key1	*rec1*	*rec2*
1	A	A2
1	B	A2
2	B	A3
5	E	E2

Outer Join

Sets the record-required parameters for all ports to true.

This example shows the execution of a left outer join on two input flows, connected to ports in0 and in1. Both ports have record formats with two fields and both are sorted on the first field

Consider the record set as mentioned in case of inner join and the transform function as follows

```
out :: joinit (in0, in1) =
begin
        out.key1 :1 :  in0.key1;
        out.key1 :: in1.key1;
        out.rec1   : 1:  in0.rec1;
        out.rec1 :: “01” ;
        out.rec2  :1:  in1.rec2;
        out.rec2  : :  "00";
end;
```

Note the two assignments to the ***rec2*** field. The function uses the second assignment only if the first produces NULL.

[this is called priority assignment]

If you set the record-required parameter for port in0 to true and port in1 to false, and set dedup for both ports to false, Join calls the transform function with the following arguments and writes the result to the out port:

key1	rec1	rec2
1	A	A2
1	B	A2
2	B	A3
3	C	00 [as per priority rule mentioned above]
4	01	D2
5	E	E2
6	F	00

Explicit Join
Allows you to set the record-required parameter for each port individually.
If the input record set is as same as above and record-required parameter is set true for in0 port
For the transform function as mentioned below
out :: joinit (in0, in1) =
begin
out.key1 : : in0.key1;
out.rec1 : : in0.rec1;
out.rec2 :1: in1.rec2;
out.rec2 : : "00";
end;

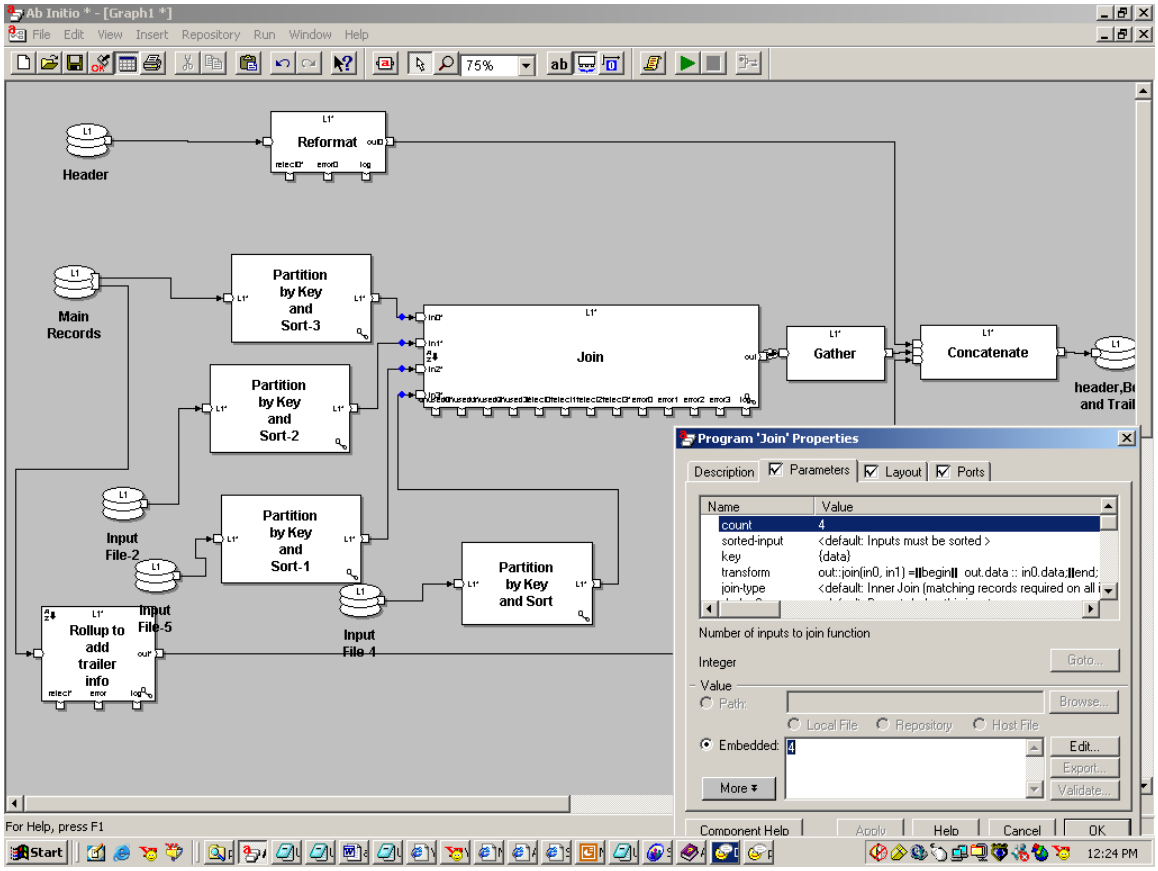
key1	rec1	rec2
1	A	A2
1	B	A2
2	B	A3
3	C	00
5	E	E2
6	F	00

Join operation is done after sorting the input flows. So depending upon that Join can be of two types

1. ***In Memory***: For this input need not to be sorted explicitly. Sorting is done in memory of the join component. So max-core value has to be specified for this. This minimizes using number of components (Sort component need to be used before Join component) .In this case max-core value has to be set (for reference see sort component).
2. ***Using sorted inputs*:** If the input is already sorted it is redundant to do In-Memory Join. For this sorted-input parameter is set to be required "***Input must be sorted***".

maintain-order parameter is required and of Boolean is set to true to ensure that records remain in the original order of the driving input..
If the maintain-order parameter to false, Join stores some of its intermediate results in temporary files on disk, which alters the order of the records in the driving input.
And if it is set to true, Join stops the execution of the graph.

Join component by default has got two input ports. For using more than two input set count parameter accordingly.
See the picture below for example



Reformat

Reformat changes the record format of data records by dropping fields, or by using DML expressions to add fields, combine fields, or transform the data in the records
By default reformat has got one output port but incrementing value of count parameter number. But for that two different transform functions has to be written for each output port.

If any selection from input ports is required the select parameter can be used instead of using '*Filter by expression*' component before reformat. (But if the records which are not selected are also required to be processed it is recommended to use ***filter_by_expression*** component)

New keys also can be introduced using reformat component
reject-threshold: for reference see ***filter_by_expression*** component

Consider the input dml
```
record
   string("~") key1;
  string("~") data;
end
```

and output dml
```
record
  string("~") key1;
  string("~") data;
  integer(8) expr_1;
end;
```

transform function for this case is

```
out::reformat(in) =
begin
  let integer(8) expr_1 =next_in_sequence()+1000;

  out.key1 :: in.key1;
  out.data :: in.data;
  out.expr_1 :: expr_1;
end;
```

A new variable is introduced expr_1 to get a new key value.
[This type of assignment can be done by Join, Rollup, Scan components also]

In case of transforming for a record if any field value is missing the reformat functionality will fail (the same is true for any other transform component like Join). In that default value should be introduced in the output dml. The output dml should be
```
record
  string("~") key1 = "";
  string("~") data = "" ;
  integer(8) expr_1 =0;
end;
```

Rollup

Rollup generates data records that summarize groups of data records on the basis of key specified.

Parts of Aggregate

- Input select (optional)
- Initialize
- Temporary variable declaration
- Rollup (Computation)
- Finalize
- Output select (optional)

Input_select : If it is defined , it filters the input records.

Initialize: rollup passes the first record in each group to the initialize transform function.

Temporary variable declaration:The initialize transform function creates a temporary record for the group, with record type temporary_type.

Rollup (Computation): Rollup calls the rollup transform function for each record in a group, using that record and the temporary record for the group as arguments. The rollup transform function returns a new temporary record.

Finalize:
If you leave sorted-input set to its default, Input must be sorted or grouped:

- Rollup calls the finalize transform function after it processes all the input records in a group.
- Rollup passes the temporary record for the group and the last input record in the group to the finalize transform function.
- The finalize transform function produces an output record for the group.
- Rollup repeats this procedure with each group.

