Ab Initio

Cook>Book

September 19, 2006 > Part Number AB0595
Ab Initio Software Corporation
201 Spring Street > Lexington MA 02421 > Voice 781.301.2000 > Fax 781.301.2001 > support@abinitio.com

>®	Continuous Flows®	Enterprise MetaEnvironment	Plan>It
Ab Initio®	Continuous>Flows®	Everything 2 Everything®	Re>Posit®
Ab Initio I>O®	Cooperating Enterprise®	Everything to Everything®	Re>Source®
Abinitio.com®	Cooperating System®	From The Beginning®	Server++®
Applications drawn to scale®	Cooperating®	GDE	Server+Server®
Co>Operating Enterprise®	Data>Profiler®	I>O®	Shop for Data®
Co>Operating System®	Dynamic Data Mart®	If You Can Think It, You Can Do It®	The Company Operating System®
Co>Operating®	EME®	init.com	Think, Draw, Run, Scale, Succeed®
Co>Operation®	EME Portal	INIT®	
Co>Operative®	Enterprise Meta>Environment®	Meta Operating System	
Co>OpSys®	Enterprise Metadata Environment	Meta OS	

Contents

Getting started

The *Cook>Book* presents a series of graphs that illustrate how you can use Ab Initio software to solve a wide variety of problems. The solutions range from the very simple, where the emphasis is on the mechanics of using the GDE™ (Graphical Development Environment), to the fairly complex, where the algorithms are described in depth.

You can use the *Cook>Book* in two ways:

- As a source of "recipes" that you can either implement as is, modify slightly, or re-create to solve a specific problem
- As a toolbox of running examples illustrating numerous features, tips, and techniques that you can apply when developing your own Ab Initio graphs

Each chapter examines a different problem, and many of the chapters describe a series of graphs that illustrate different approaches to solving it. The chapters are arranged roughly in order of increasing complexity: the earlier ones provide detailed, step-by-step instructions ("recipes") for re-creating the graphs presented. Later chapters build on those, either by fully incorporating a solution for a subproblem or by reusing a technique previously shown. The final chapters assume that you have mastered the mechanics, and present "guided tours" of complete graphs, focusing on highlights directly relevant to solving the problem at hand.

Audience and prerequisites

The *Cook>Book* is for those who have completed *The Tutorial*, the *Introductory Course*, and *Work>Book 1*, and are eager to broaden and deepen their Ab Initio software development skills.

Listings of *Cook>Book* graphs by component and feature

For examples of specific components or features used in running graphs in the *Cook>Book*, see the following sections:

- Graphs listed by component (page 213)
- Graphs listed by feature (page 218)

Setting up

To set up your *Cook>Book* environment:

1. Open a new instance of the GDE (to ensure that you will be running the setup script in the host directory specified in the **Run Settings** dialog).
2. From the GDE menu bar, choose **Run > Execute Command** (or press F8).
3. In the **Execute Command** dialog, type the following and click **OK**:

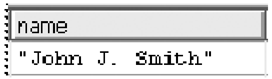
```
$AB_HOME/examples/setup/set-up-cookbook.ksh
```

The setup script creates (in the current working directory) a directory named **cookbook** that contains as subdirectories the various sandboxes referenced in this book. The script also builds a multifile system (in **AI_MFS**) and data directories (in **AI_SERIAL**) for each sandbox, and runs graphs to set up datasets.

The script can take anywhere from a few seconds to a few minutes, and displays **Done** when finished.

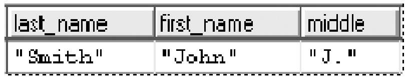
Parsing names

String manipulation is one of the most common kinds of data transformation and is part of almost every Ab Initio graph. The recipes in this chapter describe how to create a simple graph that uses DML string functions and local variables to parse a name into separate strings for first name, middle name or initial, and last name. For example, it reformats this name field:



name
"John J. Smith"

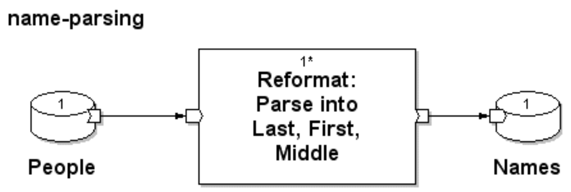
to this:



last_name	first_name	middle
"Smith"	"John"	"J."

Graph for parsing names

The graph **name-parsing.mp** (in the **appetizers** sandbox) uses a REFORMAT component to parse the input file:



The input file stores the first name, middle initial, and last name in the **name** field of the following record format (in **people.dml**):

```
record
  decimal(',') id;
  string(',') name;
  decimal('\n') age;
end
```

Creating the graph

To create a graph like name-parsing.mp:

1. Add an INPUT FILE with a record format such as the one shown above, which stores an entire name in a single field, and an input dataset such as **people.dat**.

2. Add an OUTPUT FILE, specify a name for it, and embed the following record format:

```
record
  decimal(',') id;
  string(',') last_name;
  string(',') first_name;
  string('\n') middle;
  decimal('\n') age;
end
```

3. Add a REFORMAT component between the input and output files, and connect them to it.

4. In the REFORMAT component, use the **Transform Editor**'s **Grid View** to create the local variables listed in the following table, as described in the instructions after the table:

VARIABLE NAME	TYPE	LENGTH	EXPLANATION AND VALUE
tname	string	""	Holds **in.name** with leading and trailing blank characters removed: `string_lrtrim(in.name)`
end_of_first	integer	4	Represents the location of the first blank character in **tname**: `string_index(tname, ' ')`
beginning_ of_last	integer	4	Represents the location of the last blank character in **tname**: `string_rindex(tname, ' ')`

a. Hold down the Shift key and double-click the REFORMAT component to display the **Transform Editor**.

NOTE: If the **Transform Editor** opens in **Text View**, choose **View > Grid View**.

b. Switch to the **Variables** tab and choose **Edit > Local Variable** to display the **Variables Editor**.

c. On the **Fields** tab, enter the name, type, and length attributes shown above, and save the variable.

d. On the **Variables** tab of the **Transform Editor**, type the following into the first blank line, or drag the **name** field from the **Inputs** pane into the first blank line and type the rest of the code around it:

```
string_lrtrim(in.name)
```

e. Connect the rule port to the variable **tname** in the **Outputs** pane.

If you view the transform in **Text View**, it should now contain the following line of code:

```
let string("") tname =string_lrtrim(in.name);
```

f. Create the other two variables in the same way, each time using the next blank line of the **Variables** tab.

As you do this, notice that the **Variables** section of the **Inputs** pane incrementally shows the previously created variable(s), indicating that they are available for use in creating additional variables.

If you view the transform in **Text View** after creating the three variables, you should see the following code:

```
out::reformat(in) =
begin
  let string("") tname =string_lrtrim(in.name) ;
  let integer(4) end_of_first =string_index(tname, ' ');
  let integer(4) beginning_of_last =string_rindex(tname, ' ');
end;
```

5. Create three business rules to construct the output fields (listed in the following table) for the three parts of the name. You can do this in the **Transform Editor**'s **Grid View**, as described in the instructions after the table.

OUTPUT FIELD	DESCRIPTION	BUSINESS RULE
out.last_name	Extends from the character after the last blank to the end of the string	`string_substring(tname, beginning_of_last + 1, length_of(tname))`
out.first_name	Extends from the start of **name** to the last character before the first blank	`string_substring(tname, 1, end_of_first - 1)`
out.middle	Extends from the first character after the first blank to the character before the last blank	`string_substring(tname, end_of_first + 1, beginning_of_last - end_of_first - 1)`

   a. Position the cursor in the first blank line on the **Business Rules** tab.

   b. Choose **Edit > Rule** to display the **Expression Editor**.

   c. In the **Functions** pane, expand the **String Functions** item and double-click **string_substring(str, start, length)** to display an empty function definition (**string_substring(?, ?, ?)**) in the bottom pane.

   d. Replace the question marks with the following parameters, either by typing them directly or by dragging the variable names from the **Fields** pane and typing as needed:

```
string_substring(tname, beginning_of_last + 1,
    length_of(tname))
```

   e. Click **OK** to return to the **Transform Editor**.

   f. Drag the rule port to the **last_name** field in the **Outputs** pane.

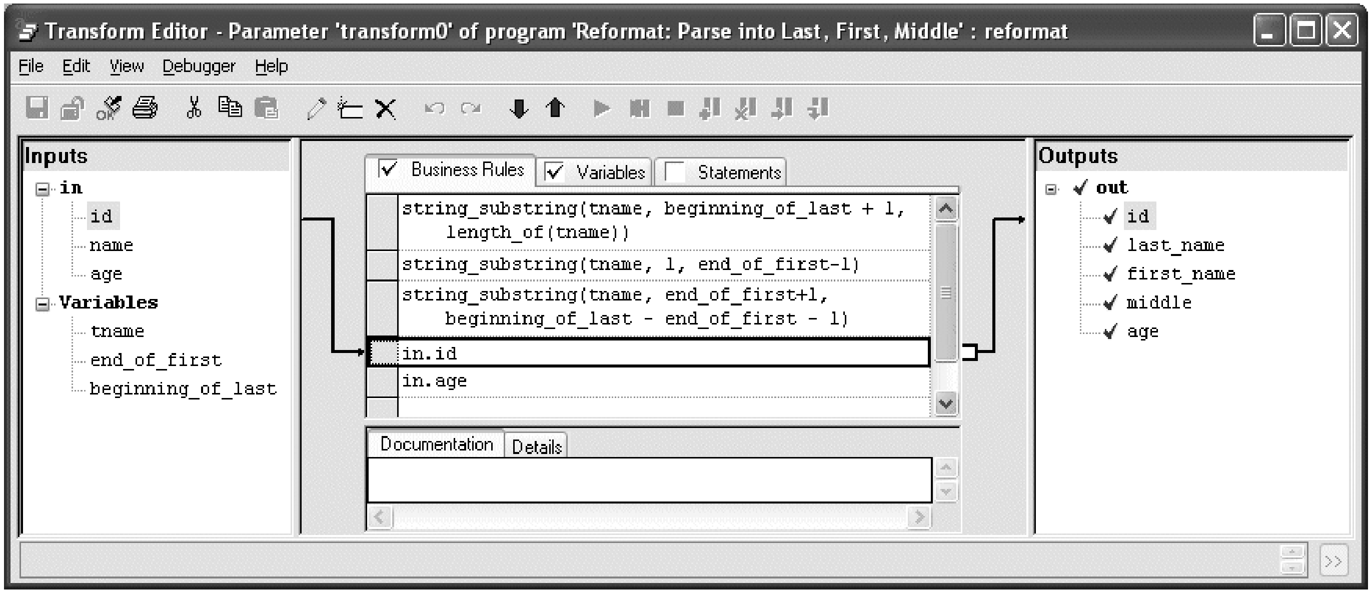
g. Create two more business rules in the same way to construct the output fields for **first_name** and **middle**.

If you view the transform in **Text View** after creating the three variables, you should see the following additional code:

```
out.last_name :: string_substring(tname,
    beginning_of_last + 1, length_of(tname));
out.first_name :: string_substring(tname, 1,
    end_of_first - 1);
out.middle :: string_substring(tname, end_of_first + 1,
    beginning_of_last - end_of_first - 1);
```

NOTE: In the statement that computes **last_name**, the call to **string_substring** uses the length of the entire string — **length_of(tname)** — for the length of the substring. This has the effect of simply taking all the characters starting at **beginning_of_last + 1**. While the length calculation could have been refined (to **length_of(tname) – beginning_of_last**), the result would be the same, as **string_substring** simply returns the available characters.

