

NULL_IF_ERROR

Produces a NULL value instead of an error. This is another name for the [fail_if_error](#) function.

Syntax:

object fail_if_error(expr, ...)

Details: Fail_if_error function produces a NULL value instead of an error, when an expression would have otherwise caused an error. An error causes immediate rejection, while producing a NULL value enables u to use prioritized rules to try other ways of computing a result. You can use this function to control errors that can arise from dirty data.

The function returns the value of the first argument whose evaluation is successful — that is, does not generate an error. It returns NULL if all expressions evaluate to NULL.

Example:

Fail_if_error(123) → 123

Fail_if_error(NULL) → Null

Fail_if_error(1/0) → Null

first_without_error

Returns the first non-error value of from one to 26 arguments.

Syntax:

object first_without_error(expr , ...)

Details: The function tests from 1 to 26 expressions in sequence: it checks each expression for errors, stops at the first non-error, & returns that value. If all arguments return errors, the function returns an error.

This function is useful for specifying actions for ur app'n to perform. For ex, u could use this function to specify what happens in cases where problems are encountered while loading lookup tables.

Example: This example shows how to use the return value from ***first_without_error*** to write a log message if it encounters an error on loading a lookup table. It contains two expressions to evaluate for a non-error: the built-in functions [lookup_load](#) and [lookup_not_loaded](#).

If [lookup_load](#) fails due to a missing data or index error, first_without_error returns the value of [lookup_not_loaded](#):

```
let lookup_identifier_type lu = first_without_error(lookup_load(missing_datapath,
absent_index, "My Lookup Template"), lookup_not_loaded());
```

```
if (lu == lookup_not_loaded())
```

```
write_to_log("snafu", string_concat("lookup_load failed:", last_error()));
```

is_valid

Tests whether the result of an expression is valid or — when called on a record — calls all validity-checking function fields in the record.

Syntax:

long is_valid (dml_expression expr)

Details: This function checks the validity of the DML expression expr. The function returns:

1 if the result of evaluating expr is valid
0 if the result of evaluating expr is invalid or evaluates to NULL

NOTE: The terms valid & defined are not synonymous: a defined item has data (it does not evaluate to NULL); a valid item has valid data. For Ex, a decimal field populated with alphabetic characters is defined, but is not valid.

This function can also check the validity of records.

Checking the validity of records: To use is_valid to check the validity of fields in a record, call the function on a record that contains validity-checking function fields. This causes each validity-checking function field in the record to be called. The record is considered valid if it meets the criteria described in ["Validity criteria for DML objects"](#).

Validity criteria for DML objects: The following table describes validity criteria — that is, criteria that must be met for is_valid to return 1 (true) — for DML objects:

Examples:

Is_valid(1) → 1

Is_valid(" a ") → 1

Is_valid(234.234) → 1

Is_valid((decimal(8))" ") → 0

Is_valid((decimal(8))" a ") → 0

Is_valid((decimal(8))" 1 ") → 0

What are .abi-unc files?

Short Ans: .abi-unc files are temporary files used by the checkout process. Under normal circumstances you should not see them. However, if the checkout failed or was interrupted, these files can be left behind. Delete them and try checking out again.

Details: The checkout procedure includes two steps to ensure that a checkout does not leave your sandbox in a half-checked-out state:

1. It checks out the files, but gives them the .abi-unc suffix (for "uncommitted").

2. Once every file is successfully checked out, the .abi-unc files replace the real files. If an error occurs midway through checkout, any .abi-unc files that have been created might be left behind.

If u receive the error msg "Cannot write project parameters file: /path/filename.abi-unc" during checkout & u see .abi-unc files, delete them & try checking out again. Uncommitted files r usually left behind because a checkout process was aborted.

Does the EME maintain a log of all operations performed?

Short Ans: Yes, EME maintains an audit trail that enables administrators to keep track of many EME operations (especially those that modify the datastore) and many air and aiw command invocations. Logging is implemented using two text files that reside in the datastore directory:

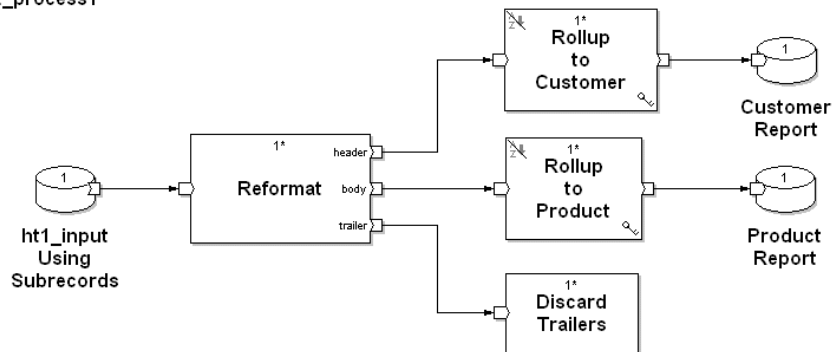
datastore-name.audit is the current log of entries. The file is world readable and writable.

datastore-name.audbak is a backup file that contains the entries the .audit file had when it last reached its maximum number of entries.

Generating separate summaries of header & body subrecords

The graph ht_process1.mp processes the subrecord view of the data to generate summaries of the customer data and the product data:

ht_process1



The graph does this as follows:

1. Uses a [REFORMAT](#) to split out the different subrecord types:

- a. Sets the output_index parameter as follows so that header records (those with kind H) will be sent to the first output's transform function (transform0), body records to the second (transform1), and trailer records to the third (transform2):

```
out :: output_index(in) =
begin
    out :: if (in.kind == 'H') 0 else if (in.kind == 'B') 1
    else 2;
end;
```

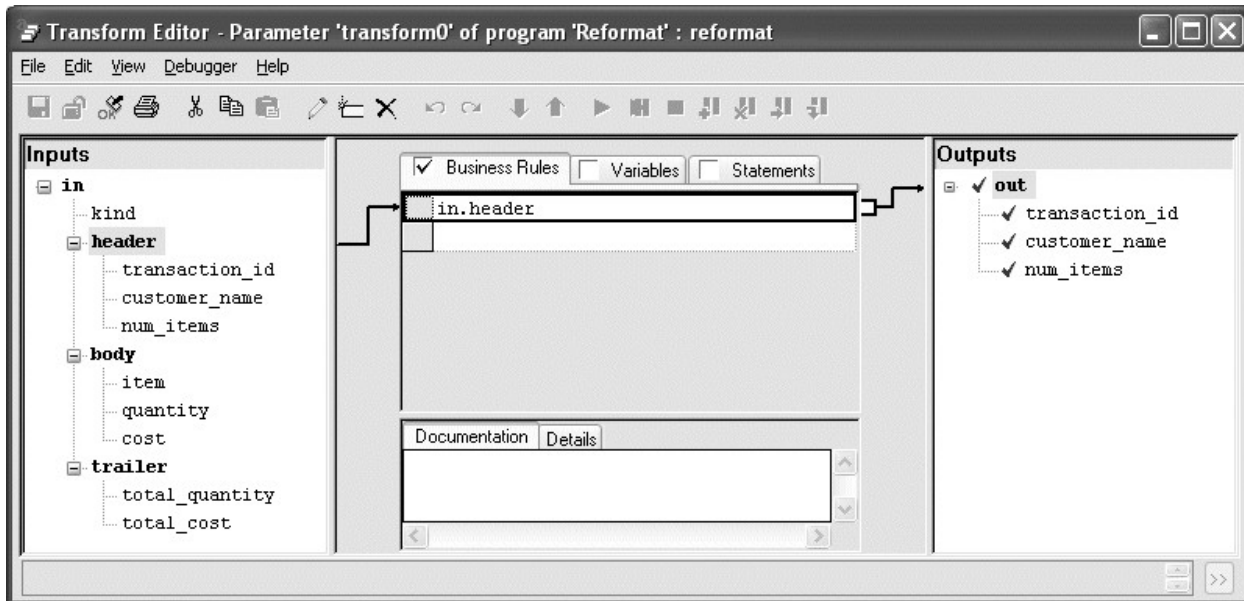
- b. Defines the following three transforms to produce the respective outputs:

```
transform0
out :: reformat(in) =
begin
out :: in.header;
end;
```

```
transform1
out :: reformat(in) =
begin
out :: in.body;
end;
```

```
transform2
out :: reformat(in) =
begin
out :: in.trailer;
end;
```