If you set sorted-input to In memory: Input need not be sorted:

- After Rollup processes all the input records, it calls the finalize transform function with the temporary record for each group and an arbitrary input record from each group as arguments.
- The finalize transform function produces an output record for each group

Output select: If you have defined the output_select transform function, it filters the output records.

Consider the example mentioned
Records

Id	*amount*
1	100
2	200
1	150
1	175
3	250
2	350
4	500
3	75

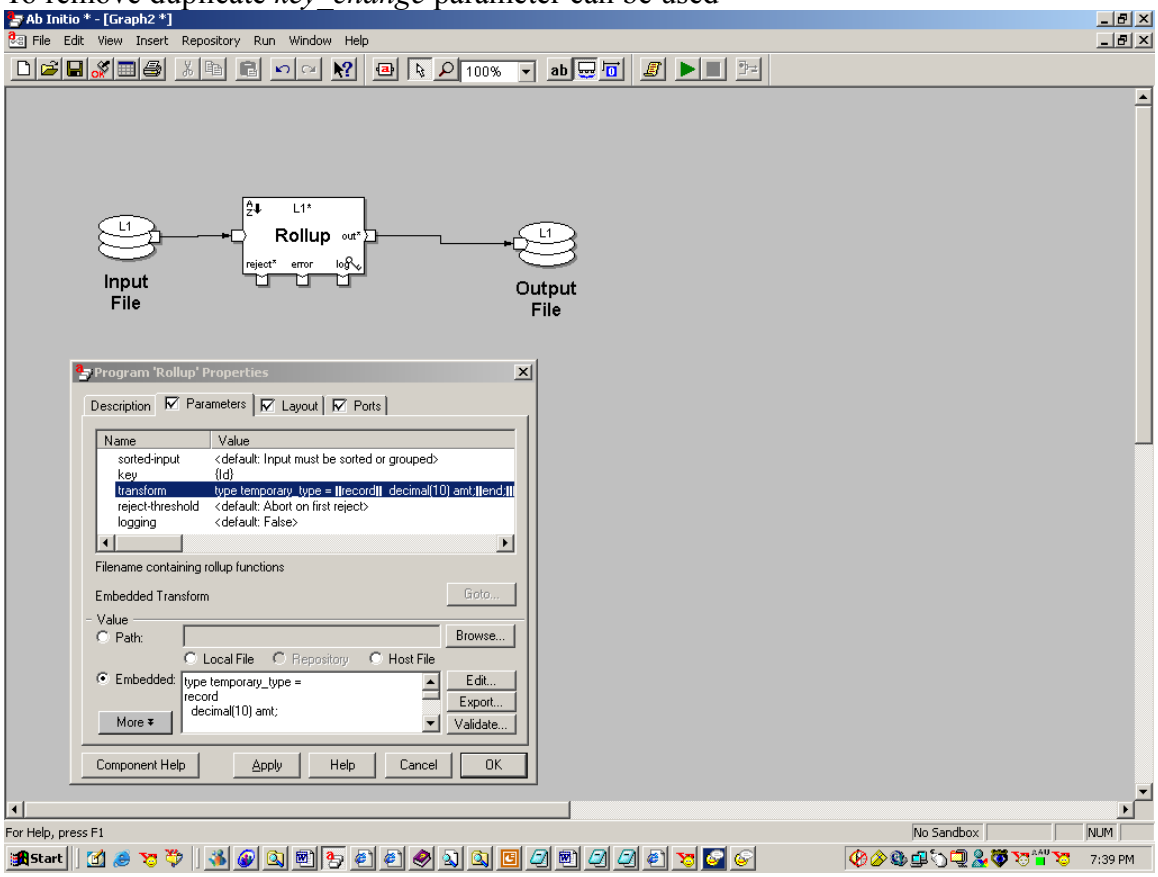
If input_select parameter is set amount>=150 and output_select parameter is set amount >200 (i.e. the output will be generated for only for those ids which has got amount >= 150 for individual record and total amount>200).
The transform function should be as follows

```
type temporary_type =
record
  decimal(10) amt;
end;

out::initialize(in) =
begin
  out.amt :: 0 ;
end;


out::rollup(tmp, in) =
begin
  out.amt :: tmp.amt+in.amount;
end;



/*Create output record*/
out::finalize(tmp, in) =
begin
  out.Id :: in.Id;  /**GENERATED*'in.Id'* */
  out.amount :: in.amount;  /**GENERATED*'in.amount'* */
end;


out::input_select(in) =
begin
  out :: in.amount>=150;
end;


out::output_select(out) =
begin
  out :: out.amount>200;
end;
```

The output will be as follows

Id	*amount*
1	325
2	550
3	750
4	500

If sorted-input parameter is specified as in-memory max-core value must be mentioned as mentioned in the previous cases.
If a new key is to be introduced and duplicate records are to be removed Rollup can also be used (this case can be considered as combined operation of dedup and reformat).
To remove duplicate *key_change* parameter can be used



Scan

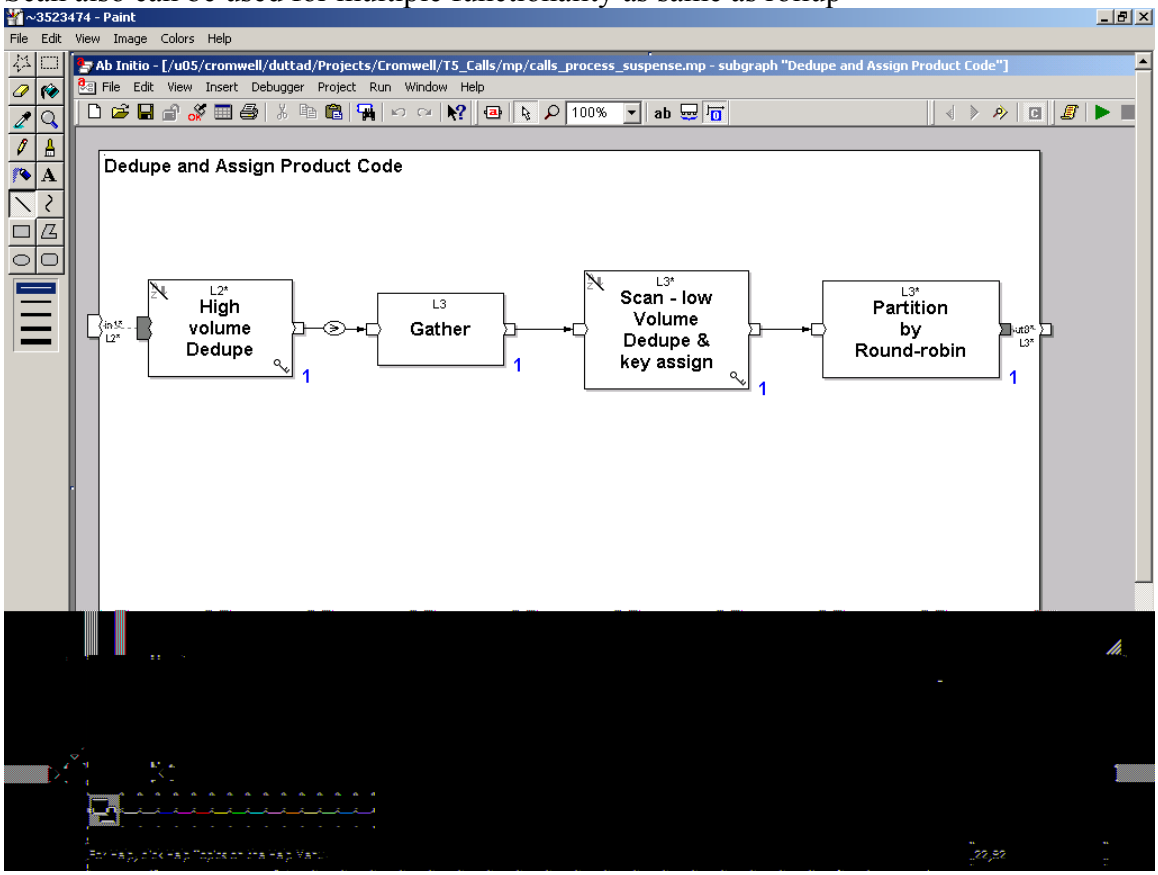
Scan generates a series of cumulative summary records for groups of data records.
Consider above case input records scan transform functions generates record in output as (if input_select and output_select parameters are not specified)

Id	*amount*
1	100
1	250

1 425
2 200
2 350
3 250
3 325
4 500

For this the transform function will be same like of Rollup except input_select and output_select parameter won't be in use.