6. Assign the **id** field and the **age** field from the inputs to the outputs: in the **Transform Editor**'s **Grid View**, drag each field from the **Inputs** pane over the first empty line at the bottom of the **Business Rules** tab and onto the field with the corresponding name in the **Outputs** pane.



```
string_substring(tname, beginning_of_last + 1,
    length_of(tname))
string_substring(tname, 1, end_of_first-1)
string_substring(tname, end_of_first+1,
    beginning_of_last - end_of_first - 1)
in.id
in.age
```

7. Run the graph and view the results.

Generating large amounts of data

To test performance, you should run every graph on a volume of records that is comparable to what you expect in production. But when developing and testing a graph, you may have only a small sample of real data, or none at all. The recipes in this chapter describe how to create graphs that generate large amounts of test data efficiently.

The **multiplying-data** sandbox contains graphs that use a NORMALIZE component to generate large amounts of data by doing the following:

- Multiplying records by a fixed number (page 14)
- Multiplying records by a parameterized number (page 15)
- Multiplying records by a parameterized number in a reusable subgraph (page 16)

Two additional graphs demonstrate how to generate data by using other components:

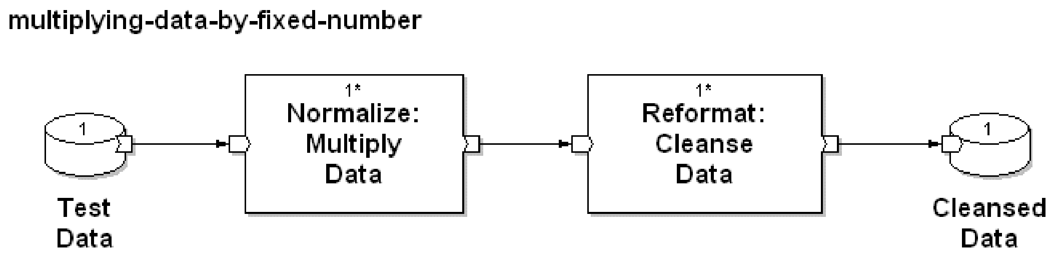
- Departition components: CONCATENATE, GATHER, and INTERLEAVE (page 18)
- Three-way self-join on an empty key with a FILTER BY EXPRESSION component (page 19)

GENERATE RECORDS COMPONENT

The GENERATE RECORDS component is good way to generate random records, but if you need very large volumes of data, you should use it with one of the techniques described here to get best performance.

Simple data multiplier graph

The graph **multiplying-data-by-fixed-number.mp** multiplies the four records in the input file 1,000,000 times, producing 4,000,000 records:



To create a graph like multiplying-data-by-fixed-number.mp:

1. Attach a NORMALIZE component to your input file.
2. Hold down the Shift key and double-click the NORMALIZE component to generate a default transform.
3. Delete the **temporary_type** variable and all default functions except **length** and **normalize**.
4. Code the **length** function (which determines the number by which to multiply the input records) as follows, assigning the multiplier to the **out** record:

```
out :: length(in) =
begin
  out :: 1000000;
end;
```

5. Code the **normalize** function to simply assign the **in** records to each **out** record:

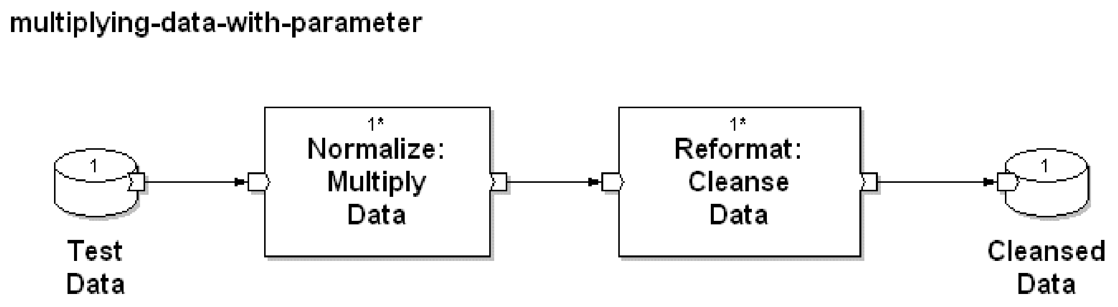
```
out :: normalize(in, index) =
begin
  out :: in;
end;
```

6. Send the generated data to the component you need to test.

> **NOTE:** The REFORMAT component in the graph is not explicitly described here, as it is merely a placeholder representing any component or set of components that need to be tested with a large volume of data.

Parameterized data multiplier graph

The graph **multiplying-data-with-parameter.mp** multiplies the records in the input file by a number specified in a graph parameter. The parameter is **multiplier** and its value is **1000000**.



To create a graph like multiplying-data-with-parameter.mp:

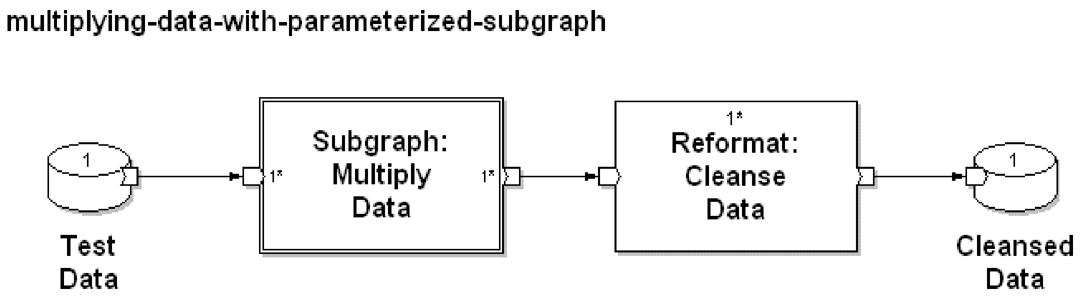
1. Create a simple data multiplier graph as described on page 14.
2. Add a graph parameter (such as **multiplier**).
3. Set the interpretation of the NORMALIZE component's **transform** parameter to **${} substitution**.

4. In the **length** function of the NORMALIZE component's transform, replace the hardcoded value with the parameter as follows:

```
out :: length(in) =
begin
  out :: ${multiplier};
end;
```

Reusable, parameterized data multiplier subgraph

The graph **multiplying-data-with-parameterized-subgraph.mp** contains a reusable, parameterized subgraph. (The parameter is **multiplier** and its value is **1000000**.)



To create a graph that contains a reusable subgraph (like multiplying-data-with-parameterized-subgraph.mp):

1. Create a parameterized data multiplier graph as described on page 15.
2. Create a subgraph from the NORMALIZE component by right-clicking the component and choosing **Subgraph > Create From Selection**.

3. Delete the **multiplier** parameter from the graph level as follows:
    a. Choose **Edit > Parameters**.
    b. In the **Edit Parameters** dialog, select the **multiplier** parameter in the **Name** field, and choose **Edit > Delete**.
4. Create a **multiplier** parameter at the subgraph level:
    a. Select the subgraph and choose **Edit > Parameters**.
    b. In the **Name** field of the **Edit Parameters** dialog, type **multiplier**.
    c. In the **Value** field of the **Edit Parameters** dialog, type the value you want.
    d. Save the changes to the parameters.

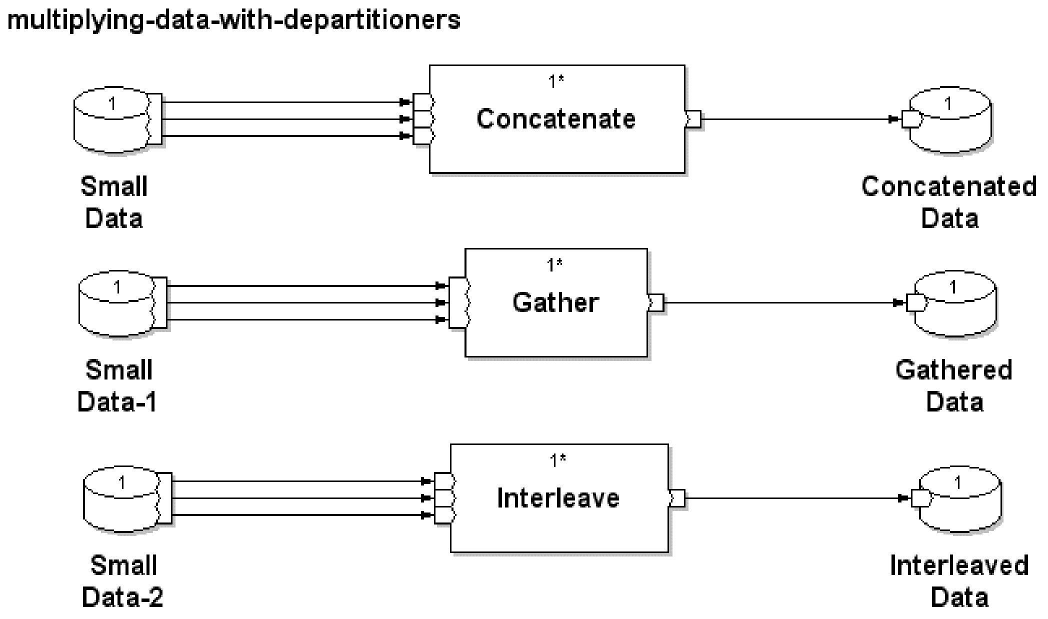
Other ways to multiply data

A NORMALIZE-based data multiplier is used in the preceding recipes because it is very efficient and completely generic. However, you can also create large amounts of data quickly by using a departitioner or a self-join on an empty key, as described in the following sections:

- Departitioners (next)
- Self-join with FILTER BY EXPRESSION to control output size (page 19)

Departitioners

You can multiply data by using a departition component, as shown in the graph **multiplying-data-with-departitioners.mp**:

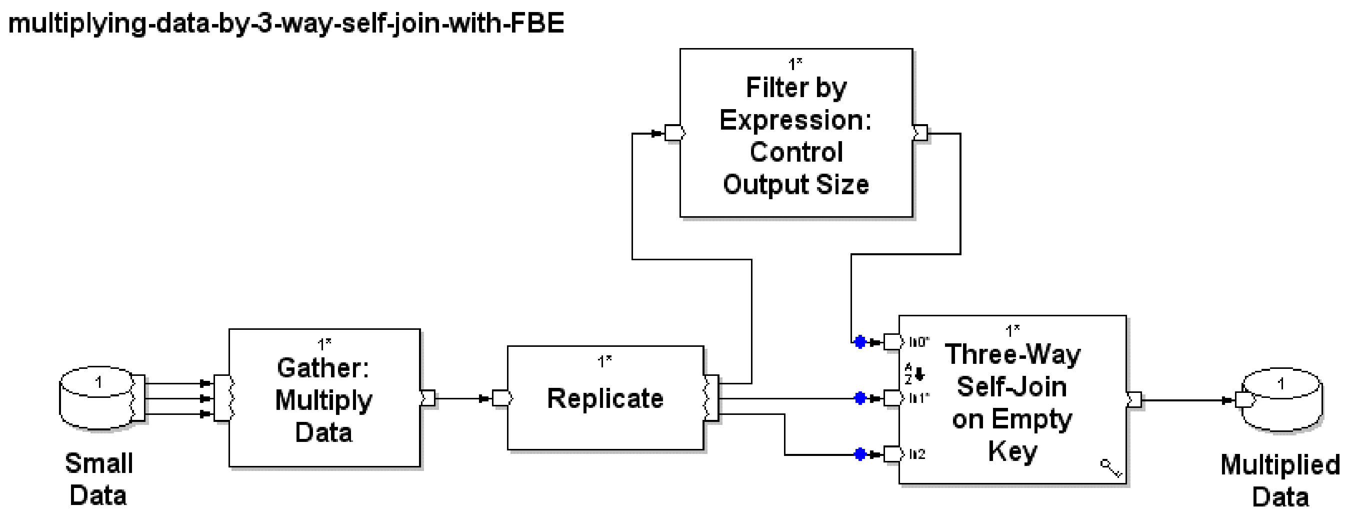


To generate *M* copies of the input data:

- Connect an input file to one of the following components with *M* flows:
  - CONCATENATE — To get *M* copies of the whole file, one after another
  - GATHER — To get *M* copies of each record, in some arbitrary order
  - INTERLEAVE — To get *M* copies of each record in succession

Self-join with FILTER BY EXPRESSION to control output size

With a two-way self-join on an empty key, you can generate *N*N* output records from *N* input records. With a three-way self-join, you can generate *N*N*N* output records. You can further control the output size by sending one of the inputs through a FILTER BY EXPRESSION before it enters the JOIN. As you can see in the graph **multiplying-data-by-3-way-self-join-with-FBE.mp**, several techniques can be applied in tandem to multiply data:


multiplying-data-by-3-way-self-join-with-FBE
Filter by Expression: Control Output Size
Small Data
Gather: Multiply Data
Replicate
Three-Way Self-Join on Empty Key
in0
in1
in2
Multiplied Data

To create a graph like multiplying-data-by-3-way-self-join-with-FBE.mp:

1. Connect multiple flows of your input file to a GATHER component.
2. Replicate the output of the GATHER.
3. Send one flow from the REPLICATE to a FILTER BY EXPRESSION component, and configure the FILTER BY EXPRESSION as follows:
   a. Hold down the Shift key and double-click the FILTER BY EXPRESSION to display the **Expression Editor**.

b. In the **Functions** pane, double-click **Miscellaneous Functions** to expand the list.

c. Double-click **random(int) returns long** to display **random(?)** in the bottom pane of the **Expression Editor**.

d. To control the percentage of the records entering the FILTER BY EXPRESSION that will be passed to its **out** port, edit the expression to something like this (see the sidebar for details about the **random** function):

```
random(100) < N
```

where *N* is the percentage of records (in this graph, *N* is **50**).

4. Connect the **out** port of the FILTER BY EXPRESSION and the remaining flows from the REPLICATE to a JOIN, and configure the JOIN as follows:

a. Set the key to the empty key: **{ }**

b. Set the **transform** parameter in the **Transform Editor**'s **Grid View** by dragging **in0** from the **Inputs** pane over the first empty line in the **Business Rules** tab and onto **out0** in the **Outputs** pane.

If you view the transform in **Text Mode** after doing this for a three-way join, you should see the following code:

```
out::join(in0, in1, in2) =
begin
  out :: in0;
end;
```

5. Connect the JOIN to an OUTPUT FILE.

USING THE random FUNCTION IN A FILTER BY EXPRESSION TO CONTROL OUTPUT SIZE

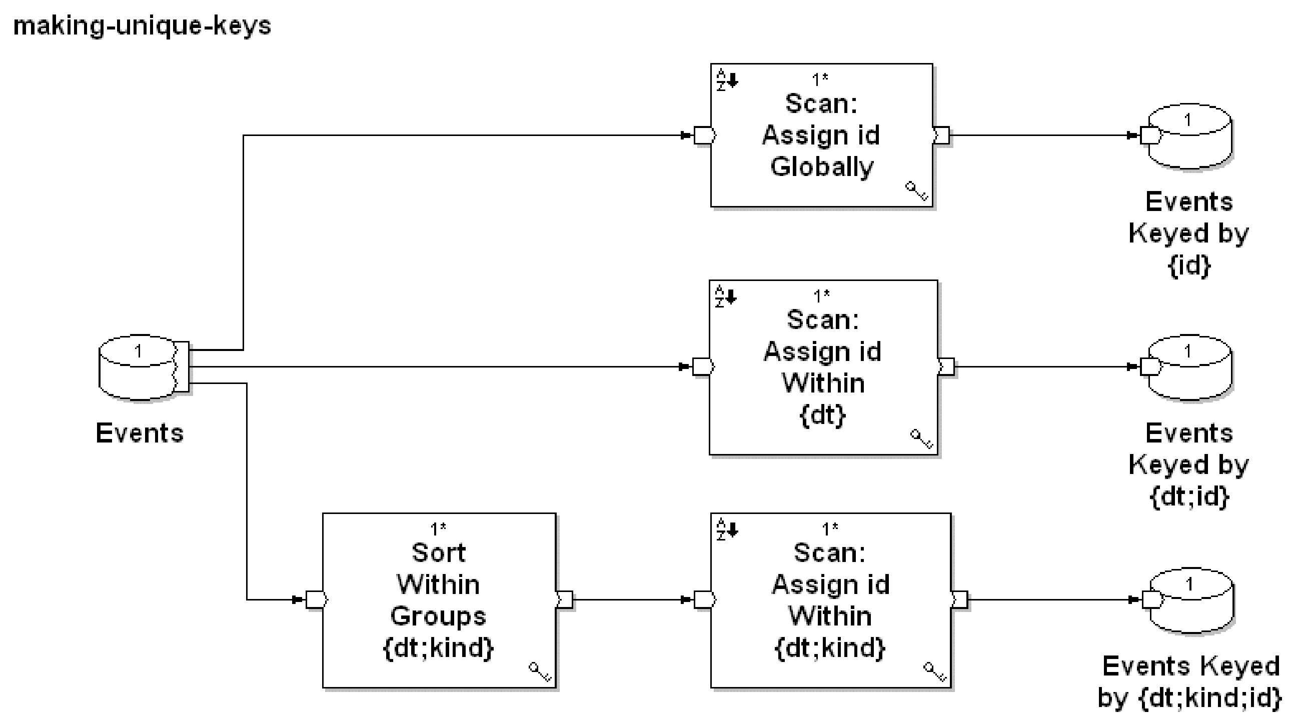
For each incoming record, the **random** function generates a random integer in the range from **0** to the value passed to it. By passing **100** to the **random** function and then testing whether the generated number is less than a specified value (in an expression such as **random(100) < N**), you cause the FILTER BY EXPRESSION to send approximately *N* percent of the incoming records to its **out** port.

Making unique keys

In some situations you may need to assign unique keys to data that is not uniquely keyed to begin with. The recipes in this chapter describe how to use the SCAN component to generate unique keys globally or within an existing key grouping.

Graph for making unique keys

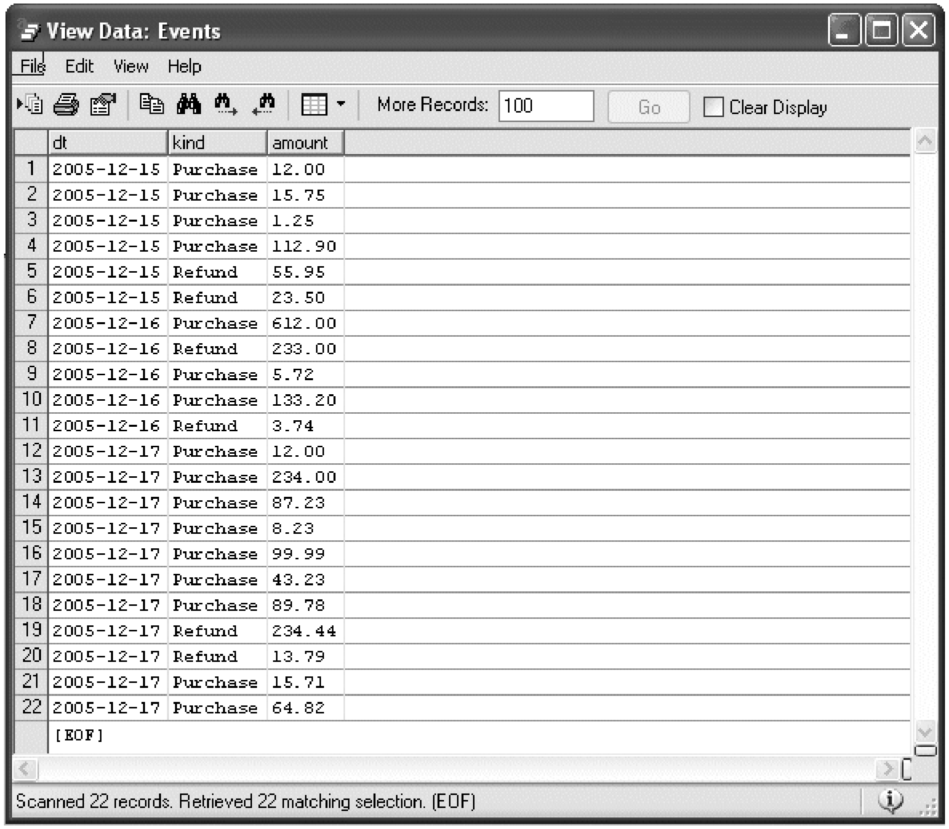
The graph **making-unique-keys.mp** (in the **appetizers** sandbox) contains three SCAN components that use the same transform code and the same output record format. But because each SCAN uses a different key, the scope of uniqueness is different for each output file.


making-unique-keys
Scan: Assign id Globally
Events Keyed by {id}
Scan: Assign id Within {dt}
Events
Events Keyed by {dt;id}
Sort Within Groups {dt;kind}
Scan: Assign id Within {dt;kind}
Events Keyed by {dt;kind;id}

The input record format for the graph is specified by **events.dml**:

```
record
  date("YYYY-MM-DD")(",") dt;
  string(",") kind;
  decimal("\n") amount;
end
```

The input file contains the following data, which is sorted on the date field (**dt**):


View Data: Events

	dt	kind	amount
1	2005-12-15	Purchase	12.00
2	2005-12-15	Purchase	15.75
3	2005-12-15	Purchase	1.25
4	2005-12-15	Purchase	112.90
5	2005-12-15	Refund	55.95
6	2005-12-15	Refund	23.50
7	2005-12-16	Purchase	612.00
8	2005-12-16	Refund	233.00
9	2005-12-16	Purchase	5.72
10	2005-12-16	Purchase	133.20
11	2005-12-16	Refund	3.74
12	2005-12-17	Purchase	12.00
13	2005-12-17	Purchase	234.00
14	2005-12-17	Purchase	87.23
15	2005-12-17	Purchase	8.23
16	2005-12-17	Purchase	99.99
17	2005-12-17	Purchase	43.23
18	2005-12-17	Purchase	89.78
19	2005-12-17	Refund	234.44
20	2005-12-17	Refund	13.79
21	2005-12-17	Purchase	15.71
22	2005-12-17	Purchase	64.82
	[EOF]		

Scanned 22 records. Retrieved 22 matching selection. (EOF)