For example, transform0 looks like this in Grid View:



2. Uses two [ROLLUP](#) components to summarize the following data in the output:

Customer data (number of store visits and number of items bought by each customer):

```
out::rollup(in) =
begin
    out.customer_name :: in.customer_name;
    out.num_store_visits :: count(1);
    out.total_num_items :: sum(in.num_items);
end;
```

Product data (item, quantity sold, and average cost):

```
out::rollup(in) =
begin
    out.item :: in.item;
    out.total_quantity :: sum(in.quantity);
    out.average_cost :: avg(in.cost);
end;
```

3. Sends the trailer records to a [TRASH](#) component:

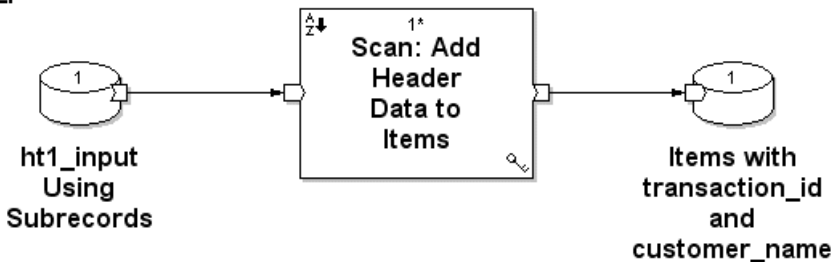
The Customer Report output of ht_process1.mp looks like this:

View Data: Customer Report			
File Edit View Help			
More Records: 100 Go <input type="checkbox"/> Clear Display			
	customer_name	num_store_visits	total_num_items
1	Frank	1	0
2	Joe	2	4
3	Koko	1	3
4	Veronica	1	2
5	Mary	2	3
6	Susan	1	5
[EOF]			
Scanned 6 records. Retrieved 6 matching selection. (EOF)			

Distributing header information to body records

The graph ht_process2.mp uses a [SCAN](#) to process the subrecord view of the data & generate a dataset containing records of a single type that combines info from header & body records.

ht_process2



The Scan does this as follows:

1. Sets the key-method parameter to Use key_change function and defines the function as shown below. This causes the Scan to execute its initialize() function whenever it encounters a record whose kind field contains H.

```

out :: key_change(previous, current) =
begin

```

```

    out :: current.kind == 'H';

```

```

end;

```

2. Defines the temporary_type as a record consisting of fields for the transaction ID and customer name:

```

type temporary_type =

```

```

record

```

```

    decimal("") transaction_id;

```

```

        string("") customer_name;
    end;

```

The temporary variable is used in the scan() function (see Step [a](#) in Step [4](#) below) to carry information from the header record to each of the other records in the group.

3. For each input record whose kind is H (and which therefore denotes the start of a new key group), the Scan initializes the temporary record to hold the contents of the two fields in the header record that match the fields in the temporary_type (that is, transaction_id and customer_name):

```

    temp :: initialize(in) =
    begin
        temp :: in.header;
    end;

```

4. For all records, the Scan does the following:

a. Returns a temporary record with the content of the current temporary record:

```

    out :: scan(temp, in) =
    begin
        out :: temp;
    end;

```

This causes the current transaction_id and customer_name fields from the header record to be written to the temporary record for each subsequent body and trailer record in the key group initialized for that header record.

b. Executes its finalize() function, outputting the content of the temporary variable and the values of the item, quantity, and cost fields:

```

    out :: finalize(temp, in) =
    begin
        out.transaction_id :: temp.transaction_id;
        out.customer_name :: temp.customer_name;
        out.item :: in.body.item;
        out.quantity :: in.body.quantity;
        out.cost :: in.body.cost;
    end;

```

NOTE: The Scan's reject-threshold parameter is set to Never abort to prevent the graph from failing when the Scan tries to assign the in.body fields item, quantity, and cost (which do not exist in the header or trailer records) to the correspondingly named output fields.

The output of ht_process2.mp looks like this:

View Data: Items with transaction_id and customer_name

File Edit View Help

More Records: 100 Go ☐ Clear Display

	transaction_id	customer_name	item	quantity	cost
1	334566	Veronica	apple	1	0.45
2	334566	Veronica	roast	1	7.85
3	334567	Joe	apple	1	0.45
4	334567	Joe	orange	2	0.55
5	334567	Joe	tomato	2	0.35
6	334568	Mary	chicken	2	5.99
7	334570	Susan	orange	5	0.55
8	334570	Susan	lettuce	1	0.95
9	334570	Susan	milk	4	4.99
10	334570	Susan	roast	1	6.25
11	334570	Susan	bread	2	2.55
12	334571	Koko	chicken	2	5.27
13	334571	Koko	orange	3	0.55
14	334571	Koko	tomato	1	0.35
15	334572	Mary	apple	2	0.45
16	334572	Mary	milk	1	4.99
17	334573	Joe	tomato	3	0.35
	[EOF]				

Scanned 17 records. Retrieved 17 matching selection. (EOF)

Normalizing vector data to header and body records

The graph `ht_process3.mp` (in the headers sandbox) produces same output as `ht_process2.mp` (a dataset containing records of a single type that combines info from header & body records) but uses a [NORMALIZE](#) to process the vector view of the data:

`ht_process3`



The Normalize does the following:

1. Uses the `num_items` field from each header record to specify the number of body records to output for each transaction:

```

out :: length(in) =
begin
    out :: in.header.num_items;

```

end;

2. Outputs (for each transaction) a record consisting of the header fields transaction_id and customer_name, along with the body fields item, quantity, and cost.