Scan also can be used for multiple functionality as same as rollup



In the above picture Scan-low Volume Dedup & key assign is basically a Scan component. The transform function for this is

```
type temporary_type =
record
  integer(4) done_one;
end /* Temporary variable*/;

temp::initialize(in) =
begin
  temp.done_one :: 0;
end;
```

```
temp::scan(temp, in) =
begin
  if (temp.done_one)
   force_error("Already seen key value.");

  temp.done_one :: 1;
end;

let integer(8) g_max_prod_code =0;
let string(1) g_first_time ="Y";

out::finalize(temp, in) =
begin
  if (g_first_time=="Y")
  begin
     g_max_prod_code=lookup("max_product_code_lookup", "MAX").max_prod_code;
     g_first_time="N";
  end

  out.* :: in.*;
  out.prod_type_code :
```
[this is real time example]

The main difference between Scan and Rollup is Scan generates intermediate (cumulative) result and Rollup summarizes.

Denormalize sorted

Denormalize Sorted consolidates groups according to the key specified of related data records into a single output record with a vector field for each group, and optionally computes summary fields in the output record for each group.
Consider the following example

id	*txn_amount*
A1	150
A2	200
A1	250
A1	100
A3	250
A2	300
B1	100
B1	150

The output is required to keep a record for individual id mentioning all the txn_amount ,like

id	*no_of_txn*	*txn_amount[1]*	*txn_amount[2]*	*txn_amount[3]*
A1	3	150	250	100
A2	2	200	300	0
A3	1	250	0	0
B1	2	100	150	0

The transform function for this is should be

```
type denormalization_type = decimal(4)[3];

out :: initial_denormalization() =
begin
        out :: 0;
end;

type temporary_type =
        record
           decimal(4) count;
        end;

out :: initialize(in) =
begin
        out.count       :: 0;
end;

out :: rollup(temp, in) =
begin
        out.count       :: temp.count + 1;
end;

out :: denormalize(temp, denormalization, in, count) =

begin
        out.index  :: count;
        out.elt    :: in.amount;
        out.update :: count < 3;
end;

out :: finalize(temp, denormalization, in) =
begin
        out.id  :: in.id;
        out.count  :: temp.count;
        out.amount      :: denormalization;
end;
```

Normalize

Normalize generates multiple data records from each input data record; you can specify the number of output records, or the number of output records can depend on a field or fields in each input data record. Normalize can separate a data record with a vector field into several individual records, each containing one element of the vector.

Consider the previous output which is taken as input for Normalize component and produces output as same as input of previous example. The transform function will be as follows

```
temp :: initialize(in) =
begin
        temp.count :: 0;
end;
out :: length(in) =
begin
        out :: in.count;
end;

out :: normalize(in, index) =
begin
        out.id :: in.id;

        out.amount     :: in.transactions[index].txn_amount;
end;
out :: normalize(in, index) =
begin
        out.customer_id :: in.customer_id;
         out.txn_amount :: in.transactions[index]. txn_amount;
end;
out :: finalize(temp, in) =
begin
        out.key     :: in.key;
        out.count   :: temp.count;
end;
```

Continuous Flow

A continuous job is a job that produces usable output before it ends. A continuous flow graph, unlike a regular Ab Initio graph, is intended to run for an indefinite period of time,

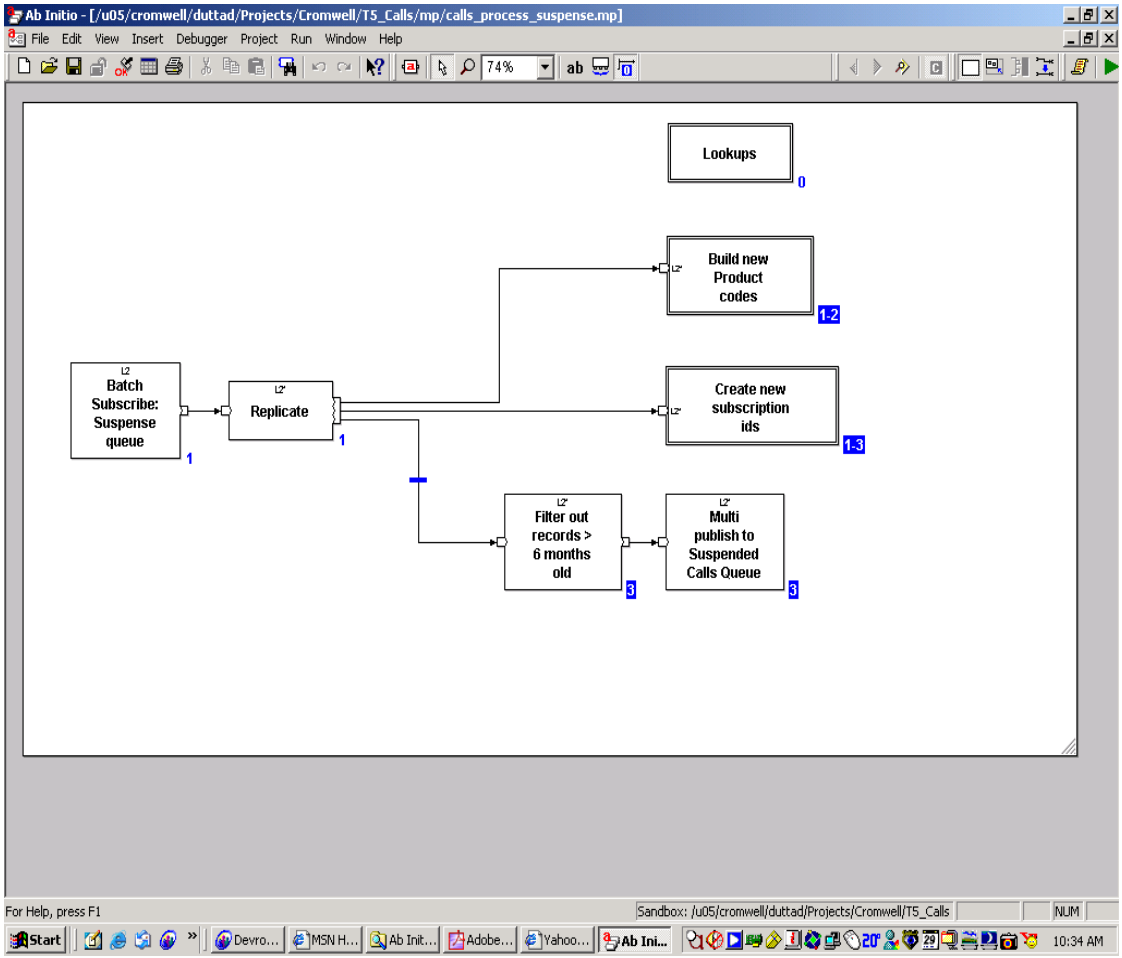
continually taking in new input and producing new, usable output while the graph keeps running. A continuous flow graph might or might not go on forever.

The advantages of continuous flow graphs include better performance and latency. There is no overhead for starting the job each time a new batch of data arrives. Results of the job are available sooner than for a non-continuous graph.