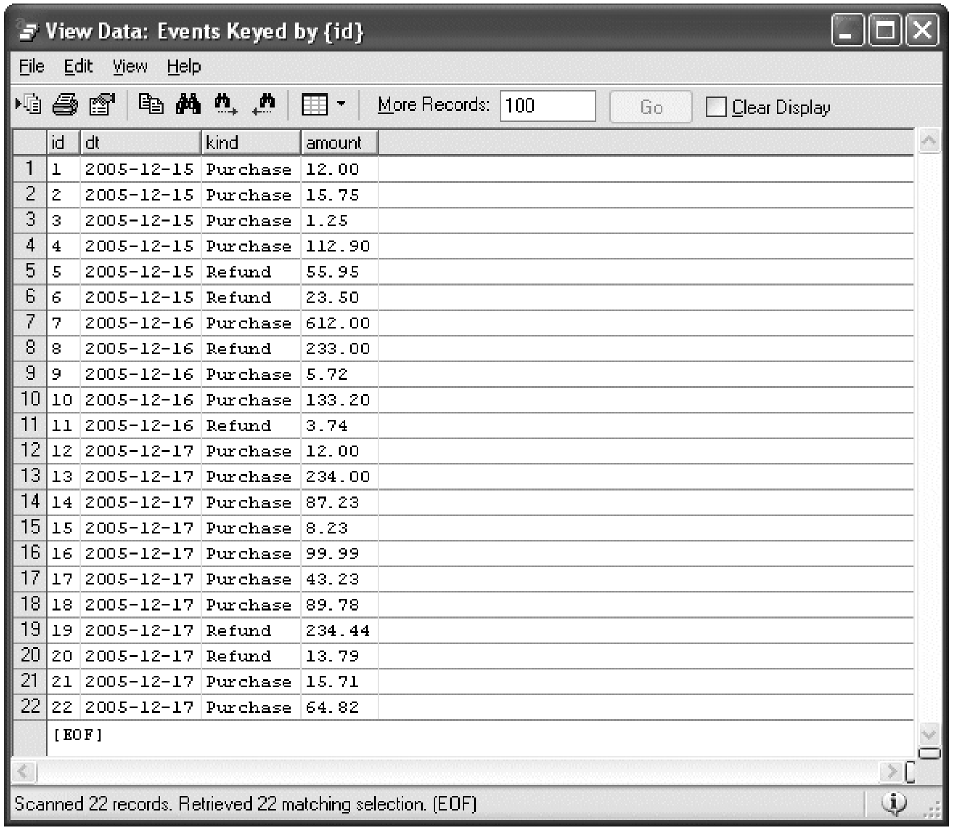
The output record format contains one additional field, **id**, and is specified by **events-keyed.dml**:

```
record
  decimal(",") id;
  date("YYYY-MM-DD")(",") dt;
  string(",") kind;
  decimal("\n") amount;
end
```

Each SCAN generates a **count** value and assigns it to the **id** field in the output file, but each does this in a different way, as described in the following sections.

Globally unique ID

The **key** parameter of the **Scan: Assign id Globally** component is set to the empty key: **{ }**, causing the entire input file to be treated as a single key group. The SCAN simply counts the records and assigns the current **count** to the **id** field for each record, so that every record is uniquely identified across the entire output file. The output data looks like this:



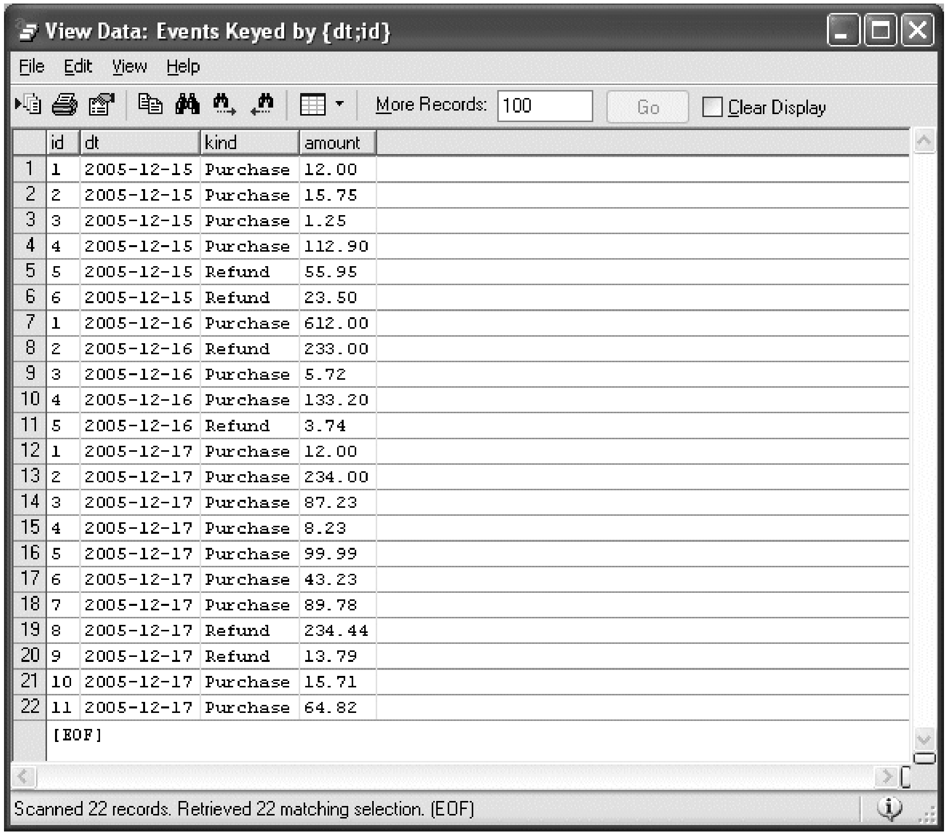
View Data: Events Keyed by {id}

	id	dt	kind	amount
1	1	2005-12-15	Purchase	12.00
2	2	2005-12-15	Purchase	15.75
3	3	2005-12-15	Purchase	1.25
4	4	2005-12-15	Purchase	112.90
5	5	2005-12-15	Refund	55.95
6	6	2005-12-15	Refund	23.50
7	7	2005-12-16	Purchase	612.00
8	8	2005-12-16	Refund	233.00
9	9	2005-12-16	Purchase	5.72
10	10	2005-12-16	Purchase	133.20
11	11	2005-12-16	Refund	3.74
12	12	2005-12-17	Purchase	12.00
13	13	2005-12-17	Purchase	234.00
14	14	2005-12-17	Purchase	87.23
15	15	2005-12-17	Purchase	8.23
16	16	2005-12-17	Purchase	99.99
17	17	2005-12-17	Purchase	43.23
18	18	2005-12-17	Purchase	89.78
19	19	2005-12-17	Refund	234.44
20	20	2005-12-17	Refund	13.79
21	21	2005-12-17	Purchase	15.71
22	22	2005-12-17	Purchase	64.82
	[EOF]			

Scanned 22 records. Retrieved 22 matching selection. (EOF)

Unique ID within a single-field key group

The **key** parameter of the **Scan: Assign id Within {dt}** component is set to the date field, **{dt}**, causing each record with the same **dt** value to be treated as part of the same key group. The SCAN counts the records in each key group and assigns the current **count** within that key group to the **id** field for each record. In the output, the combination of the **dt** field and the **id** field uniquely identifies each record. The output data looks like this:



	id	dt	kind	amount
1	1	2005-12-15	Purchase	12.00
2	2	2005-12-15	Purchase	15.75
3	3	2005-12-15	Purchase	1.25
4	4	2005-12-15	Purchase	112.90
5	5	2005-12-15	Refund	55.95
6	6	2005-12-15	Refund	23.50
7	1	2005-12-16	Purchase	612.00
8	2	2005-12-16	Refund	233.00
9	3	2005-12-16	Purchase	5.72
10	4	2005-12-16	Purchase	133.20
11	5	2005-12-16	Refund	3.74
12	1	2005-12-17	Purchase	12.00
13	2	2005-12-17	Purchase	234.00
14	3	2005-12-17	Purchase	87.23
15	4	2005-12-17	Purchase	8.23
16	5	2005-12-17	Purchase	99.99
17	6	2005-12-17	Purchase	43.23
18	7	2005-12-17	Purchase	89.78
19	8	2005-12-17	Refund	234.44
20	9	2005-12-17	Refund	13.79
21	10	2005-12-17	Purchase	15.71
22	11	2005-12-17	Purchase	64.82
	[EOF]			

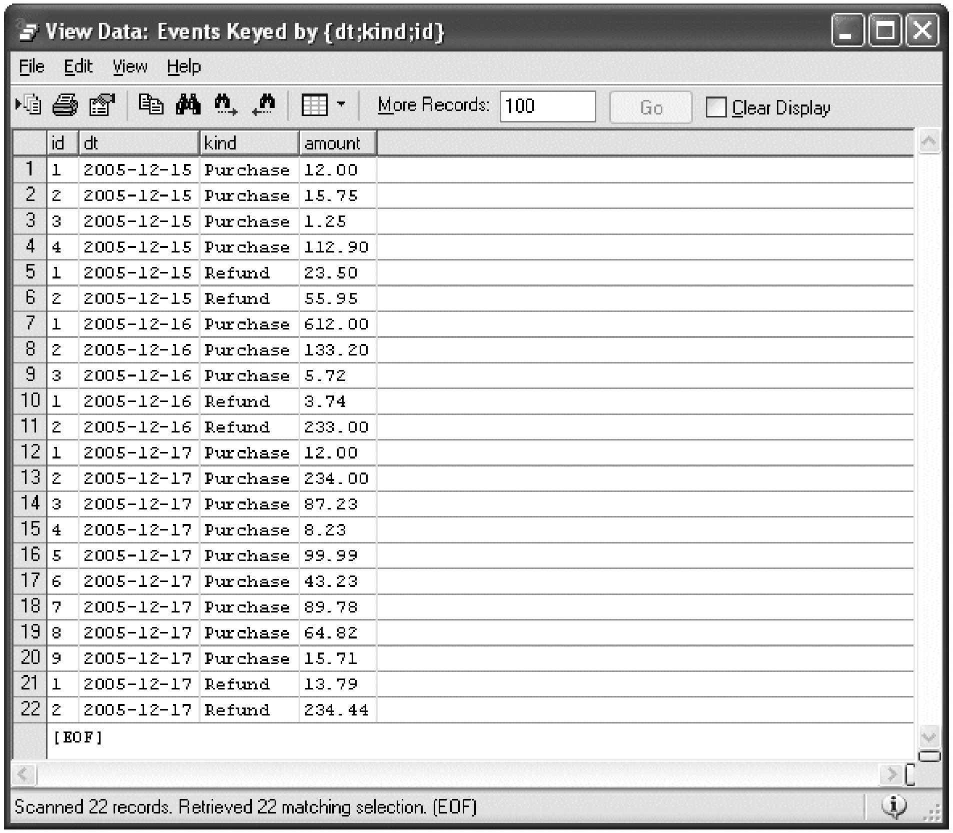
Scanned 22 records. Retrieved 22 matching selection. (EOF)

Unique ID within a two-field key group

The **Scan: Assign id Within {dt; kind}** component has a two-part **key** parameter **{dt; kind}**, so it treats each record with the same pair of **dt** and **kind** values as part of the same key group. Again, it counts the records in each key group and assigns the current **count** within that key group to the **id** field for each record. In the output, the combination of the three fields **dt**, **kind**, and **id** uniquely identifies each record.

NOTE: Like the other two SCAN components in the graph, this is a sorted SCAN (its **sorted_input** parameter is set to **Input must be sorted or grouped**). The other two SCANs work without upstream sorting, because the input data was already sorted on the **dt** field, and **key** was set to either the empty key: **{ }** or the **dt** field. But because the third SCAN has a two-field **key** parameter, **{dt; kind}**, the transform will fail if the input data is not sorted on both of those fields. Therefore, this SCAN must be preceded by a SORT WITHIN GROUPS component that uses **dt** as its **major_key** parameter and **kind** as its **minor_key** parameter.