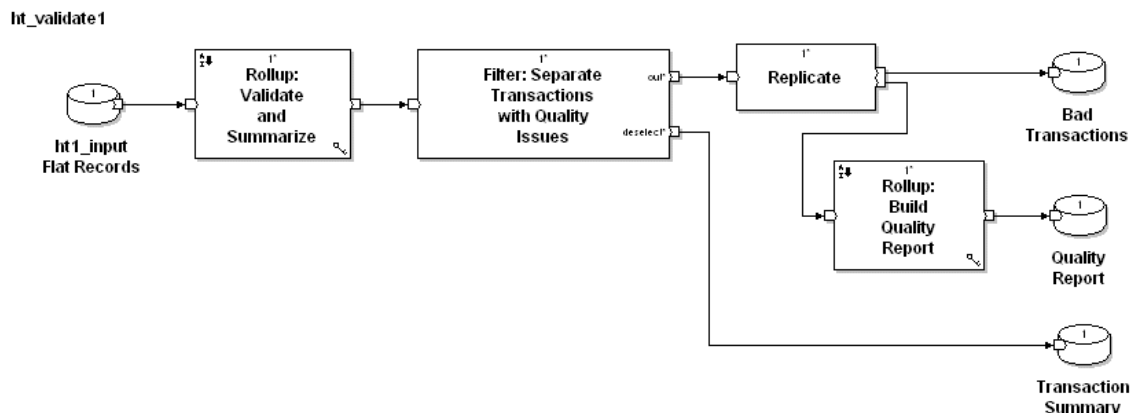
```
out :: normalize(in, count) =  
begin  
    out.transaction_id :: in.header.transaction_id;  
    out.customer_name :: in.header.customer_name;  
    out.item :: in.body[count].item;  
    out.quantity :: in.body[count].quantity;  
    out.cost :: in.body[count].cost;  
end;
```

Validating header/body/trailer input data

In some situations u may need to assess the quality of data across a group of records consisting of a header record, some number of body records, & a trailer record. Exactly how to do this depends on the representation of the data. The following topics describe how to assess the quality of data represented in two ways:

Validating flat header/body/trailer input data

The graph ht_validate1.mp shows how to validate the flat representation of the input data described in ["Flat record format to describe data with headers, bodies, and trailers"](#):



The components of ht_validate1.mp do the following:

1. The Rollup: Validate and Summarize component does what its name suggests, as follows:

c. Sets the key-method parameter to Use key_change function, and defines the function as shown below. This causes the [ROLLUP](#) to treat each set of header, body, and trailer records (which begins with a record whose kind field is H) as a separate group.


```

out :: key_change(previous, current) =
begin
    out :: current.kind == 'H';
end;

```

d. Creates variables to hold (for each group of header, body, and trailer records) the totals calculated for the different items purchased, the quantity, and the cost:

```

out :: rollup(in) =
begin
    let decimal("") actual_num_items = first_defined(count(in.item), 0);
    let decimal("") actual_total_quantity = first_defined(sum(in.quantity),
0);
    let decimal("") actual_total_cost = first_defined(sum(in.quantity *
in.cost), 0);

```

NULL values in [ROLLUP](#) aggregation functions

In the conditional representation used for the flat data, fields that are not part of a given kind of record are NULL. (For Ex, quantity field of all header & trailer records in ht_input Flat Records is NULL.) Aggregation functions such as sum() handle these records by simply ignoring NULL values.

first_defined() function

The first_defined() function returns the first non-NULL argument passed to it. Here, it ensures that for any groups that do not have any body records, zero will be returned instead of NULL.

Outputs fields from the header and trailer records, along with booleans indicating whether any of the calculated values in the original data are incorrect:

```

out.transaction_id :: first(in.transaction_id);
out.customer_name :: first(in.customer_name);
out.mismatched_num_items :: actual_num_items != first(in.num_items);
out.mismatched_total_quantity :: actual_total_quantity !=
last(in.total_quantity);
out.mismatched_total_cost :: actual_total_cost != last(in.total_cost);
out.num_items :: first(in.num_items);
out.total_quantity :: last(in.total_quantity);
out.total_cost :: last(in.total_cost);
end;

```

first() and last() aggregation functions

The first() and last() functions return values from the first and last records in a key group. In ht_validate1.mp, these are the header and trailer records, respectively.

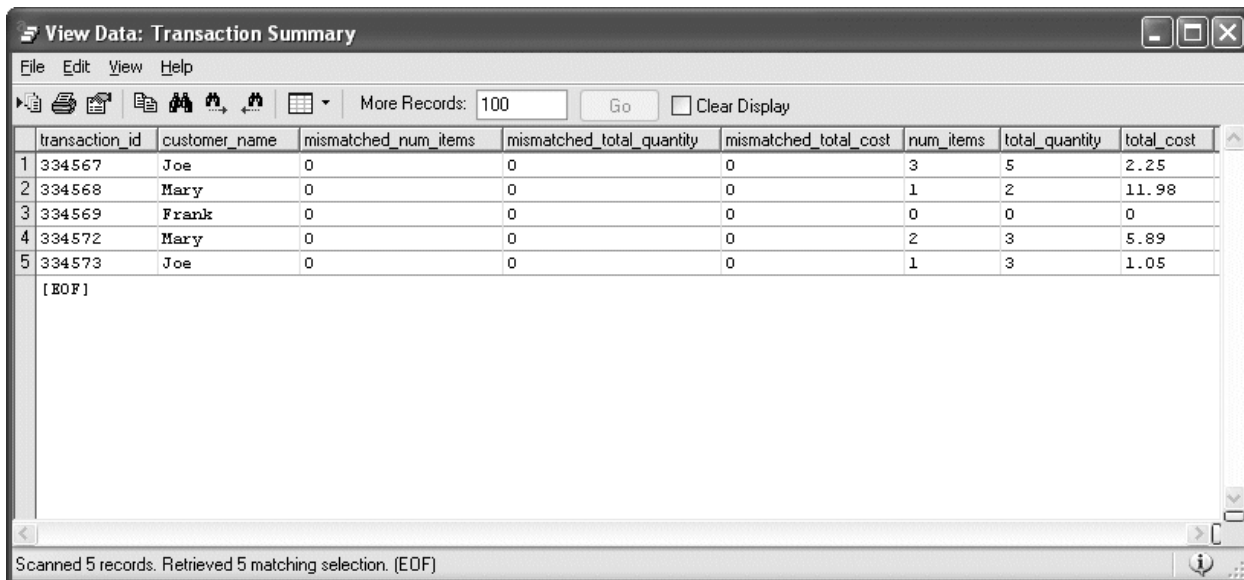
2. The [FILTER BY EXPRESSION](#) component sets its select_expr parameter as follows, so that its out port emits only records for which at least one of the mismatched fields is true:

mismatched_num_items or

mismatched_total_quantity or

mismatched_total_cost

Records that do not have mismatched numbers are sent to the [FILTER BY EXPRESSION](#)'s deselect port, and from there to the Transaction Summary output file:



The screenshot shows a window titled "View Data: Transaction Summary" with a menu bar (File, Edit, View, Help) and a toolbar. Below the toolbar is a table with 8 columns: transaction_id, customer_name, mismatched_num_items, mismatched_total_quantity, mismatched_total_cost, num_items, total_quantity, and total_cost. The table contains 5 rows of data. Below the table, the text "[EOF]" is visible. At the bottom of the window, a status bar reads "Scanned 5 records. Retrieved 5 matching selection. (EOF)".

	transaction_id	customer_name	mismatched_num_items	mismatched_total_quantity	mismatched_total_cost	num_items	total_quantity	total_cost
1	334567	Joe	0	0	0	3	5	2.25
2	334568	Mary	0	0	0	1	2	11.98
3	334569	Frank	0	0	0	0	0	0
4	334572	Mary	0	0	0	2	3	5.89
5	334573	Joe	0	0	0	1	3	1.05

[EOF]

Scanned 5 records. Retrieved 5 matching selection. (EOF)

3. The [REPLICATE](#) copies the output of the Filter by Expression so that it can be used for two purposes:

To generate a report containing the unmodified output of the Filter by Expression — that is, all transactions with mismatched totals. (See the Bad Transactions output dataset below.)