A continuous flow graph includes

- One or more subscribers. A subscriber is the only allowed data source.
- A publisher at the end of every data flow.
- Any continuous or continuously enabled component can be in the middle, between a subscriber and a publisher.

Example of Continuous flow



Restrictions for continuous flow

- All components in the graph must be continuous components or they must be continuously enabled.

- All components with no output flows must be publishers. There must be at least one publisher in the graph for the graph to determine when a checkpoint can be committed.
- All subscribers must issue checkpoints and compute points in the same sequence.
- The graph must execute in a single phase. More than one phase is not allowed.
- All data in a continuous flow graph must come from a subscriber component. The source of data cannot be an Input File or Input Table component.

Ab_Initio Queue

Ab Initio queues are an adaptation of the first-in/first-out queue concept:

- They provide record-based persistence.
- Publishers write data to the queue.
- Subscribers to the queue read the data in the order it was written.

In many ways, Ab Initio queues are analogous to multifiles in ordinary Ab Initio graphs. They provide a method for storing records in an ordered sequence of files. However, they also hold additional information necessary to allow both the removal of data from disk when it is no longer needed, and the recovery of graphs that stop running for any reason.

Queues are the most reliable method for storing continuous flow data. We recommend that you get your data into this format as early as possible in the data processing stream

You use the **m_queue** command to create an Ab Initio queue. When you execute this command, you specify the name of the directory or multidirectory that you want to contain the new queue. This is the value of the *queue_path* argument to the **m_queue** command. The Co>Operating System creates the directory you specify and sets up a queue inside it. A directory or multidirectory can contain only one queue. Consequently, although the directory is not the queue, the name of the directory that contains the queue identifies the queue.

Each Ab Initio queue directory contains a number of files, which contain the queue data and the queue infrastructure. If you look at the queue in the file system, you can see these files. However, you never directly operate on them.

m_queue command operartion

m_queue create -f <name of directory> < subscriber1> < subscriber2> < subscriber3>....

Example

m_queue create -f
/u05/cromwell/puranika/Projects/Cromwell/Registartion/data/input/queue sub1 sub2

Stopping Continuous Flow Graphs

Use one of the following two commands to stop a continuous flow graph:

- m_shutdown [-f | -status] job_name Waits until the graph commits the next checkpoint in order to end the job cleanly, then stops the execution of the graph and deletes all checkpoint temporary files.
- m_kill job_name The execution of the graph stops immediately

Advanced Dataset Components

Lookup

Lookup File represents one or multiple serial files or a multifile of data records small enough to be held in main memory, letting a transform function retrieve records much more quickly than it could retrieve them if they were stored on disk.

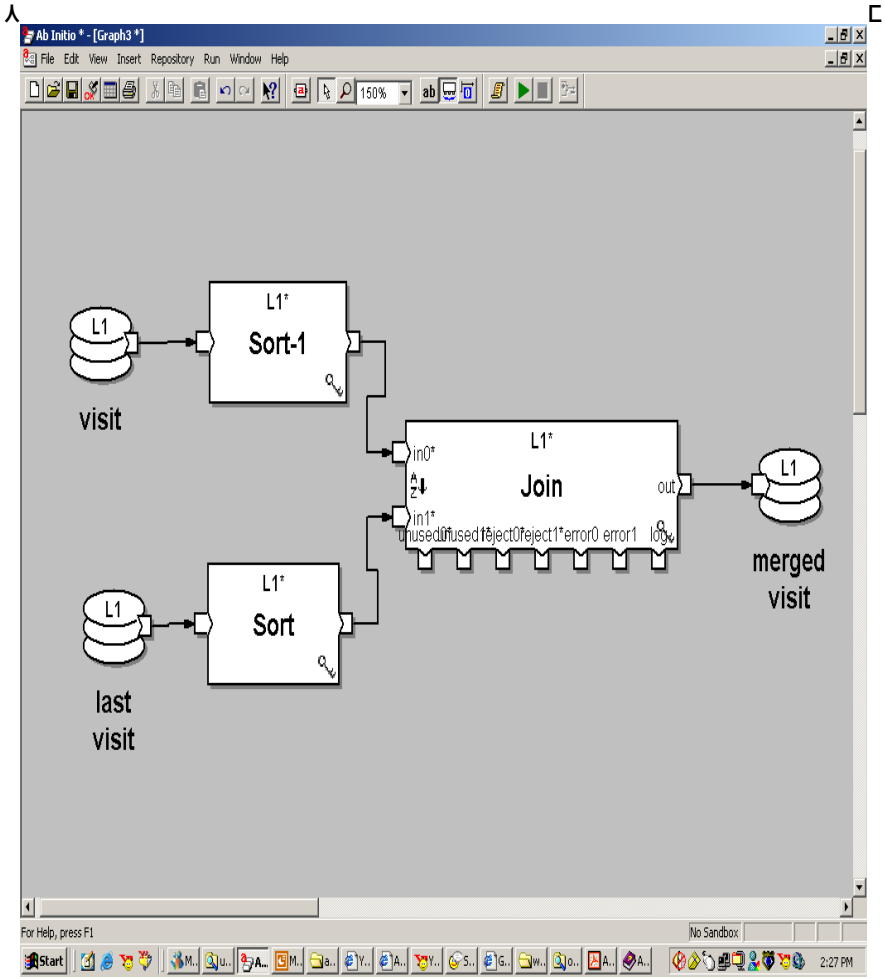
Lookup File associates key values with corresponding data values to index records and retrieve them.

It is better to use catalogs to share lookup files in multiple graphs. Use lookup files to quickly retrieve values from a small dataset; this is often the best way to merge a large dataset with a small one. Although a Join component can perform the same task, lookup files run faster because they are stored in memory.

CAUTION: Beware of memory overload.

This example shows how to use a Lookup File to improve the performance of a graph.

The first part of the example (see the following graph) joins the Visits dataset with the Last Visits dataset to produce the Merged Visits dataset



In the above graph, Join has a transform function which assigns values from the Last Visits dataset to the lastdate field of the Merged Visits dataset, as follows:

out :: merge(visits, lastvisits) =

begin

out.id : : visits.id;

out.lastdate :1: lastvisits.dt;

out.lastdate : : 0;

end;

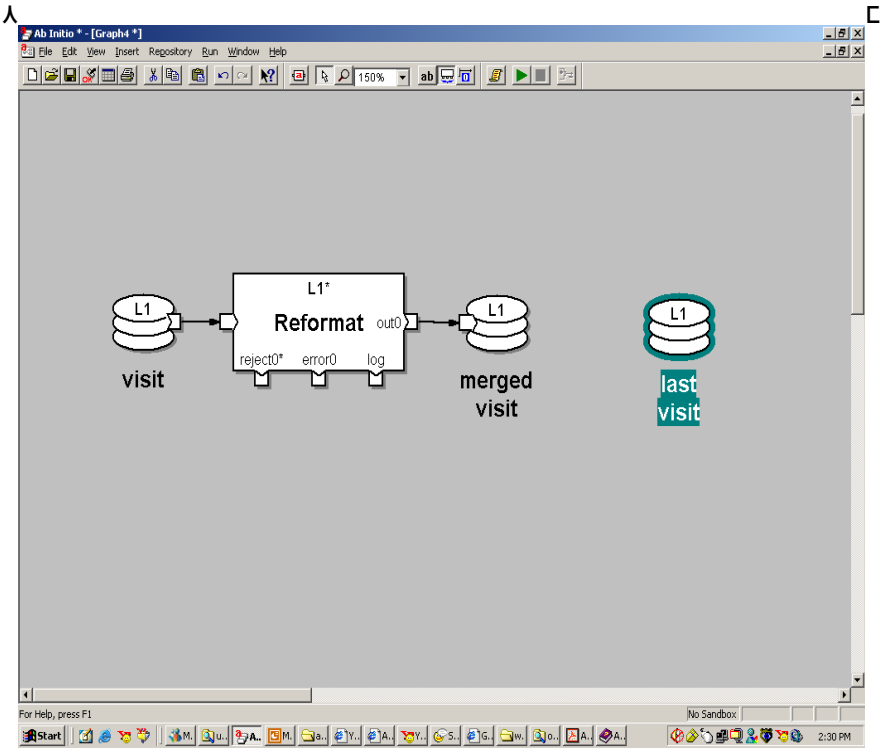
The performance of the above example can be improved using the following option.