The output data looks like this:



View Data: Events Keyed by {dt;kind;id}

	id	dt	kind	amount
1	1	2005-12-15	Purchase	12.00
2	2	2005-12-15	Purchase	15.75
3	3	2005-12-15	Purchase	1.25
4	4	2005-12-15	Purchase	112.90
5	1	2005-12-15	Refund	23.50
6	2	2005-12-15	Refund	55.95
7	1	2005-12-16	Purchase	612.00
8	2	2005-12-16	Purchase	133.20
9	3	2005-12-16	Purchase	5.72
10	1	2005-12-16	Refund	3.74
11	2	2005-12-16	Refund	233.00
12	1	2005-12-17	Purchase	12.00
13	2	2005-12-17	Purchase	234.00
14	3	2005-12-17	Purchase	87.23
15	4	2005-12-17	Purchase	8.23
16	5	2005-12-17	Purchase	99.99
17	6	2005-12-17	Purchase	43.23
18	7	2005-12-17	Purchase	89.78
19	8	2005-12-17	Purchase	64.82
20	9	2005-12-17	Purchase	15.71
21	1	2005-12-17	Refund	13.79
22	2	2005-12-17	Refund	234.44
	[EOF]			

Scanned 22 records. Retrieved 22 matching selection. (EOF)

Using a SCAN to generate unique keys

To create a graph with a SCAN that generates unique keys for the records in a dataset, like the SCAN components in making-unique-keys.mp:

> **NOTE:** The scope of uniqueness for the **count** value generated by the SCAN and assigned to an output field is determined by the **key** parameter of the SCAN, as described in "Using the SCAN's key parameter to specify the key field(s) in the output" (page 31).

1. Attach a SCAN component to your input file.
2. Hold down the Shift key and double-click the SCAN component to display the **Transform Editor**.

   > **NOTE:** If the **Transform Editor** opens in **Grid View**, choose **View > Text**.

3. Delete all default transform functions except **initialize**, **scan**, and **finalize**.
4. Define the type **temporary_type** to be a **decimal("")** with an appropriate name (such as **count**) by editing the default definition to read as follows:

```
type temporary_type =
record
  decimal("") count;
end;
```

5. Code the **initialize** function as follows to initialize **count** to zero:

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

6. Code the **scan** function as follows to increment **count** by **1** for each record it processes:

```
temp::scan(temp, in) =
begin
  temp.count :: temp.count + 1;
end;
```

7. Code the **finalize** function as follows so that for each record, the SCAN will assign the value of **count** to the appropriate output field (for example, **id**) and will assign all other input fields to the output fields with the same names:

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

Using the SCAN's key parameter to specify the key field(s) in the output

If you use a SCAN transform like the one in **making-unique-keys.mp** to generate a unique identifier, the **key** parameter of the SCAN component determines the scope within which the identifier is unique, and thus also determines the key field or fields that uniquely identify each record in the output.

TO SPECIFY THIS SCOPE	WHICH MEANS	SET THE KEY PARAMETER TO THIS
Global	Every record across the entire output dataset is uniquely identified by the output field to which the SCAN's **count** field is assigned. In the output for the first SCAN in **making-unique-keys.mp**, the unique key is the **id** field.	The empty key: **{ }**
A specific key group	Each record within a key group is uniquely identified by the combination of the field(s) specified in the **key** parameter, and the output field to which the SCAN's **count** field is assigned: • In the output for the second SCAN in **making-unique-keys.mp**, the unique key is the combination of the **dt** and **id** fields. • In the output for the third SCAN in **making-unique-keys.mp**, the unique key is the combination of the **dt**, **kind**, and **id** fields.	The field or fields that form the key of the key group **NOTE:** If the input file is not already sorted on this field or these fields, you can use a SORT or SORT WITHIN GROUPS as necessary, or use an in-memory SCAN instead.

Filling gaps

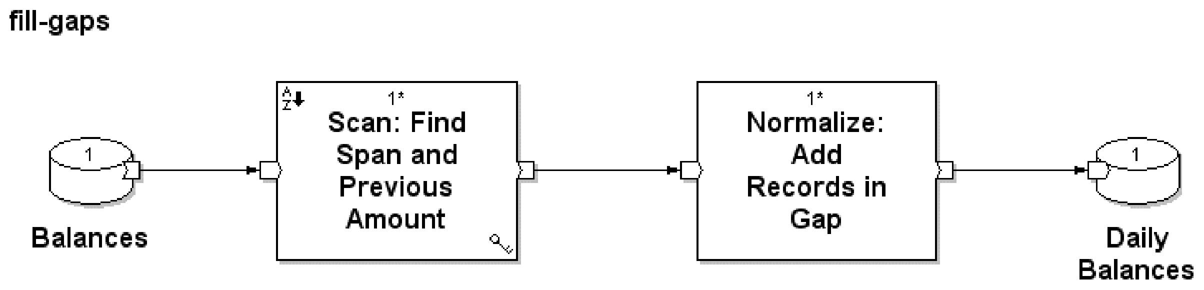
Sometimes data is provided at a coarser granularity than is required for a given business task. For example, your input data may contain account balances only for days with activity, but you may need to process that data at the daily level. The input data is effectively “missing” records for days on which there was no account activity.

The recipe in this chapter demonstrates how to generate daily balances for days or sequences of days on which an account showed no activity and for which the input data therefore lacks records.

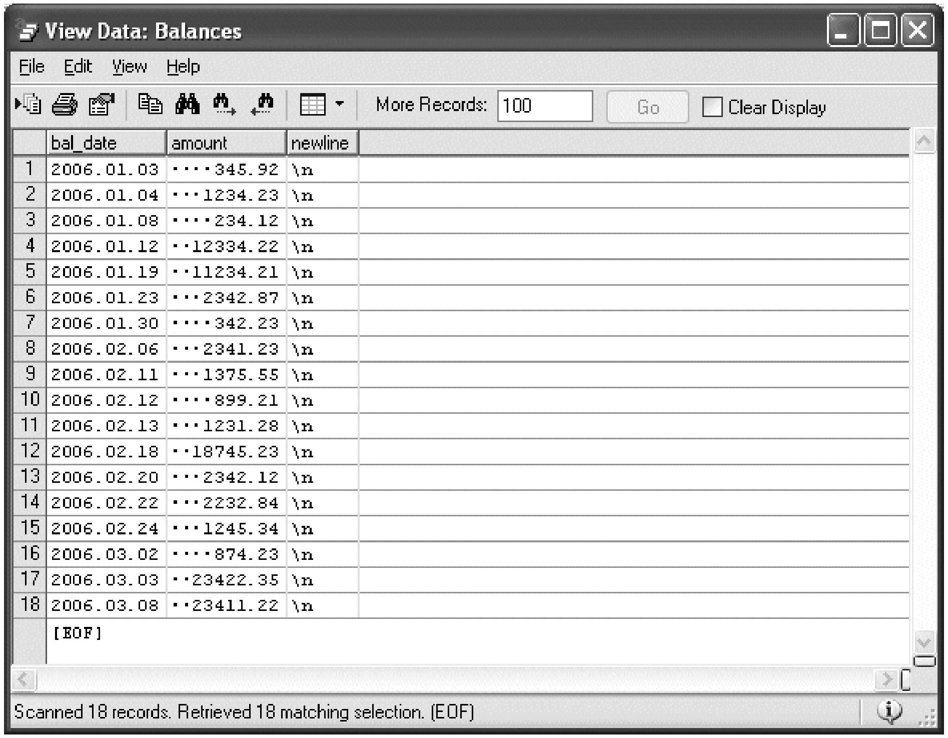
Graph for filling gaps

The graph **fill-gaps.mp** (in the **appetizers** sandbox) does this by using the following components:

- SCAN — Detects each gap in the input data and calculates the span of the gap (that is, the number of days for which the graph needs to generate balances). The SCAN's output also preserves the last balance preceding each gap, so that it can be used in the new records.
- NORMALIZE — Uses the spans to generate the correct number of records to fill each gap, and carries the balance from the last record preceding each gap into the generated records.


fill-gaps
1
Balances
1*
Scan: Find Span and Previous Amount
1*
Normalize: Add Records in Gap
1
Daily Balances

The input file for **fill-gaps.mp** contains the following data, which has gaps between several of the dates:



	bal_date	amount	newline
1	2006.01.03	····345.92	\n
2	2006.01.04	···1234.23	\n
3	2006.01.08	····234.12	\n
4	2006.01.12	··12334.22	\n
5	2006.01.19	··11234.21	\n
6	2006.01.23	···2342.87	\n
7	2006.01.30	····342.23	\n
8	2006.02.06	···2341.23	\n
9	2006.02.11	···1375.55	\n
10	2006.02.12	····899.21	\n
11	2006.02.13	···1231.28	\n
12	2006.02.18	··18745.23	\n
13	2006.02.20	···2342.12	\n
14	2006.02.22	···2232.84	\n
15	2006.02.24	···1245.34	\n
16	2006.03.02	····874.23	\n
17	2006.03.03	··23422.35	\n
18	2006.03.08	··23411.22	\n
	[EOF]		

Scanned 18 records. Retrieved 18 matching selection. (EOF)

Configuring the SCAN

This section describes how to configure a SCAN to detect each gap in the input records and calculate the span of the gap.

To configure the SCAN:

1. Set the key to the empty key: **{ }** so that all records belong to the same key group.
2. Calculate the length of the gaps as follows:

   a. In the SCAN's **temporary_type**, initialize **this_date** and **prev_date** to a special value, such as the empty string:

```
type temporary_type =
  record
    date("YYYY.MM.DD") this_date = '';
    date("YYYY.MM.DD") prev_date = '';
    ...
```

   NOTE: Using a special value allows you to treat the first record differently (see the **finalize** function in Step **c**).

   b. In the **scan** function, write the current balance date to the **this_date** field of the current temporary record (for use in processing the next record), and the **this_date** field of the previous temporary record to the **prev_date** field of the current temporary record:

```
out :: scan(temp, in) =
begin
  out.this_date :: in.bal_date;
  out.prev_date :: temp.this_date;
  ...
```

c. In the **finalize** function, output the **in.bal_date** and calculate the **span** by subtracting the date of the previous record from the date of the current record (or returning **1** if the previous record has an invalid date):

```
out :: finalize(temp, in) =
begin
  out.* :: in.*;
  ...
  out.span :1: if (not is_blank(temp.prev_date))
      in.bal_date - temp.prev_date;
  out.span :2: 1;
  ...
```

The following table shows what this code does with the first four records:

		TEMPORARY VALUE		OUTPUT
RECORD	in.bal_date	this_date	prev_date	SPAN
1	2006.01.03	2006.01.03	" "	1
2	2006.01.04	2006.01.04	2006.01.03	1
3	2006.01.08	2006.01.08	2006.01.04	4
4	2006.01.12	2006.01.12	2006.01.08	4

3. Calculate the values needed to fill the gaps as follows:

a. In the SCAN's **temporary_type**, initialize **this_amount** and **prev_amount** to zero:

```
type temporary_type =
  record
    ...
    decimal(10.2) this_amount = 0;
    decimal(10.2) prev_amount = 0;
  end;
```

b. In the **scan** function, write the current balance amount to the **this_amount** field of the current temporary record (for use in processing the next record), and the **this_amount** field of the previous temporary record to the **prev_amount** field of the current temporary record:

```
out :: scan(temp, in) =
begin
  ...
  out.this_amount :: in.amount;
  out.prev_amount :: temp.this_amount;
end;
```

c. In the **finalize** function, output **in.amount** and **prev_amount**:

```
out :: finalize(temp, in) =
begin
  out.* :: in.*;
  out.prev_amount :: temp.prev_amount;
  ...
```

The following table shows what this code does with the first four records:

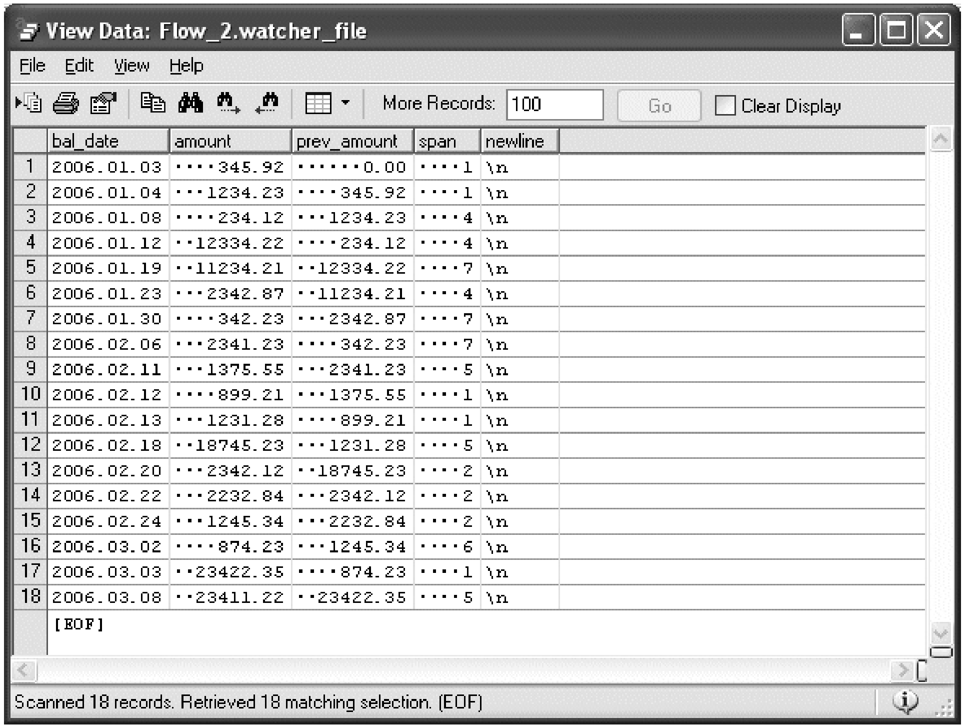
		TEMPORARY VALUE		OUTPUT	
RECORD	in.amount	this_amount	prev_amount	amount	prev_amount
1	345.92	345.92	0	345.92	0
2	1234.23	1234.23	345.92	1234.23	345.92
3	234.12	234.12	1234.23	234.12	1234.23
4	12334.22	12334.22	234.12	12334.22	234.12

Following is the complete transform code for the SCAN:

```
type temporary_type =
  record
    date("YYYY.MM.DD") this_date = '';
    date("YYYY.MM.DD") prev_date = '';
    decimal(10.2) this_amount = 0;
    decimal(10.2) prev_amount = 0;
  end;

out :: scan(temp, in) =
begin
  out.this_date :: in.bal_date;
  out.prev_date :: temp.this_date;
  out.this_amount :: in.amount;
  out.prev_amount :: temp.this_amount;
end;

out :: finalize(temp, in) =
begin
  out.* :: in.*;
  out.prev_amount :: temp.prev_amount;
  out.span :1: if (not is_blank(temp.prev_date))
      in.bal_date - temp.prev_date;
  out.span :2: 1;
end;
```

The SCAN's output

If you want to view the output of the SCAN, choose **Debugger > Enable Debugger**, right-click the **out** flow, and click **Watcher on Flow**. The output should look like this:



	bal_date	amount	prev_amount	span	newline
1	2006.01.03	····345.92	······0.00	····1	\n
2	2006.01.04	···1234.23	····345.92	····1	\n
3	2006.01.08	····234.12	···1234.23	····4	\n
4	2006.01.12	··12334.22	····234.12	····4	\n
5	2006.01.19	··11234.21	··12334.22	····7	\n
6	2006.01.23	···2342.87	··11234.21	····4	\n
7	2006.01.30	····342.23	···2342.87	····7	\n
8	2006.02.06	···2341.23	····342.23	····7	\n
9	2006.02.11	···1375.55	···2341.23	····5	\n
10	2006.02.12	····899.21	···1375.55	····1	\n
11	2006.02.13	···1231.28	····899.21	····1	\n
12	2006.02.18	··18745.23	···1231.28	····5	\n
13	2006.02.20	···2342.12	··18745.23	····2	\n
14	2006.02.22	···2232.84	···2342.12	····2	\n
15	2006.02.24	···1245.34	···2232.84	····2	\n
16	2006.03.02	····874.23	···1245.34	····6	\n
17	2006.03.03	··23422.35	····874.23	····1	\n
18	2006.03.08	··23411.22	··23422.35	····5	\n

[EOF]

Scanned 18 records. Retrieved 18 matching selection. (EOF)

Writing the NORMALIZE transform

The NORMALIZE component reads the output of the SCAN and uses it to generate the “missing” records. The NORMALIZE reads the **span** field to determine how many output records are needed to fill each gap, generates the records, and writes the value from **prev_amount** to the **amount** field as necessary to fill the gaps.

To generate the missing records with the appropriate values:

1. In the NORMALIZE’s **length** function, use the **span** field from the SCAN component to specify the number of output records needed to fill the gap between the current and the next input record:

```
out :: length(in) =
begin
  out :: in.span;
end;
```

2. In the NORMALIZE’s **normalize** function, generate a record for each date in the gap as follows:

   a. Calculate the date by subtracting the difference between the entire span and the index into it from the current balance date, and adding **1**:

```
out :: normalize(in, index) =
begin
  out.bal_date :: in.bal_date - (in.span - index) + 1;
```

   This returns subsequent dates in the range covering the gap.

   b. Use the previous amount for records in the gap (**index < in.span - 1**), and the current amount for the final record:

```
  out.amount :1: if (index < in.span - 1) in.prev_amount;
  out.amount :2: in.amount;
  out.* :: in.*;
end;
```

Specifying the output record format

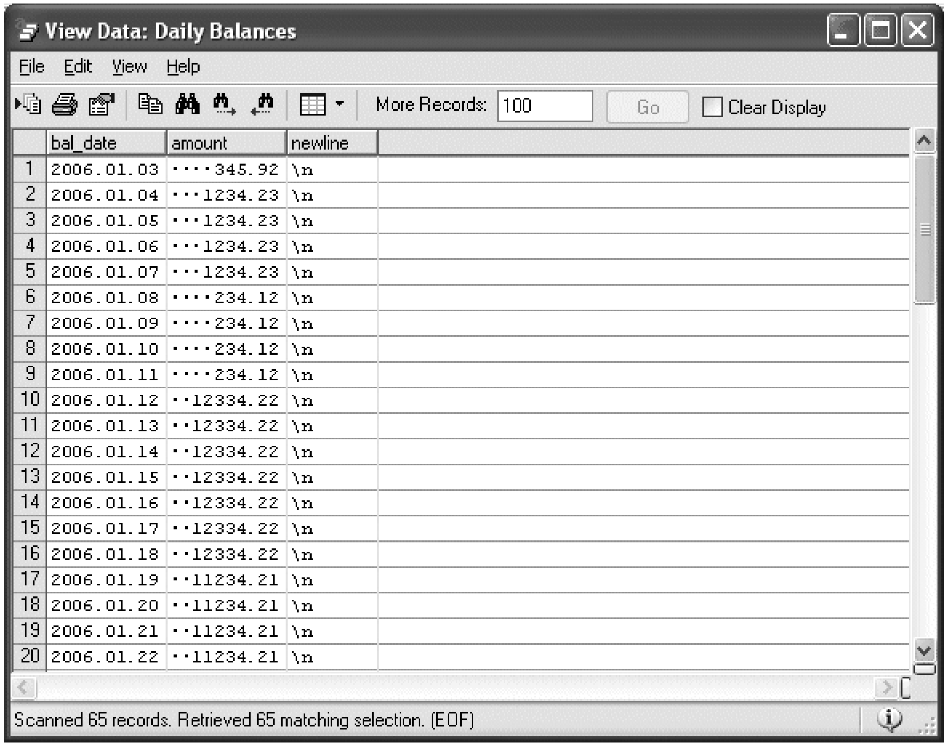
To specify the output record format:

1. Attach an OUTPUT FILE component to the SCAN.
2. Specify the URL to a file containing the following record format, or embed it directly into the OUTPUT FILE:

```
record
  date("YYYY.MM.DD") bal_date;
  decimal(10.2) amount;
  string(1) newline = '\n';
end
```

Running the graph and viewing the results

When you run the graph and view the results, the first 20 records should look like this:



	bal_date	amount	newline
1	2006.01.03	····345.92	\n
2	2006.01.04	···1234.23	\n
3	2006.01.05	···1234.23	\n
4	2006.01.06	···1234.23	\n
5	2006.01.07	···1234.23	\n
6	2006.01.08	····234.12	\n
7	2006.01.09	····234.12	\n
8	2006.01.10	····234.12	\n
9	2006.01.11	····234.12	\n
10	2006.01.12	··12334.22	\n
11	2006.01.13	··12334.22	\n
12	2006.01.14	··12334.22	\n
13	2006.01.15	··12334.22	\n
14	2006.01.16	··12334.22	\n
15	2006.01.17	··12334.22	\n
16	2006.01.18	··12334.22	\n
17	2006.01.19	··11234.21	\n
18	2006.01.20	··11234.21	\n
19	2006.01.21	··11234.21	\n
20	2006.01.22	··11234.21	\n

Scanned 65 records. Retrieved 65 matching selection. (EOF)

Measuring performance

When building graphs in Ab Initio software, you will often discover that you can accomplish the same task in more than one way. In deciding on the best approach, you may need to consider various factors, and although clarity and maintainability may be paramount in some cases, performance will often be a concern. However, before you can make an informed decision about structuring your graph to maximize performance, you must have a methodology for measuring it.

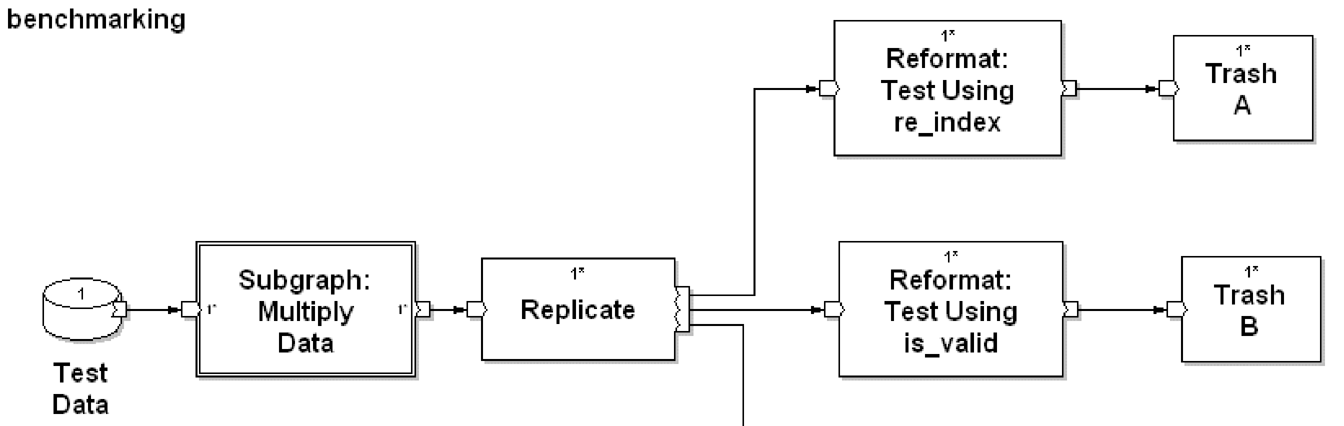
The recipes in this chapter explain how to compare the performance of various approaches to a task while minimizing or eliminating factors that could distort the comparison. It describes the following:

- A simple benchmarking graph (next)
- A benchmarking graph that checks its results (page 49)
- Other suggestions for measuring performance (page 51)

Simple benchmarking graph

The graph **benchmarking.mp** (in the **appetizers** sandbox) measures the speed of three different transform functions that test the validity of account numbers. It minimizes the effects of the following factors:

- **Startup time** Graphs can spend a significant amount of time just starting up. This graph minimizes the effect of startup time by using the **Multiply Data** subgraph to generate a large amount of data. For details on this subgraph, see "Reusable, parameterized data multiplier subgraph" (page 16).
- **Environmental factors** Factors such as system load can vary between runs of a graph, thereby affecting the accuracy and repeatability of performance measurements. This graph eliminates such factors by running all three transform functions on replicated data in the same graph.



benchmarking
Test Data
Subgraph: Multiply Data
Replicate
Reformat: Test Using re_index
Reformat: Test Using is_valid
Reformat: Test Using string_filter_out
Trash A
Trash B
Trash C

To create a graph like benchmarking.mp:

1. If your input dataset does not contain enough records to test different approaches to the task, attach a **Multiply Data** subgraph to your input dataset, and set its **multiplier** parameter appropriately.