To calculate the total number of records for each type of mismatch (see Step [4](#)).

View Data: Bad Transactions

File Edit View Help

More Records: 100 Go ☐ Clear Display

	transaction_id	customer_name	mismatched_num_items	mismatched_total_quantity	mismatched_total_cost	num_items	total_quantity	total_cost
1	334566	Veronica	0	0	1	2	2	9.68
2	334570	Susan	0	0	1	5	13	36.61
3	334571	Koko	0	1	1	3	5	999.99

[EOF]

Scanned 3 records. Retrieved 3 matching selection. (EOF)

4. The Rollup: Build Quality Report component calculates the sums of the mismatched fields from all records and sends them to the Quality Report output dataset:

out::rollup(in) =

begin

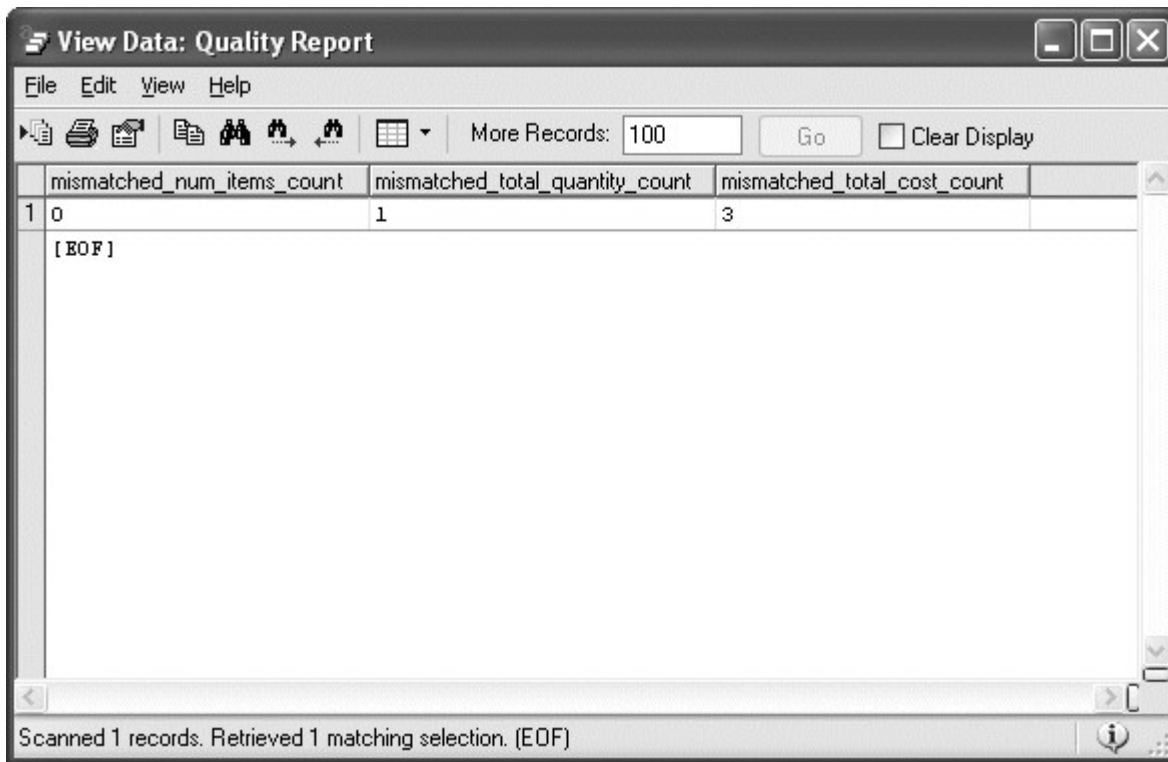
```

    out.mismatched_num_items_count ::
    sum(in.mismatched_num_items);
    out.mismatched_total_quantity_count ::
    sum(in.mismatched_total_quantity);
    out.mismatched_total_cost_count ::
    sum(in.mismatched_total_cost);

```

end;

The output looks like this:



force_error

Raises an error and sends the specified message to the component's error or similar port.

Syntax:

`force_error(string message)`

Details: This function does not return a value. It sets the value that [last_error](#) returns.

The `force_error` message is written to the error or similar port of the component from which this function is called.

`force_error` can be used to filter records having particular values or to handle records with badly formatted data:

4. Test for the value to filter. If the value is found, call `force_error` with an appropriate message.
5. Set the component's reject-threshold parameter to one of the following:
 Never abort — To filter out all badly formatted records
 Limit/Ramp — To set a limit on the number of badly formatted records that are acceptable before stopping the graph
6. To view the filtered data, attach an Output File or Trash component to the component's error port and place a watcher on the flow.

Examples

Example 1

This simple example shows how to specify the syntax for this function:

```
if (trans_code > 1000) force_error("Value is out of bounds.")
```

Example 2

This longer example presents a context in which you might use force_error.

Suppose ur graph includes a Join component & u want to filter out transactions not from Region 1 from the join process. If you were to use the Join's select parameter, the filtered records would go to the unused port along with other records that are unused due to the type of join.

Suppose, however, that u want to distinguish bet'n the two types of unused records. You can use force_error to filter records instead of the select parameter: this will send the filtered records to the reject0 port, thus keeping them separate from those that go to the unused port. To do this:

7. Remove the filter expression from the select parameter of the JOIN.

8. Set the reject-threshold parameter of Join to Never abort. (If reject-threshold is set to Abort on first reject, the first record sent to the reject0 port will cause the graph to fail.)

9. Add force_error to the Join transform function. For example, to filter records not from Region 1, enter:

```
if ( in1.region != 1 ) force_error("Transaction NOT from Region 1")
```

NOTE: Both true Ab Initio errors such as record formatting errors and data type errors will also be sent to the reject0 port with the "Transaction NOT from Region 1" message.

first_defined

Returns the first defined (non-NULL) value of from two to 26 arguments.

Syntax:

```
object first_defined( a , b , c , d , ... )
```

Details: This function is useful in assigning default values to input fields whose values are NULL. The function tests from 2 to 26 values in sequence: it checks each value to see if the value is defined, stops at the first defined value, and returns that value.