If the Last Visits dataset is small enough to fit into memory, use it as a Lookup File and call the lookup function in the Reformat transform function.

The Reformat component's transform function also sets the lastdate field as follows.

The Reformat component's transform function also sets the lastdate field as follows:

```
out :: reformat(visits) =

begin

        out.id       :: visits.id;

        out.lastdate :: if (lookup("Last Visits", visits.id))

                                lookup("Last Visits", visits.id).dt

                        else

                                0;

end;
```



Read multiple file

Read Multiple Files reads input records, and derives a filename from each one. Records are read from each of these files and passed to the out port, passing through an optional transform function. Filtering criteria allow a set number of records to be skipped at the beginning of each file and a limit on the number that are read from each file.

The Read Multiple Files component:

Reads records from the in port.

Passes each input record to the get_filename() function. The get_filename() function returns a string containing a filename.

The filename must be local to the machine on which the component is running.

Note that in multifile layouts, different instances of the component can be running on different computers, so input filenames which refer to directories that exist only on

particular computers will have to be partitioned carefully. Relative paths also are impractical, unless you are using a custom layout with "exact paths" specified.

For each filename retrieved:

The file is opened. If an input_type data type is defined in the transform package, this type is used to read the file. Otherwise, the record type of the out port is used.

The records are read from the file. For each record, processing proceeds as follows:

i. If the filter parameter was set to range, and skip-records is non-zero, and the current record count is lower than or equal to it, the record is discarded. Once the count is equal to skip-records, subsequent records will be processed.

ii. If the filter parameter was set to range, and read-records records have been processed, the current input file is closed.

iii. If a reformat() function is present in the package, the filename and the input record are passed to that function. The resulting record is written to the out port. If no reformat() function is present, the input record is written to the out port. In the latter case, data transformations will still take place if appropriate to the input and output DML, as in a Reformat component with no transform.

Write multiple file

Write Multiple Files reads input records, and derives a filename from each one. Each input record is written to the designated file, passing through an optional transform function. The number of simultaneously open files can be limited by number or by memory usage.

The Write Multiple Files component:

Reads records from the in port.

Passes each input record to the get_filename() function. The get_filename() function returns a string containing a filename.

The filename must be local to the computer on which the component is running.

Note that in multifile layouts, different instances of the component can be running on different computers, so output filenames which refer to directories that exist only on particular computers will have to be partitioned carefully. Relative paths also are impractical, unless you are using a custom layout with "exact paths" specified.

For each record:

If the designated output file is not open, it is opened. If an output_type data type is defined in the transform package, this type is used to write the file. Otherwise, the record type of the in port is used.

If the number of open files is limited, and the opening of a new output file would exceed the limit of open files, the output file least recently opened is closed before the new file is opened.

If a reformat() function is present in the package, the filename and the input record are passed to that function. The resulting record is written to the output file. If no reformat() function is present, the input record is written to the output file. In the latter case, data transformations will still take place if appropriate to the input and output DML, as in a Reformat component with no transform.

If the get_filename function returns NULL, Write Multiple Files writes the current input record to the reject port. The component stops the execution of the graph when the number of reject events exceeds the result of the following formula:

limit +

(ramp * number_of_records_processed_so_far)

See About the ramp and limit Parameters.

Also, Write Multiple Files writes an error message to the error port.

Output record format

If you wish, in the transform (whether this is in the form of embedded text or a transform file) you can create an optional DML type named output_type to define the record format of the output files. If output_type is omitted, records are written using the record format defined on the in port.

Macro Components

Ab-initio provides three facilities developing graphs beyond simply inserting pre-built components into workplace and connecting them each other. These are

Custom components

Subgraphs

Macros

When to use Custom component

If the solution to the task is a single executable, you need to use custom component.

Typically you have a program or script you have created in past to perform some type of data transformation, and you now want to use it in Ab-Initio graph. Alternatively, you can write a new program or script with a specific purpose in mind.

A custom component lets you integrate your program or script into an Ab-Initio graph. You can use a custom component in a same way that you would use an Ab-initio pre-built component

When to use a Sub-graph

If you can construct the solution to the task from Ab-Initio pre-built components and you can keep the number and arrangements static from one run of the graph to another you can use sub-graph. Of the three above mentioned facilities sub-graph is the easiest to use.

When you use a sub-graph you can define components parameters at runtime. You can change the value of parameter from one graph to another but the number of component remains static.

When to use Macro

You need a macro if the solution to the task requires that you change from one graph to another, any of the following:

the number of components

which components you use

the order in which you connect the components

In a macro, the components, the flows that connect them and their parameters become runtime parameters of the graph. You can change some or all of them from one run of the graph to another.

For example

you need to divide data records by month in to separate files

at one run of the graph you need four output files, one for each last three months and one for all earlier records.

at a later run of the same graph you need six output files , one for each of the last five months and one for all earlier records.

In other words, the number of output files you need from one run of the graph to another varies.

Creating a macro

Creating a macro is more complicated than creating a sub-graph,although there are similarities.

A macro consists of two parts

Creating a macro is more complicated than creating a sub-graph, although there are similarities.

A macro consists of two parts

1) A program specification file also known as .mpc file.

2) A .ksh script that builds a graph fragment using the mp commands of the Shell

Development Environment.

The program specification file

Program specification files (.mpc) provide the Ab-Initio Co-Operating System with the information it needs to run your .ksh script. All the .mpc files must start with <mpcfile>. The <mpcfile> line is followed by a series of attribute:value lines that describe the attributes of your .ksh script.

The following are some details that distinguish a .mpc file for a macro:

In .mpc file you must use the mpname line. It must begin with @, which tells the Co-Operating system that this is a macro and then it must specify the complete path to the .ksh script that the macro uses.

[do not use the mpname line for a custom component.]

In almost all cases you need the port lines to define soc port, since this is the default Ab-Initio Interface

Place the completed program specification file on both the computer running GDE and the UNIX control node running the Co-Operating System:

In GDE save the file in :c\program files\ab initio\ab initio gde\components\my components

On the UNIX control node place the file in $AB_HOME/lib or set the $AB_LAYERED_COMPONENTS_PATH environment variable to the directory where the file resides

The .ksh script

The .ksh script provides the code the macro uses to accomplish the task.

This should be saved in a directory of the UNIX control node that is accessible to any Ab-Initio graph. This directory must be the one specified in the mp name line of the program specification file.

Following is the simplest way to find the basic mp commands you need for a script:

1. Insert the components you want to use in the macro into a graph in the GDE.

2. From the Edit menu choose Script>Generated Script.

   If the "Errors were detected during compilation ..." message appears, click OK then

   NO.

3. Find the lines in the script beginning with mp component_name.

Conditional Components

The GDE supports Conditional Components where a shell expression determines, at runtime, whether or not to include certain components.

To turn on this feature click on

File -> Preferences -> “Parameters” section of dialog

Check “Conditional Components”

The “Condition” tab appears on all components

[for reference see the picture below]

Conditional Components example

The Condition expression is a shell expression that, if it evaluates to “0” or “false”, will cause the component to not exist. Any other value means the component will exist.

Make sure the shell expression returns the string “0” or “false”, not numerical 0.

Components which are conditioned out can be replaced by a flow or removed completely. When removing completely, make sure you don’t leave any required ports unconnected.

To apply the same condition to more than one component, make them a subgraph and condition the subgraph.