   In **benchmarking.mp**, the test data consists of only four records, and **multiplier** is set to **500000**. For details on creating a **Multiply Data** subgraph, see "Reusable, parameterized data multiplier subgraph" (page 16).

2. Attach a REPLICATE component so that you can replicate the data once for each approach you want to test.
3. Attach the appropriate component for each test to a separate flow from the REPLICATE, and code the components appropriately.

   In **benchmarking.mp**, three REFORMAT components each use a different transform to validate the account numbers. For details on the transforms, see "About the REFORMAT components in benchmarking.mp" (page 48).

4. Send the resulting data to a TRASH component.

   In this case simply discard the output. When you need to be sure that different approaches lead to the same results, you should also check the accuracy of the outputs as described in "Checking the accuracy of the outputs" (page 49).

5. If appropriate, turn on **Text Tracking** (see the sidebar for details).
6. Run the graph and view the tracking data for each component.

   The section **Phase 0 ended** shows that the **Reformat: Test Using is_valid** component uses less CPU time than the other two REFORMAT components:

```
-----------------------------------------------------------------
Feb 13 17:17:24   Phase 0 ended (14 seconds)
          CPU Time   Status Skew Vertex
          3.359 [   : 1] 0% Reformat_Test_Using_is_valid
          5.453 [   : 1] 0% Reformat_Test_Using_re_index
```

TEXT TRACKING

Text tracking provides CPU utilization times to the precision of hundredths (and sometimes thousandths) of a second, whereas the tracking information in the GDE **Tracking Details** dialogs is often rounded to the nearest second, possibly obscuring small differences in performance. For this reason, text tracking is sometimes a better tool for examining performance.

When using Version 2.14 (or higher) of the Co>Operating System® and Version 1.14 (or higher) of the GDE, you can display both text tracking and GDE tracking (in the **Tracking Details** dialog). With earlier versions, you must choose between text tracking and the **Tracking Details** dialog.

To turn on text tracking:

1. Choose **Run > Settings**.
2. Click the **Tracking** tab.
3. Select the **Text Tracking** checkbox.

Text tracking appears in the **Job Output** dialog as the graph runs.

```
           4.141 [   : 1] 0% Reformat_Test_Using_string_filter_out
           1.078 [   : 1] 0% Replicate
           1.859 [   : 1] 0% Subgraph_Multiply_Data.Normalize
           0.313 [   : 1] 0% Trash_A
           0.422 [   : 1] 0% Trash_B
           0.297 [   : 1] 0% Trash_C
-----------------------------------------------------------------
```

About the REFORMAT components in benchmarking.mp

In **benchmarking.mp**, each of the three REFORMAT components uses a different transform function to test the validity of the **account_num** field in the input dataset:

- **re_index** — Uses regular expression matching to check whether the account number consists only of digits:

```
out::reformat(in) =
begin
  out.account_num :1: if (re_index(in.account_num,
      "[^0-9]+") == 0)
      in.account_num;
  out.account_num :: 0;
  out.* :: in.*;
end;
```

- **is_valid** — Checks whether the account number is a valid decimal number:

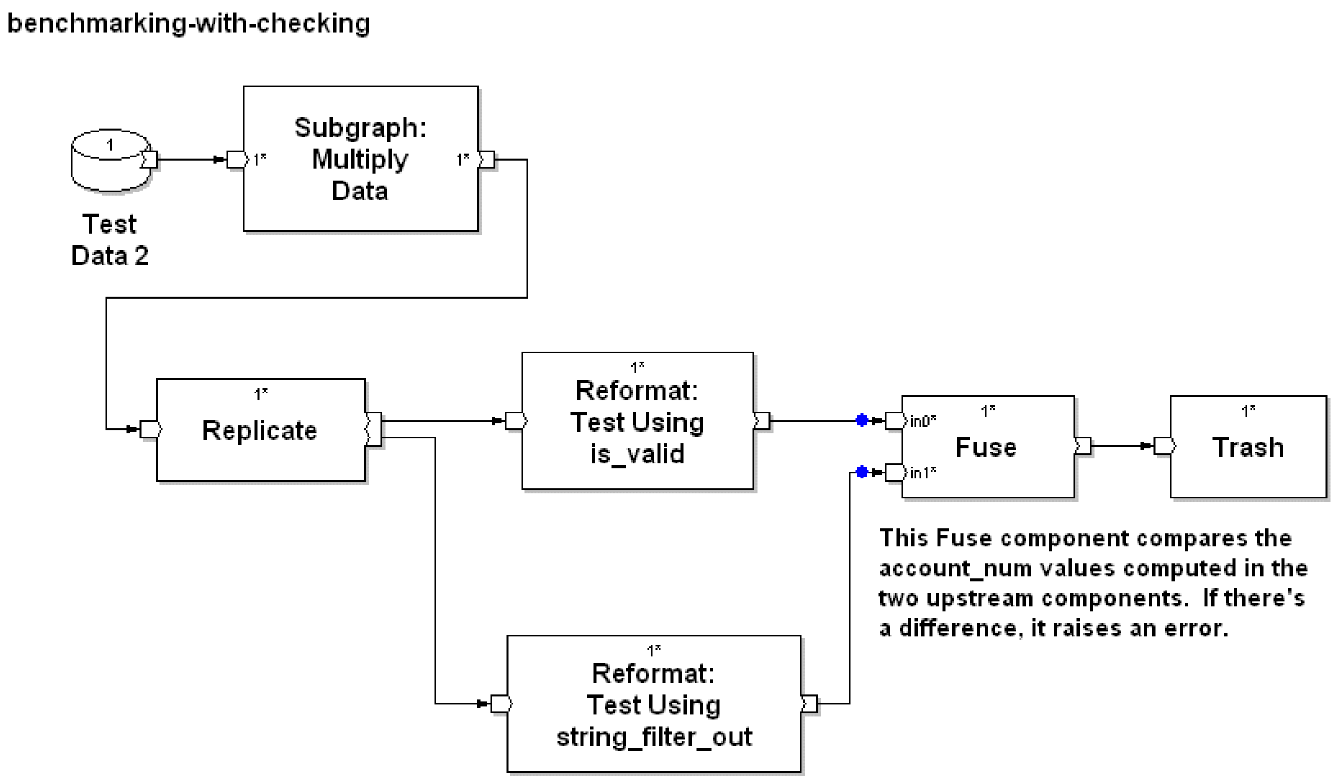
```
out::reformat(in) =
begin
  out.account_num :1: if (is_valid(in.account_num))
      in.account_num;
  out.account_num :: 0;
  out.* :: in.*;
end;
```

- **string_filter_out** — Checks whether each character in the **account_num** field is a digit:

  ```
  out::reformat(in) =
  begin
    out.account_num :1: if (length_of(string_filter_out(
        in.account_num, "0123456789")) == 0)
        in.account_num;
    out.account_num :: 0;
    out.* :: in.*;
  end;
  ```

Checking the accuracy of the outputs

Obviously it does not make sense to compare the performance of transforms whose accuracy is questionable. However, if you do an initial comparison and then change your benchmarking graph to improve the performance of any of its transforms, you may also inadvertently change its behavior in unintended ways. To recheck the accuracy of your transforms, create a graph such as **benchmarking-with-checking.mp**:


benchmarking-with-checking
1
Test Data 2
1*
Subgraph: Multiply Data
1*
1*
Replicate
1*
Reformat: Test Using is_valid
in0*
in1*
1*
Fuse
1*
Trash
1*
Reformat: Test Using string_filter_out
This Fuse component compares the account_num values computed in the two upstream components. If there's a difference, it raises an error.

This graph uses a FUSE component as a quick and easy way to check specifically for differences between the values of the **account_num** field in the outputs of the previous graph's two fastest tests, **Test Using is_valid** and **Test Using string_filter_out**. To check for more general differences (such as added, deleted, or changed records), see "Finding differences" (page 65) and "Automated generation of record formats and transforms" (page 79).

In **Text View**, the FUSE component's transform looks like this:

```
out::fuse(in0, in1) =
begin
  out :: if (in0.account_num != in1.account_num)
             force_error(
                string_concat("Mismatch in account_num (in0='",
                               in0.account_num, "'; in1='",
                               in1.account_num, "')"))
         else
            in0;
end;
```

If you run this graph, you will see that **Test Using is_valid** accepts the account number **45.8**, whereas **Test Using string_filter_out** "catches" it and sets the **account_num** field to zero:

```
Error from Component 'Fuse', Partition 0
[U119]
Exceeded reject threshold after 1 rejected records.
While executing transform fuse:

Mismatch in account_num (in0='    45.8'; in1='       0') in assignment
for out.
```

ABOUT RECORD ORDER

REFORMAT components do not change the order of records, so you can use a FUSE to compare the results of the different transforms. If you are measuring the performance of collections of components that may result in a different ordering of data, it may be appropriate to sort before a FUSE or use the more general data differencing technique described in "Finding differences" (page 65) and "Automated generation of record formats and transforms" (page 79).

Other suggestions for measuring performance

The graph **benchmarking.mp** focuses on CPU time, and it measures the performance of transforms in individual components. You can use similar techniques to measure the performance of collections of components in the same graph.

To measure the performance of different approaches when you cannot use the same graph:

1. Do at least three runs of each approach, perhaps alternating them.

2. Take the minimum time for each approach. Use the minimum (instead of the average, for example) — because environmental factors may make a graph slower, but they will never make it faster.

Startup time

The effect of startup time is proportionally larger when processing small data than when processing large data, so you may need to take it into account when measuring or estimating performance. Following is a description of how to do this for the sample record numbers and times listed in the table below.

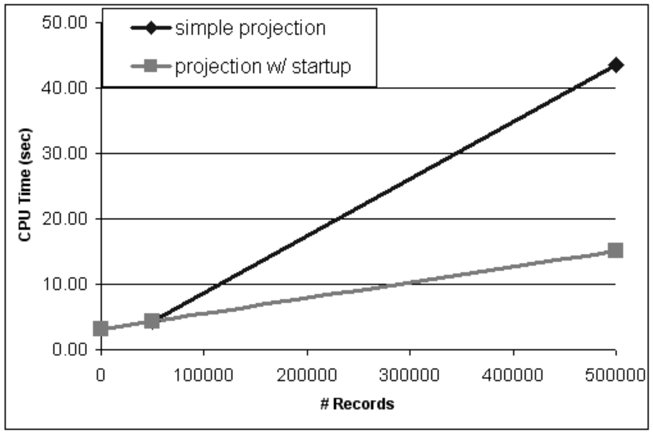
	TIME (SEC)		
RECORDS	MEASURED (ACTUAL)	SIMPLE PROJECTION	PROJECTION WITH STARTUP
1	3.15	—	3.15
50,000	4.35	4.35	4.35
500,000	—	43.50	15.15

To project the time (incorporating startup time) for a run on big data:

1. Run the graph on *very* small data (one record) to get the startup time (here **3.15** sec).
2. Run the graph on small data (here **50,000** records take **4.35** sec).
3. Subtract the startup time from the total time for small data to get a scaling factor for use with large data (here **4.35** – **3.15**).
4. Multiply the ratio of big data to small data by the scaling factor, and then add back the startup time — for example:

```
(500,000/50,000) * (4.35 - 3.15) + 3.15 = 15.15
```

Taking startup time into account makes performance projections much more accurate. The following chart shows that the difference between a projection that incorporates startup time and one that does not can be huge.