If all arguments evaluate to NULL, this function returns NULL.

NOTE: The Oracle NVL function is similar to this function.

Examples

```
First_defined("test", "jan", "feb", "mar") → "test"
```

```
First_defined(NULL, "jan", "feb") → "jan"
```

```
first_defined(NULL,42,43,44) → 42
```

first_without_error

Returns the first non-error value of from one to 26 arguments.

Syntax:

object first_without_error(expr , ...)

Details: Function tests from 1 - 26 expressions in sequence: it checks each expression for errors, stops at 1ST non-error, & returns that value. If all arguments return errors, the function returns an error.

This function is useful for specifying actions for ur app'n to perform. Ex, u could use this function to specify what happens in cases where problems are encountered while loading lookup tables.

Example

This example shows how to use the return value from first_without_error to write a log message if it encounters an error on loading a lookup table. It contains two expressions to evaluate for a non-error: the built-in functions [lookup_load](#) and [lookup_not_loaded](#).

If [lookup_load](#) fails due to a missing data or index error, first_without_error returns the value of [lookup_not_loaded](#):

```
let lookup_identifier_type lu = first_without_error(lookup_load(missing_datapath,
absent_index, "My Lookup Template"), lookup_not_loaded());
```

```
if (lu == lookup_not_loaded())
```

```
write_to_log("snafu", string_concat("lookup_load failed:", last_error()));
```

force_abort

Sends the specified message to the standard error stream and exits the process with error status.

Syntax:

force_abort(string message)

Details: If this function is called in a component, it causes the graph containing the component to abort. U can use this function to terminate a graph if a component's transform logic encounters a severe error, and the component's reject-threshold parameter is set to Never abort.

This function does not return a value. Error functions like [null_if_error](#) and the reject-threshold parameter's limit/ramp setting do not affect the behavior of force_abort.

Example

In this transform example, the REFORMAT component's reject-threshold parameter is set to Never abort. However, if the value of abort_field is 3, the graph will abort and

display the supplied error message (in this case to the GDE Job tab in the Application Output window):

```
out::reformat(in)=
begin
  if (in.abort_field == 3)
    force_abort("Aborting component.");
    out.a :: "OK";
  end;
```

The error message is:

Error from Component 'Reformat', Partition 0

[M1080]

Aborting component.

[Show Details] [Go to Component]

ABINITIO (2.15.2.r24.0): Tue May 6 16:32:29 2008

ABINITIO: Rolling back ... Done.

ABINITIO: No checkpoint detected. About to close job and delete recovery file
/disk2/sand/mkatz/functions/run/force_abort.rec

ABINITIO: Job closed; recovery file deleted.

Failed

is_error

Tests whether an error occurs while evaluating an expression.

Syntax:

```
int is_error( expr )
```

Details: You can use this function to control errors that can arise from dirty data.

This function returns:

1 if evaluating the expression results in error

0 otherwise

The following table compares `is_error`, [is_defined](#), and [is_null](#).

If expr results in	is_error returns	is_defined returns	is_null returns
Error	1	error	error
NULL	0	0	1

Success	0	1	0
---------	---	---	---

Examples

is_error(a + b) 0

is_error(1 / 0) 1

size_of

Returns the size of a specified value in bytes, including delimiter characters.

Syntax:

long size_of(value)

Details: The function returns NULL if value is NULL.

Examples:

This group of examples shows how different DML types return different byte sizes.

String examples:

size_of("10001") 5

size_of("abcdefghi") 9

An integer example, followed by a comparison of the integer 1 and the string 1:

size_of(1000000000) 9

size_of(1) 8

size_of("1") 1

A decimal example:

size_of(42.65) 5

A unicode example:

size_of(U"abc\u0064\u0065") 10

What does the error message "Failed to allocate <n> bytes" mean?

Short Ans: This error message is generated when an Ab Initio process has exceeded its limit for some type of memory allocation.

Three things can prevent a process from being able to allocate memory:

The user data limit (ulimit -Sd and ulimit -Hd). These settings do not apply to Windows systems.

Address space limit.

The entire computer is out of swap space.

Details: Sometimes an Ab Initio process might try to allocate a resource beyond its available limit. When this happens, an [error message](#) with the following format is displayed:

```
===== Error from <COMPONENT_NAME> on <MACHINE_NAME>
=====
```

Failed to allocate A bytes.

Current data limit is B bytes.

Current vmem limit is C bytes.

Heap is at least D bytes already.

This basic message is sometimes followed by additional information, most commonly one of the following messages:

Allocation context stack: attempting to create LookupFile "<lookupfile>.lookup"

Swap space: E pages free of F pages total

Fork failed: Not enough space

Swap space: G pages free of H pages total.

What does the error message "File table overflow" mean?

Short Ans: This error msg indicates that system-wide limit on open files has been exceeded. Either there r too many processes running on the system, or the kernel configuration needs to be changed.

Details: This error msg might occur if maximum number of open files allowed on the machine is set too low, or if max-core is set too low in components that r processing large amounts of data. In the latter case, much of the data processed in a component (such as [SORT/JOIN](#) component) spills to disk, causing many files to be opened. Increasing value of max-core is an appropriate 1st step in the case of a sort, because it reduces the number of separate merge files that must be opened at the conclusion of the sort.

NOTE: Because increasing max-core also results in the memory requirements of your graph increasing be careful not to increase it too much (and you might need to consider changing the graph's phasing to reduce memory requirements). It is seldom necessary to increase max-core beyond 100MB.

If the error still occurs, see ur system administrator. Note that kernel setting for the maximum number of system-wide open files is OS-dependent (for example, this is the nfile parameter on Unix systems), and, on many platforms, requires a reboot in order to take effect. See the Ab Initio Server Software Installation Guide for Unix for the recommended settings.

What does the error message "Failed to allocate <n> bytes" mean?

Short Ans: This error message is generated when an Ab Initio process has exceeded its limit for some type of memory allocation.

3 things can prevent a process from being able to allocate memory:

User data limit (ulimit -Sd & ulimit -Hd). These settings do not apply to Windows systems.

Address space limit.

The entire computer is out of swap space.

Details: Sometimes an Ab Initio process might try to allocate a resource beyond its available limit. When this happens, an [error message](#) with the following format is displayed:

```
===== Error from <COMPONENT_NAME> on <MACHINE_NAME>
=====
```

Failed to allocate A bytes.

Current data limit is B bytes.

Current vmem limit is C bytes.

Heap is at least D bytes already.

This basic message is sometimes followed by additional information, most commonly one of the following messages:

Allocation context stack: attempting to create LookupFile "<lookupfile>.lookup"

Swap space: E pages free of F pages total

Fork failed: Not enough space

Swap space: G pages free of H pages total.

What does the error message "broken pipe" mean?