Parallelism

Ab-initio can process data in a parallel runtime environment

There are three forms of parallelism

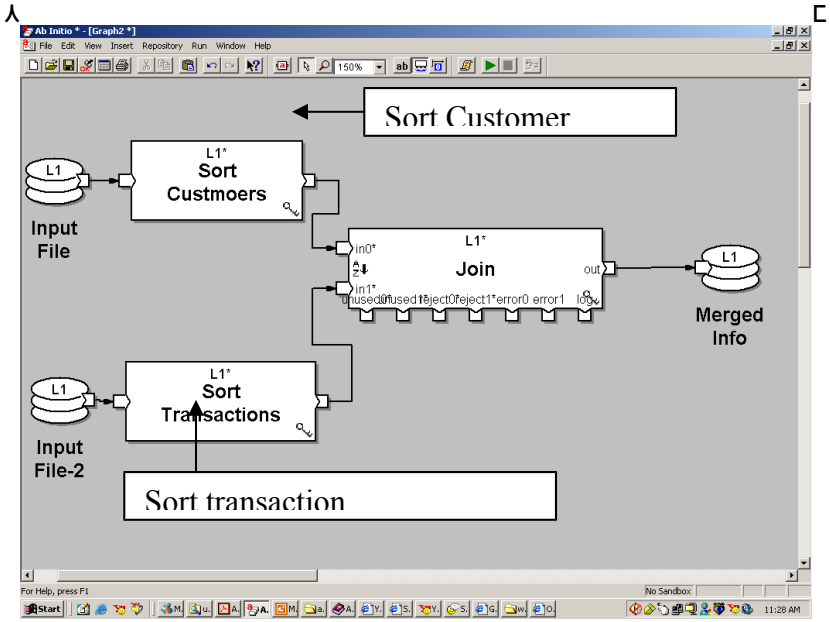
Component parallelism

Pipeline parallelism

Data parallelism

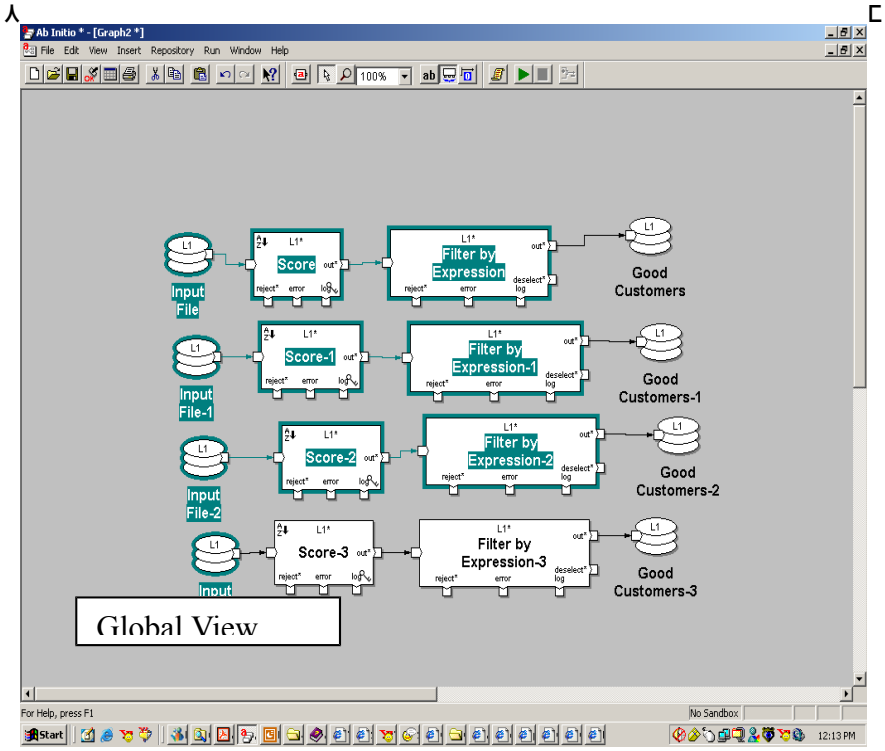
Component Parallelism

An application with multiple processes running on separate data uses component-level parallelism.



Data parallelism

An application that deals with data divided into segments and operates on each segment simultaneously uses data parallelism. Nearly all commercial data processing tasks can use data parallelism.

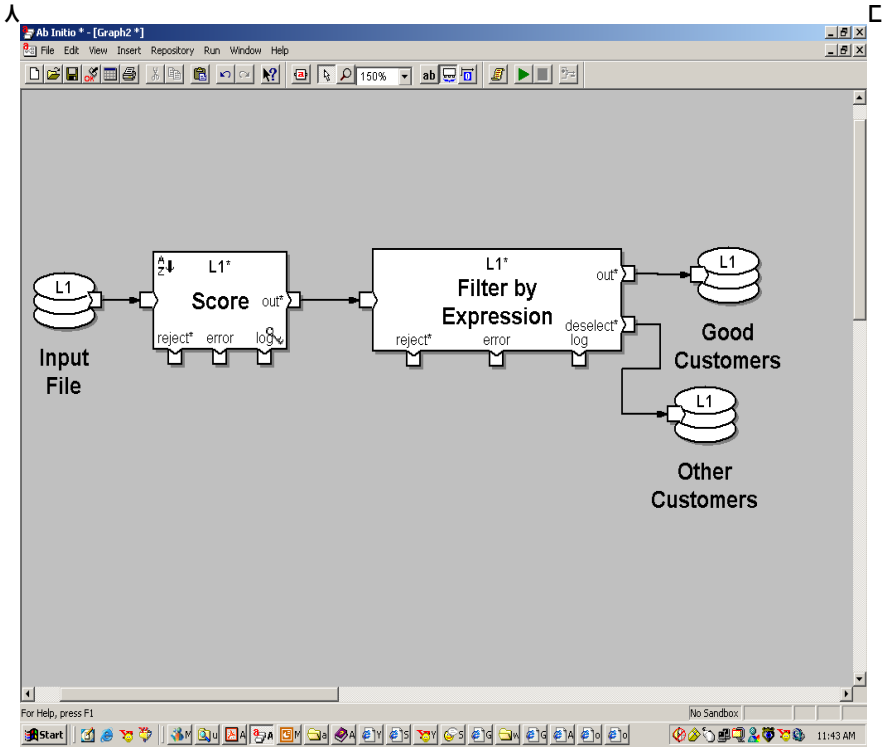


Partitioning is an example of data parallelism

Pipeline Parallelism

An application with multiple components running simultaneously on the same data uses pipeline parallelism.

Each component in the pipeline continuously reads from upstream components, processes data, and writes to downstream components. Since a downstream component can process records previously written by an upstream component, both components can operate in parallel.



Advanced Concepts

Phasing

A Phase is a stage of application that runs to completion before the start of the next stage. By dividing the graph into phases one can save resources and avoid deadlock (discussed later).

e.g.

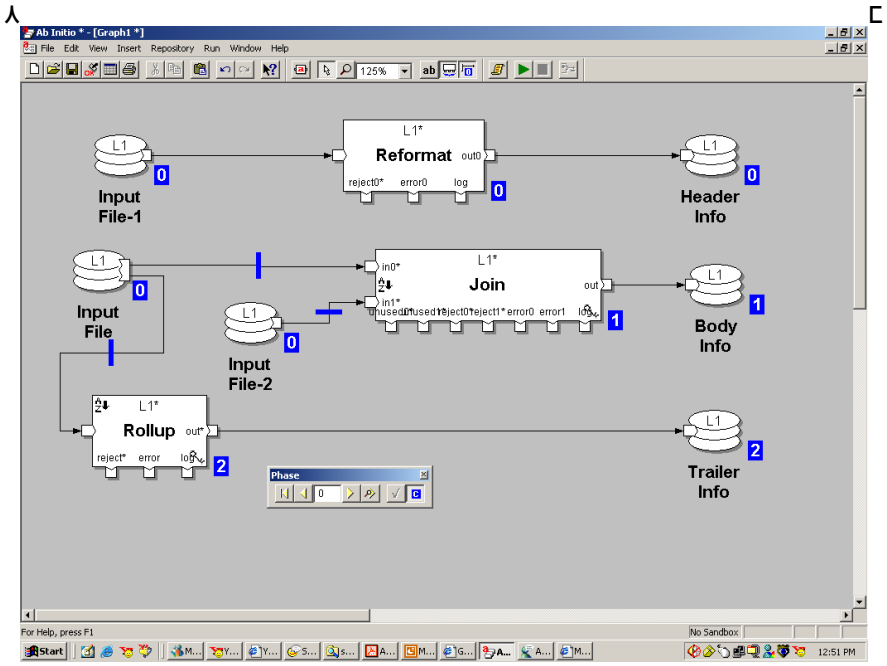
the requirement is to generate a output file with header ,body and trailer information and for this the same file has to be written with different information in different phase i.e. first header , then body and last trailer.