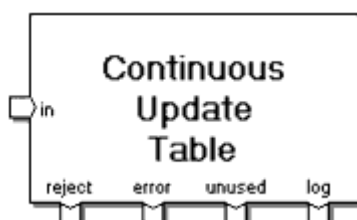
Short Ans: This error msg means that a downstream component has gone away unexpectedly, so the [flow](#) is broken. Ex, the database might have run out of memory making database components in the graph unavailable.

Details: In general, broken pipe errors indicate the failure of a downstream component, often a custom component or a database component. When the downstream component failed, the named pipe the component was writing to broke.

In majority of cases, the problem is that the database ran out of memory, or some other problem occurred during database load. There could be a networking problem, seen in graphs running across multiple machines where a TCP/IP problem causes the sender to see a "Connection reset by peer" message from the remote machine.

If a component has failed, you typically see either of two scenarios.

CONTINUOUS UPDATE TABLE



Purpose: Continuous Update Table modifies database tables by executing embedded SQL UPDATE, INSERT, or DELETE statements.

Continuous only: Continuous Update Table can be used only in continuous graphs.

Parameters for CONTINUOUS UPDATE TABLE

Continuous: ([boolean](#), required) Specifies whether this component will work correctly in a continuous graph — specifically, whether recovery will work correctly.

True — the component will save its internal state and be fully recoverable in a continuous graph.

False — the component will not save its state and will not be recoverable in a continuous graph.

Default is True.

NOTE: The continuous parameter is ignored in batch graphs.

DBConfigFile: ([filename](#), required): Name of the database table configuration file.

Dbms: ([string](#), required) Type of database. Must match the setting of the dbms tag in the .dbc file specified in the [DBConfigFile](#) parameter.

UpdateSqlFile: ([filename](#), required) Reference to an SQL statement (UPDATE, INSERT, DELETE) in embedded SQL format that is executed against the specified table.

InsertSqlFile: ([filename](#), optional) Reference to an SQL statement (UPDATE, INSERT, or DELETE) in embedded SQL format that is executed against the specified table.

CommitNumber: Ignored. (But note that you will get an error if you set commitNumber to 1 or more and do not specify commitTable.) Continuous Update Table uses only compute points and checkpoints to decide when to commit.

CommitTable: ([tablename](#), required) Database table in which to store commit progress. This is an internally used table; it is not the table against which the SQL is executed. The target table is specified in the SQL. The table (if specified) must be created by running a utility script, & is only for internal use by Continuous Update Table.

Continuous Update Table always commits on compute points & checkpoints.

Note that a commit table is never required for Continuous Update Table as long as commitNumber is less than 1.

Jobid: (optional) Unique job identifier. In general, you should not specify a value for this parameter. See ["How to use the jobid parameter"](#) for the update table component.

Database string — '\$' is reserved; use \${ } for env substitution.

updateSqlOnceOnly

Not used.

ActionRequired: ([boolean](#), optional)

True — Continuous Update Table issues an error message ("No data in the database was modified") when an input record results in no update to the table.

Continuous Update Table writes this error message to the error port, and sends the unused record to the reject port. If the reject port is not connected, Continuous Update Table (and the graph) fails.

False — Continuous Update Table does not issue an error if the input record results in no update to the table. This input record is discarded unless you have connected the unused port.

Default is True.

Teradata parameters for CONTINUOUS UPDATE TABLE

Note that all of these parameters (except `ter_interface`, `logfile`, & `vartext`) are passed directly to the utility specified by `ter_interface` & have interface-specific meanings.

Note also that none of these parameters (except `ter_interface`) appear if the value of the `ter_interface` parameter is `api`.

Alphabetical list of Teradata parameters

[brief](#)
[checkpoint](#)
[errlimit](#)
[errpercent](#)
[errtab_name](#)
[ignore_dupl_insert](#)
[ignore_dupl_update](#)
[ignore_missing_delete](#)
[ignore_missing_update](#)
[logfile](#)
[logtab_name](#)
[macrodb](#)
[np_axsmod_dir](#)
[pack](#)
[rate](#)
[serialize](#)
[sessions](#)
[sleep](#)
[tenacity](#)
[ter_interface](#)
[vartext](#)

Teradata parameter descriptions

ter_interface: ([choice](#), optional) T pump or api. Default is api.

Brief: ([boolean](#), optional)

True — Continuous Update Table generates only brief log information.

False — Continuous Update Table generates full log information.

Available only when `ter_interface` is `Tpump`.

Sleep: ([integer](#), optional) Number of minutes to wait before retrying logon. Available only when `ter_interface` is `Tpump`.

Tenacity: ([integer](#), optional) Maximum number of hours during which to continue to retry logon before giving up. Available only when `ter_interface` is `Tpump`.

Sessions: ([integer](#), optional) Number of sessions. Available only when `ter_interface` is `Tpump`.

Vartext: ([boolean](#), optional) If True, send data to Teradata utility in vartext mode. Available only when `ter_interface` is `Tpump`. Default is False.

logtab_name: ([string](#), optional) User-specified logtable name. Available only when `ter_interface` is `Tpump`. The name is autogenerated if not specified by the user here.

np_axsmo_dir: ([pathname](#), required) Name of directory for Named Pipe AXSMOD storage. Available only when `ter_interface` is `Tpump`.

One checkpoint's worth of data is stored in this directory. The Named Pipe AXSMOD is always used on Solaris & MP-RAS, & usually on other platforms when available; but the directory must be specified regardless of the platform being used. Other platforms use this directory to hold data on graph restart; on these platforms the directory should contain enough space for all data to be loaded.

Checkpoint: ([integer](#), optional) Checkpoint rate to be used by Teradata utility. Available only when `ter_interface` is `Tpump`.

Logfile: ([filename](#), optional) Absolute pathname of file in which to store load utility log info. Available only when `ter_interface` is `Tpump`. This file is in effect an alternative to the log port.

errtab_name: (optional) User-specified errtable name. Available only when `ter_interface` is `Tpump`.

There might actually be 2 errtables, & specified string will have 1 & 2 appended to form the two names. The names are autogenerated if not specified by the user here. Available only when `ter_interface` is `Tpump`.

Errlimit: ([integer](#), optional) Specifies the value for the first part (number of errors allowed) of the Teradata ERRLIMIT clause. Available only when `ter_interface` is `Tpump`.

Errpercent: ([integer](#), optional) Specifies value for second part (percentage of errors allowed) of the Teradata ERRLIMIT clause. Available only when `ter_interface` is `Tpump`.

ignore_dupl_insert: ([boolean](#), optional) Ignore duplicate rows on insert. Available only when `ter_interface` is `Tpump`.

ignore_dupl_update: ([boolean](#), optional) Ignore duplicate rows on update. Available only when `ter_interface` is `Tpump`.

ignore_missing_delete: ([boolean](#), optional) Ignore missing rows on delete. Available only when `ter_interface` is `Tpump`.

ignore_missing_update: ([boolean](#), optional) Ignore missing rows on update. Available only when `ter_interface` is `Tpump`.

Macrodb: ([string](#), optional) Database to use for Tpump macro creation and usage. Available only when ter_interface is Tpump.

Serialize: ([boolean](#), optional)

True — Specifies to guarantee serializability of operations on the same key.

False — Specifies not to guarantee serializability of operations on the same key.

Available only when ter_interface is Tpump.

Pack: ([integer](#), optional) Specifies the number of statements to pack into a single Tpump request. Available only when ter_interface is Tpump.

Default is 1.

Rate: ([integer](#), optional) Specifies the number of statements to send per minute to Teradata database. Available only when ter_interface is Tpump.

How CONTINUOUS UPDATE TABLE executes SQL statements

Continuous Update Table executes embedded SQL UPDATE, INSERT, or DELETE statements located in the files or embedded statements referenced by the updateSqlFile and (if specified) in insertSqlFile parameters.

The statements are applied to each incoming record as follows:

The statement referenced by updateSqlFile is tried first.

If the statement can be successfully applied to the current record, it is executed; and the statement referenced by insertSqlFile is skipped.

If the updateSqlFile statement cannot be applied to the current record, the statement referenced by insertSqlFile is tried.

There should be only 1 SQL stmt each per updateSqlFile & insertSqlFile. The updateSqlFile stmt need not be UPDATE & the insertSqlFile need not be INSERT; however, u can only use UPDATE, INSERT, or DELETE statements.

The updateSqlFile & insertSqlFile parameters need not be files. U can embed the SQL stmt's in the component (select Embedded on the Parameters properties tab & type the stmt in adjacent text box). An advantage of coding this way is that embedded stmt's can contain references to other graph parameters. The parameters must be specified with the \${} syntax; Ex, \${table_name}.

Applying parallelism to CONTINUOUS UPDATE TABLE

You can apply any level of parallelism to the layout of Continuous Update Table. However, each partition of a Continuous Update Table running in parallel can compete for database locks on the table it references and deadlock may result.

When using commitTable, make sure the update data always goes to the same components in the same order every time the graph is rerun with the same inputs. This means that one of the following must be true:

The data must be partitioned by key and sorted before being sent to Continuous Update Table.

The data must be input directly from a flat file or multifile.

Stored procedures and CONTINUOUS UPDATE TABLE

Instead of using a statement in SQL, you can perform the update operation by calling a stored procedure, specified in the primarySqlIn and/or secondarySqlIn parameters.

Stored procedures are currently supported for the following databases:

Oracle 8i and higher versions (including those residing in packages)

DB2 Universal Database 6.x and higher (Unix and Windows)

Microsoft SQL Server (OLEDB or ODBC interface)

Sybase

Note the following:

Stored procedures with input and/or output parameters are not supported.

If :a is an input parameter, you can repeat the parameter in the call. For example:

call ab_proc_select(:a, :a, :a)

However, output parameters cannot be repeated in this way. Each output parameter name can appear only once in an invocation.

Syntax for Oracle, DB2, and Microsoft SQL Server

The syntax for calling a stored procedure using Oracle, DB2, or Microsoft SQL Server (OLEDB or ODBC interface) is as follows:

{call | exec | execute} [schema.][package.]stored_procedure(:a, :b, ...)

where:

:a and :b are input arguments.

Note that return codes (:a in the above syntax) from DB2 are not supported by the component.

a and b are parameters bound to DML variables. The DML variable names are a and b, and they are referenced by prefixing their names with a colon ("bind variable" syntax).

schema specifies the schema where the stored procedure is defined.

package specifies the package where the stored procedure is defined (Oracle only).

You cannot specify literals (such as NULL, 1, or '2005-05-15 00:00.000') in stored procedure arguments when using Oracle, DB2, or Microsoft SQL Server using the ODBC interface. However, you can specify literals as arguments to stored procedures when using Microsoft SQL Server with OLEDB.

You can call Oracle stored procedures through synonyms. You can also overload Oracle procedure names, as long as the different procedures have argument lists of different size (that is, with different numbers of arguments). However, if the same-named procedures differ only in the data types of their arguments and not in their number, the Co>Op will not necessarily correctly distinguish between them.

Sybase syntax:

The syntax for calling a stored procedure using Sybase is as follows:

{exec | execute} [schema.][owner.]stored_procedure [parm [, parm, ...]]

where parm can be any of the following:

A literal value input argument

An input parameter referencing a DML variable

The DML variable is referenced by prefixing its name with a colon; for example, :my_var.

An output parameter referencing a DML variable

The DML variable is referenced by prefixing its name with a colon. The variable name must also be followed by the output keyword; for example, :my_out_var output.

The order of the arguments is determined by the stored procedure definition.

A Sybase stored procedure used with a CONTINUOUS Update Table can return either one result set, or output parameters, though not both. It can also return nothing. However, neither result set nor output parameters are returned to the CONTINUOUS Update Table.

Note that you can specify literals as arguments to stored procedures when using Sybase.

Transaction handling

Stored procedures should not issue transactional commands (such as commit statements). You should leave transactionality to be handled by CONTINUOUS Update Table.

Error handling

Stored procedures are free to raise or return errors to the calling component. The component handles any such errors the same way it would errors from SQL statements.

Ex, u can connect the reject and error ports on and obtain per-record error information for stored procedures, in exactly the same way as for SQL statements.

Example

For examples of using stored procedures to do updates, see the ["Stored procedure example"](#).

Specifying SQL statements for CONTINUOUS UPDATE TABLE

Record format field names referenced in the SQL must be prefixed with a colon ("bind variable" notation).

Parameter substitution

If u write an SQL stmt as an embedded value for a parameter, and you want to reference another parameter from the stmt, u must use \${} substitution, not \$ substitution. If u do not use the curly braces, the dollar sign (\$) is taken literally and no substitution occurs. To use \${} substitution, click More on the Parameters properties tab and select \${} substitution from the Interpretation drop-down list.

Updating or inserting Oracle date fields

If u want to use a bind variable value as a date or datetime value, we recommend that you declare the field as date or datetime in the DML: this automatically takes care of any special formatting.

Bind variables referencing fields declared otherwise than as date or datetime require the conversion technique shown below in order to be put into the appropriate format.

In UPDATE or INSERT stmts for date fields, use the following syntax to specify the record format (unless fieldname was already declared as date or datetime in the DML):

```
TO_DATE (:fieldname, 'format')
```

For example, if your input data for the date1 field is in YYYY-MM-DD format, you would write the following in the UPDATE or INSERT statement:

```
TO_DATE (:date1, 'YYYY-MM-DD')
```

SQL example: Suppose the input flow (connected to the in port) to Continuous Update Table has the following record format:

```
record
    string(5) name;
    decimal(5) age;
end
```

Suppose also that the database table to be updated contains the following columns:

```
emp_name CHAR(5),
emp_age NUMBER
```

The updateSqlFile statement might look like this:

```
update employees set emp_name = :name where emp_age = :age
```

The insertSqlFile statement might look like this:

```
insert into employees values (:name, :age)
```

Problem with connecting to host

Problem: The GDE could not connect to the server specified in the Host Connection Settings associated with the graph being packaged. U must resolve this in order to continue.

Solution: Resolve the login problem using the Connections dialog:

1. In the GDE, select Settings > Connections.
2. Select an entry and click Edit to modify the settings.
3. When your changes are complete, click Test Connection.
4. Once the login succeeds, you can repackage the graph.