Another example can be mentioned 1st a file being generated. In the later part of the same graph the same file is used as lookup. The file is generated in 1st phase and used as lookup in latter phase.

Check point

A checkpoint is a special time of phasing that saves status information, so as to restart the graph from the point of failure

[the above example can be consulted for this also]



Deadlock

Deadlock occurs when a program cannot progress. It depends on the patterns of your data and typically occurs in graphs with data flows that split then join.

A graph carries a potential deadlock when flows diverge and converge within the same phase. If the flows converge at a component that reads its input flows in a particular order, that component may wait for records to arrive on one flow even as the unread data accumulates on others because components have a limited buffering capacity.

Tip: To avoid deadlock, put components in separate phases, or set flow buffering.

Flow

A flow carries a stream of data between components in a graph. Flows connect components via ⅄ ports ⊏. Ab Initio software supplies four kinds of flows with different patterns:

ㅅ straight ㅅ ㄷㄷ

ㅅ fan-in ㅅ ㄷㄷ

ㅅ fan-out ㅅ ㄷㄷ

ㅅ all-to-all ㅅ ㄷㄷ.

Straight

Straight flows connect ㅅ components ㄷ that have the same number of ㅅ partitions ㄷ. Partitions of components connected with straight ㅅ flows ㄷ have a one-to-one correspondence. Straight flows are the most common flow pattern.

fan-in

Fan-in flows connect ㅅ components ㄷ with a large number of ㅅ partitions ㄷ to components with a smaller number of partitions. The most common use of fan-in is to connect ㅅ flows ㄷ to Departition components. This flow pattern is used to merge data divided into many segments back into a single segment, so other programs can access the data.

When you connect a component running in parallel to any component via a fan-in flow, the number of partitions of the original component must be a multiple of the number of partitions of the component receiving the fan-in flow. For example, you can connect a component running 9 ways parallel to a component running 3 ways parallel, but not to a component running 4 ways parallel.

CAUTION: Although the GDE allows you to use a fan-in flow between components as stated above, the results of using fan-in rather than repartitioning may leave records in the wrong partitions.

fan-out

Fan-out flows connect components with a small number of partitions to components with a larger number of partitions. The most common use of fan-out is to connect flows from partition components. This flow pattern is used to divide data into many segments for performance improvements.

When you connect a Partition component running in parallel to another component running in parallel via a fan-out flow, the number of partitions of the component receiving the fan-out flow must be a multiple of the number of partitions of the Partition component. For example, you can connect a Partition component with 3 partitions via a fan-out flow to a component with 9 partitions, but not to a component with 10 partitions.

To deal with the latter case, Repartition the data by inserting a Departition component after the Partition component. This turns the fan-out flow into an All-to-all flow, allowing a record from any partition of the Partition component to flow into any partition of the target component.

all-to-all

All-to-all flows typically connect components with different numbers of partitions. Data from any of the upstream partitions is sent to any of the downstream partitions. The most common use of all-to-all flows is to repartition data as in the following example

Sandbox Parameters Editor

What is Sandbox?

A sandbox is a collection of graphs and related files that are stored in a single directory tree, and treated as a group for purposes of version control, navigation, and migration.

A sandbox can be a file system copy of a datastore project.

To reach the Sandbox Parameters Editor:

Choose Project > Edit Sandbox on the main menu bar of the GDE.

The Select Path dialog appears.

Do one of the following:

Enter the path to a sandbox directory, beginning with a slash (/).

Click Browse, navigate to the sandbox directory you want, and click Select.

Click OK.

The Sandbox Parameters Editor opens.

Use the Sandbox Parameters Editor to:

Add parameters to a Project, while working in a Sandbox, by specifying information about them in the Sandbox Parameters Editor columns

Edit parameters in a project by changing the information in the Editor's columns.

Each row of the Editor's grid represents one project or parameter. Click the button to the left of any row to select the entire row.

Use the Sandbox Parameters Editor menus and Sandbox Parameters Editor toolbar to perform related tasks. View status messages in the text box across the bottom of the Editor.

Graph Parameters Editor

To reach the Graph Parameters Editor, choose Edit > Parameters on the GDE menu bar.

Use the Graph Parameters Editor to:

Add ⅄ Parameters ⊏ to a graph by specifying information about them in the ⅄ Graph Parameters Editor columns ⊏.

Edit parameters in a graph, including the parameters of any subgraph or component in the graph, by changing the information in the Editor's columns.

Use the ⅄ Graph Parameters Editor menus ⊏ and ⅄ Graph Parameters Editor toolbar ⊏ to perform related tasks.

Each row of the Editor's grid represents one graph, subgraph, component, or parameter. Click the button to the left of any row to select the entire row. View status messages in the box across the bottom of the Editor.

The Graph Parameters Editor serves the same purpose as the tabs of the ⅄ Properties dialog ⊏, but provides access to all the parameters in a graph at once, rather than having to open the Properties dialog separately for the graph, and for each subgraph and component in it.

NOTE: Certain parameters that appear in the Parameters Editor — such as Protection, Condition, config_file, table_spec, Layout, and Port — do not appear as parameters in the Properties dialog. Instead, in the Properties dialog, you specify the values represented by these parameters on the Access, Condition, Description, Layout, and Ports tabs.

Sub-Graph Parameters Editor

The sub-graph parameter is almost as same as Graph Parameter . The difference is only it works for sub-graphs.

Project Parameters Editor

To reach the Project Parameters Editor:

Choose Project > Administrative > Edit Project on the main menu bar of the GDE.

The ⅄ Select Project dialog ⊏ appears.

Do one of the following:

Enter the name of a project, beginning with a slash (/).

Click Browse, navigate to the project you want, and click Select.

Click OK.

The Project Parameters Editor opens.

Use the Project Parameters Editor to:

Add parameters to a Project by specifying information about them in the Project Parameters Editor columns.

Edit parameters in a project by changing the information in the Editor's columns.

Each row of the Editor's grid represents one project or parameter. Click the button to the left of any row to select the entire row.

Use the Project Parameters Editor menus and Project Parameters Editor toolbar to perform related tasks. View status messages in the text box across the bottom of the Editor.

Null

NULL represents the absence of a value. When the Co>Operating® System can't evaluate an expression, it produces NULL. NULL is used in transform functions to force evaluation of a rule with a lower priority.

An expression that produces NULL is also called a failing expression.

The following expression may produce NULL.

If an argument to an operator is NULL, the result is NULL.

If a conditional expression does not have an else-clause, and the test-expression is false, the result of the conditional expression is NULL.

If a field expression refers to an absent conditional field in a record, the result is NULL.

If a field expression refers to a field with a value equal to its Null default value, it is NULL.

In the Aggregate component, a group is defined by the key parameter. At the beginning of a group, the accumulating record is NULL.

In the ⅄ Match Sorted ⊏ component, if an input port lacks a record that matches the current key value, the corresponding input variable is NULL.

In the ⅄ Join ⊏ component, if an input port lacks a record that matches the current key value, and the record required parameter for that port is false, the corresponding input variable is NULL.

If a ⅄ lookup ⊏ or ⅄ lookup_local ⊏ function cannot find a matching key in its lookup file, the result is NULL.

The NULL ⅄ Expression ⊏⅄⊏ is, by definition, NULL.

NOTE: Passing a NULL expression to a function call does not necessarily result in NULL

Functions

Validating Functions

There are number ways (function) to validate NULL

is_defined

Tests whether an expression is not NULL.

The is_defined function returns:

The value 1 if expr evaluates to a non-NULL value.

The value 0 otherwise.

The inverse of is_defined is is_null.

The following are examples

is_defined(123) => 1

is_defined(NULL) => 0

is_valid

Tests whether a value is valid (for all types of datatype)

The is_valid function returns:

The value 1 if expr is a valid data item.

The value 0 if the expression does not evaluate to NULL.

If expr is a record type that has field-validity checking functions, the is_valid function calls each field-validity checking function. The is_valid function returns 0 if any field-validity checking function returns 0 or NULL A data item is valid if it can be used in computation.

Values of decimal type must be able to be interpreted as a number in the specified format.

Values of date type must be able to be interpreted as a date in the specified format and must be an actual date. February 28, 1992 is a valid date, but February 31, 1992 is not.

Each element of a vector type must be valid.

Each field of a record type must be valid. If the record type defines function fields with names beginning with is_valid, then the is_valid function of the record type calls each is_valid function field. For the record type to be valid, each function field must be valid.

Note the distinction between being valid and being defined: a defined item has data, but a valid item has valid data. For example, a decimal field filled with alphabetic characters is defined but not valid.

DML calls is_valid field functions only when is_valid is called on the record, not every time it reads or creates a data record of that type.

The following are examples:

is_valid(1) =>1

is_valid(string_lrtrim("□a□")) =>1 [string_lrtrim a string function which removes spaces from both side of a string if any particular character is not specified]

is_valid(234.234) => 1

is_valid((decimal(8))"□□□") => 0

is_valid((decimal(8))"□a□")=> 0

is_valid(decimal(8)(decimal_strip ("□1□")) =>1[decimal strip is function ,the returned value is string, treated as a decimal number, with leading and trailing spaces, and leading zeros removed. Returns the string 0 if no numeric characters are found.]

is_valid((decimal(8))"1,000") =>0

is_valid((ebcdic decimal(8))"1,000") =>1

is_valid((date("YYYYMMDD"))"19960504")=> 1

is_valid((date("YYYYMMDD"))"19920231") =>0

is_valid((date("YYYYMMDD"))"19920228") =>1

is_valid((date("YYYYMMDD"))"abcdefgh") =>0

is_valid((date("YYYY-MMM-DD"))"1996-May-04") =>1

is_valid((date("YYYY-MMM-DD"))"1996*May&04") =>0

There are some string validation function

is_blank

Tests whether a string contains only blank characters or not.

The is_blank function returns:

- The value 1 if the given string contains only blank characters, or is a zero-length string.

  Because every character in a zero-length string is a blank, this function returns 1 for zero-length strings.

- The value 0 otherwise.

For information about the use of blank characters in Unicode.

The following are examples:

is_blank("") =>1

is_blank("abc") =>0

is_blank(string_lrtrim("□□□")) =>1

is_blank(string_lrtrim ("□□□.□□□")) =>0

There are number of string function which are used for different purpose.

string_compare

The **string_compare** function returns the following after comparing the lexicographic order of *str1* and *str2*:

- **-1**, if the first argument is less than the second
- **0**, if the arguments are equal
- **1**, if the first argument is more than the second

Most built-in DML functions work on Unicode string arguments and produce Unicode string results where specified. For specific information about each supported character set and casting between character sets.

For information about comparing strings using comparison operators.

The following are examples:

string_compare("abcd", "abcd")=> 0

string_compare("aaaa", "bbbb") =>-1

string_compare("bbbb", "aaaa") =>□□1

string_compare("aaaa", "a") =>1

string_compare("AAAA", "aaaa") =>-1

string_compare((ebcdic string(4))"AAAA",(ebcdic string(4))"aaa" =>1

String Functions

There are number of functions to manipulate different string

string_filter_out

The **string_filter_out** function returns a string containing the characters that appear in *str1* but do not appear in *str2*. The function drops characters that do appear in *str2*. The order of the characters in the returned string is the same as the order in which they appear in *str1*.

The following are examples that use the **string_filter_out** function:

string_filter_out("AXBYCZ", "ABCDEF") =>"XYZ"

string_filter_out("Apt. #2", ".#,%") =>"Apt 2"

string_filter

The **string_filter** function returns a string containing the characters that appear in both *str1* and *str2*. The function drops characters that do not appear in *str2*. The order of the characters in the returned string is the same as the order in which they appear in *str1*.

The following are examples that use the **string_filter** function:

string_filter("AXBYCZ", "ABCDEF") => "ABC"

string_filter("023#46#13", "0123456789") =>"0234613" .

string_replace

The **string_replace** function replaces each non-overlapping occurrence of *seek* in *str* with *new* and returns the resulting string. If *seek* is zero-length, the function inserts *new* before each character and after the last, and returns the resulting string.

The following are examples that use **string_replace**:

string_replace("a man a plan a canal", "an", "ew") => "a mew a plew a cewal"

string_replace("a man a plan a canal", "ship", "boat") => "a man a plan a canal"

string_replace("abcde", "", "*") =>"*a*b*c*d*e*"

string_replace(U"ab", U"b", U"\u0099") => U"a\u0099"

string_substring

The **string_substring** function returns a substring of a string. You specify the substring by indicating the index of the first character (*start*) and the length of the substring.

If *start* is less than **1**, the function uses **1** as the value *start*.

If *length* is less than **1**, the function uses **0** as the value of *length*.

If *start* is greater than the length of the string, the function returns "".

If *length* is greater than the length of the string, the function returns just the available characters.

Strings are indexed starting at **1**.

The following are examples that use **string_substring**:

string_substring("John Smith", 8, 5) =>"ith"

string_substring("abcdefgh", 4, 3)=>"def"

string_substring("qqqqqqqq", 9, 3) ""

string_substring("abcdefgh", 3, 12)=>"cdefgh"

string_substring("abcdefgh", -3, 4) =>"abcd"

Other Important Functions

next_in_sequence

This function is used to identify sequence of sequence in particular partition .It basically returns a sequence of integers on successive calls, starting with **1.**

Each partition of each component has its own private sequence. The sequence of numbers returned is shared among all callers within a component. So, for example, calls to

next_in_sequence from different transform functions associated with different outputs of a Reformat component do affect one another.

The following are examples:

next_in_sequence()=>1

next_in_sequence()=>2

next_in_sequence()=>3

Usage

To select the 1st record of a partition or of a serial flow in a fiter_by_exppression component the following statement is used

next_in_sequence() ==1;

It can also be used to add unique key for each record of a serial flow in a transform component using the following statement.

```
let integer(8) expr_1 =next_in_sequence();
```

later the ***expr_1*** variable should be assigned to field of the output dml

[for mfs see later]

this_partition

Returns the partition number of the component from which the function was called.

The **this_partition** function returns a number in the range of **0** and **number_of_partitions** ( ) **-1**, inclusive. The returned number represents the number of the component partition that called **this_partition**. If a component did not call **this_partition**, it returns **-1**.

number_of_partitions

The **number_of_partitions** function returns:

- The number of partitions of the component that called the function. The number of partitions is also known as the degree of parallelism.
- The value **-1** if not called from within a component.

For mfs to identify a particular record of the flow or adding unique key to each record of the flow the following expression is used instead of using only next_in_sequence() function

next_in_sequence()*number_of_partitions() + this_partition();

Tuning Tips

- Avoid Sorts
- Use Lookups
- Use In-memory Join/Rollup
- Allocate memory correctly
- Phasing
- Minimize number of components

[All the above mentioned points are discussed already. For minimizing number of components it has been discussed for individual components]