What does the error message "NULL value in assignment for left-hand side" mean?

Short Ans: This error msg occurs most commonly when u try to assign a value to a vector but u have not declared any storage for the variable.

Details: Consider the following vector variable declaration:

```
let record
    string("\307") cust_name;
end[in.num_cust_recs] cust_rec;
```

This declaration defines a type for variable, but does not allocate any storage for it. If you then try to assign a value to the vector, this error msg is displayed when u execute the graph.

To assign a value to a vector element, you must first explicitly allocate storage for the vector. In Version 2.12.2 of the Co>Op, u can easily do this using the [allocate](#) function.

For example:

```
let record
string("\307") cust_name;
end[in.num_cust_recs] cust_rec = allocate();
```

U can allocate storage & initialize vector elements at the same time using [make_constant_vector](#) function:

```
let record
    string("\307") cust_name;
end[in.num_cust_recs] cust_rec =
    make_constant_vector(in.num_cust_recs,[record cust_name ""]);
```

What does the error message "Remote job failed to start up" mean?

Short Ans: Every time u run a graph, communication takes place bet'n various parts of the Ab Initio software. Incorrect setup of this communication can lead to this error.

In some cases, it might not be apparent that there is more than 1 machine involved in execution of the graph. This situation typically arises if some of the components in graph r configured to run on a remote machine. If the communication bet'n the machine with Co>Op specified in the [Connections](#) dialog & remote machine is not set up properly, u could see this error message.

U can also get this error if u r using an EME datastore or a database on a remote machine.

To troubleshoot this error:

1. Determine which machines are involved in the execution of the graph.

2. Make sure the Co>Op is installed on each machine. (This step is not required if the machine only has a database on it that you plan to access using a database client.)
3. Set up the communications between these machines using the appropriate configuration files.

Details: This error typically occurs when the Co>Op on one machine cannot communicate with the Co>Op on a remote machine. This can occur in the following situations:

Graph running across multiple machines — Some graphs have some components executing on one machine and other components executing on other machines. Where a particular component executes is determined by the [layout](#) of that component. The layout of a particular component is usually specified in terms of a filesystem on a machine. To determine whether a graph runs across multiple machines, check the layouts of the components & see if any of the layouts specify filesystems located on remote machines.

Co>Op/EME communications — If the graph accesses an EME datastore on a remote machine, u need to set up communication bet'n the Co>Op & that EME. One way to test the connection bet'n the run host & the EME host is to run the command `m_ls full_path_to_a_directory_on_EME_host`. If this command fails, you must set up the communication between the run host and the EME host.

NOTE: The fact that u can connect to the EME from the GDE by clicking the Connect button in the EME Datastore Settings dialog does not imply that the connection between the Co>Op & the EME is operational.

Database on a different machine — This is yet another situation where u might have multiple Co>Op's communicating with each other. The layout of database components will specify which machine they r running on. If layout is specified as Database:default, the component is executing on the same machine as the database itself.

To troubleshoot this error:

2. Determine which machines are involved in the execution of the graph.
3. Make sure the Co>Op is installed on each machine.
4. Test the connections between pairs of machines.

To determine whether connection bet'n a pair of machines is set up properly, run the command `m_ls` on one machine with an argument that points to a directory on the remote machine.

If `m_ls` command fails, fix the connection bet'n the machines by using one of the following files:

The `.abinitiorc` file that resides in the user's top-level directory on the run host

A system-wide `abinitiorc` file

Typically, these configuration files contain information such as:

The name of the remote machine

The user name and password to be used when connecting to the remote machine

The location of the Co>Op on the remote machine (\$AB_HOME)

The connection method used; make sure the connection method you specify is a valid connection method on your system.

A sample of the kind of information that can be included in these files can be found in the file abinitiorc.example, located in \$AB_HOME/config.

4. Once u have set up the .abinitiorc file, test the connection using the m_Is command.

Database specifics:

If ur db is on a remote machine that does not have a Co>Op installed, u must connect to the db using a local db client (specific to the db u r using). In this case, the [.dbc](#) file needs to be set up to use the db client rather than attempting to connect directly. If this is not done, u might get this error — bcoz the component will try to connect directly to remote machine, expecting the Co>Op to be installed in the same place as on the run host.

SIGPIPE errors

Often the job output will indicate a SIGPIPE error:

```
===== Error from GenerateData.000 on rs1 =====
```

Component terminated by signal SIGPIPE

This is probably due to a downstream process exiting prematurely

```
===== Error from Sort.000 on rs1 =====
```

Trouble writing sort temp file /small/lukas/d1/.WORK/ccd7a743-381d1f49-32a8/Sort.000.013: No space left on device

The SIGPIPE error is typically not the actual error — the SIGPIPE error is a by-product of pipelining. The actual error message comes from a component downstream from the SIGPIPE, as indicated in the lines of the message following the line containing SIGPIPE.

Why am I getting SQL parsing errors during dependency analysis?

Ans: The SQL parser for dependency analysis currently has some limitations. In particular, it is unable to handle database-specific extensions to the ANSI SQL standard, nested SELECTs, and unions. Improvements to the SQL parser are planned for future releases of the product.

Details: To work around these limitations, use a graph/sandbox parameter for the problematic SQL stmts; & add a `_REPO` parameter to replace the runtime value with a value that has the same dependencies and that the SQL parser can parse. The `_REPO` mechanism enables you to remap a standard parameter to an EME datastore-specific value.

Why am I getting SQL analysis errors when I try to check my project into the EME?

Short answer

There might be differences between the table format and the DML on some component. Try regenerating the DML before checking the project into the EME.

Details

There are many kinds of SQL analysis errors; some common ones are that a field cannot be found in a given record format or table, or that there is no record format for the table. The EME relies on the GDE to provide information about the structure of the database tables referenced in the graph. For the EME to create a dataset object for an Input Table component and use it in dependency analysis, you must associate the component in the GDE with the correct table and schema, and have the GDE generate the appropriate record format.

When you have a database component automatically generate its record format, the GDE queries the database for the structure of all tables referenced. This information is not used by the graph in execution but is stored in the .mp file for dependency analysis. The EME is not only concerned with the record format of the port (for the data coming out of the Input Table component), but also with the record formats of all relevant tables (for the data conceptually coming into the component). This is necessary to do dependency analysis through the database, as an earlier graph might write these tables.

If you change which table or schema the component references (for example, if you use a configuration variable for the table or schema and you change its value), you must regenerate the record format before checking your project into the EME datastore. Otherwise, the graph's information about the table structure will still correspond to the original table — even if the record format might have been changed to reflect the current table. For this reason it is important not to edit the generated record format by adding or removing fields.

If these errors occur at checkin, that means the errors are not a direct result of the checkin but rather of dependency analysis, which is often performed along with checkin. You can check projects into the EME without doing dependency analysis as part of the checkin process (that is, using translation only), which will improve your check-in performance.

NOTE: In Version 2.14, dependency analysis is not part of the checkin process.