Elapsed time

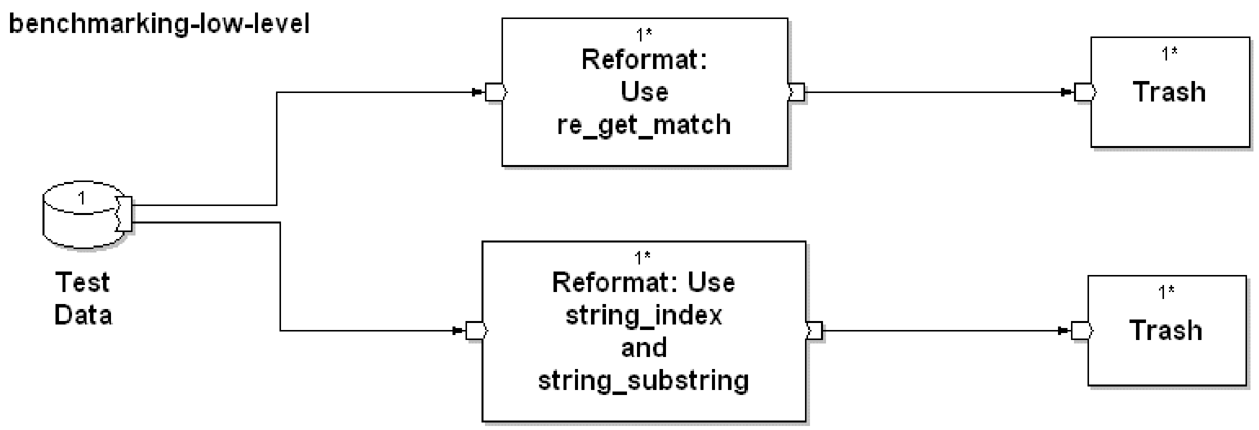
You can measure elapsed time in different ways. Since elapsed time is sensitive to load, you should run on an unloaded system and/or do multiple runs and take the minimum, as described earlier. You can get information on elapsed time in several ways:

- **Elapsed time for an entire graph** — Displayed on the bottom right, in the GDE status bar. If the status bar is not visible, choose **View > Toolbars > Status Bar**.

- **Elapsed time per phase** — Provided in these ways:
  - As part of *text tracking,* which outputs the performance data for the graph. (For information on text tracking, see "Text tracking" on page 47.)
  - In the graph's summary file. For details on the summary file, see "Writing tracking information to a summary file and finding the flows needed for the audit" (page 113) and Ab Initio Help.

Low-level operations

To measure the cost of a very-low-level operation (processing you might do multiple times per record or field) and factor out per-record processing and I/O overhead, use a **for** loop to execute the operation *N* times per record. The graph **benchmarking-low-level.mp** uses this technique to compare the performance of two different DML functions that extract strings with a specific pattern.


benchmarking-low-level
1*
Reformat: Use re_get_match
1*
Trash
1
Test Data
1*
Reformat: Use string_index and string_substring
1*
Trash

To create a graph like benchmarking-low-level.mp:

1. Add an input parameter (for example, **REPEAT_AMT**; see the sidebar for details), and set its value to the number of times the low-level operation must be performed to eliminate the effects of per-record processing and I/O overhead. In **benchmarking-low-level.mp**, the parameter is set to **500000**.

   To determine a reasonable value, start with a low number and increase it until the weight of the other factors is low enough that the processing time increases almost exactly in step with **REPEAT_AMT**.

2. Add an input file containing your test data.

3. Attach two REFORMAT components to the input file and code the transform in each to perform the desired operation the number of times specified in the input parameter.

   For example, the transforms in the two REFORMAT components of **benchmarking-low-level.mp** are coded as follows (the benchmarked code is underlined):

   Reformat: Use re_get_match

```
out::reformat(in) =
begin
  let string("") result = "";
  let integer(4) i = 0;
  for (i, i < ${REPEAT_AMT})
        result = re_get_match(in.name, "Bad.*");
  out.* :: in.*;
  out.name :1: result;
  out.name :: "";
end;
```

   Reformat: Use string_index and string_substring

```
out::reformat(in) =
begin
  let string("") result = "";
  let integer(4) i = 0;
  for (i, i < ${REPEAT_AMT})
```

TO ADD AND SET AN INPUT PARAMETER

1. Choose **Edit** > **Parameters.**
2. In the **Name** field of the **Edit Parameters** dialog, type a name for the parameter (for example, **REPEAT_AMT**).
3. In the **Value** field, type a value for the parameter (for example, **500000**).

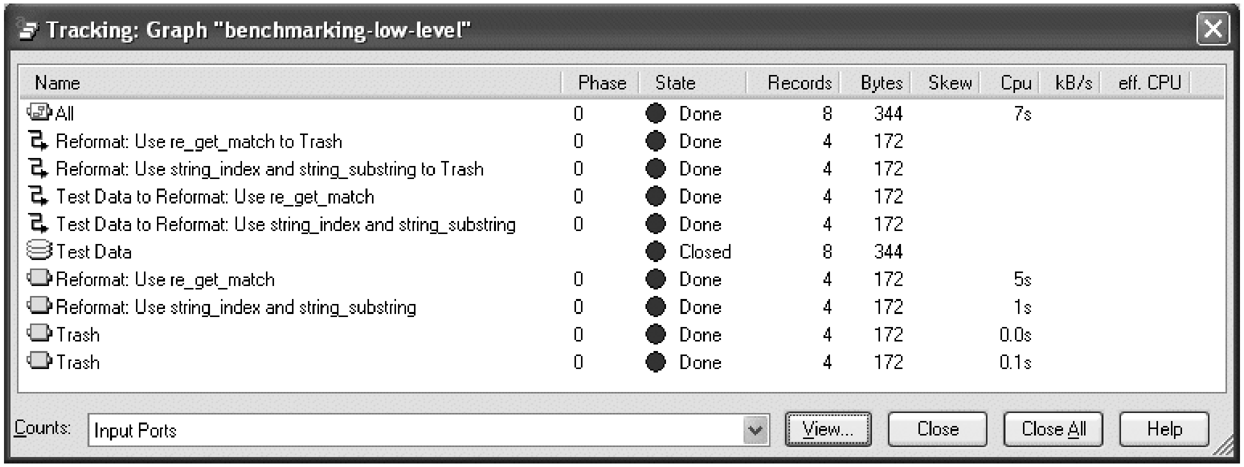
DO NOT USE REPLICATE COMPONENTS UNNECESSARILY

When you need to read data from a file multiple times before processing it in any way, simply attach multiple flows to the file, as in **benchmarking-low-level.mp**. This allows downstream components to run as fast as they can, without waiting for others to receive their copy of the data from the REPLICATE. While the effect may be small in simple graphs such as these, it can be significant in larger graphs, so avoiding REPLICATE in all such situations is a good practice.

```
        if (string_index(in.name, "Bad") > 0)
          result = string_substring(in.name,
                                      string_index(in.name, "Bad"),
                                      string_length(in.name));
      out.* :: in.*;
      out.name :1: result;
      out.name :: "";
    end;
```

4. Run the graph.
5. When the graph finishes, right-click anywhere on the graph and choose **Tracking Detail**.

   The information in the **Cpu** column shows that the expression using **string_index** and **string_substring** is more efficient for this particular operation.



Tracking: Graph "benchmarking-low-level"

Name	Phase	State	Records	Bytes	Skew	Cpu	kB/s	eff. CPU
All	0	● Done	8	344		7s		
Reformat: Use re_get_match to Trash	0	● Done	4	172				
Reformat: Use string_index and string_substring to Trash	0	● Done	4	172				
Test Data to Reformat: Use re_get_match	0	● Done	4	172				
Test Data to Reformat: Use string_index and string_substring	0	● Done	4	172				
Test Data		● Closed	8	344				
Reformat: Use re_get_match	0	● Done	4	172		5s		
Reformat: Use string_index and string_substring	0	● Done	4	172		1s		
Trash	0	● Done	4	172		0.0s		
Trash	0	● Done	4	172		0.1s		

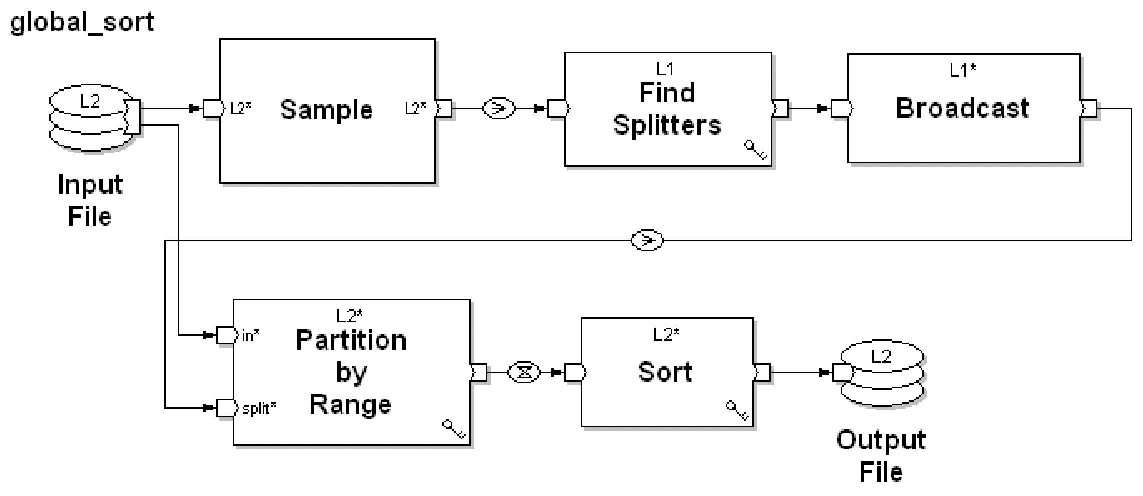
Counts: Input Ports

Global sort

While it is quite common to maintain sorted data in multifile datasets, such data is usually sorted only within each partition, not across partitions. In some situations, however, a global sort order is required. For example, a global sort is a key step in computing medians or quantiles in parallel; see “Quantiling” (page 89). The recipes in this chapter describe a simple parallel global sort and variations on it to produce sorted serial output.

Simple global sort

To do a simple global sort, create a graph such as **global_sort.mp** (in the **global-sort** sandbox) by following the steps summarized after the figure below and described in detail in the rest of this chapter.



1. Sample the data (page 59).

   Because your graph will need to partition the input data by key range, you need to calculate which values of the key field will divide the data into groups of roughly equal size. Calculating these dividing points, or *splitters*, by reading all the records would be inefficient and unnecessarily precise; it is better to take a random sample of the data and calculate the splitters just for the sample.

2. Calculate and broadcast the splitter values for partitioning the data by key (page 59).

   The FIND SPLITTERS component reads its entire input flow into memory, sorts it by key, and outputs records that are equally spaced in the sorted order, thus dividing the input records as evenly as possible. The number of output records is one less than the number of partitions, which you specify as the **num_partitions** parameter. FIND SPLITTERS is designed to produce the division of records that will yield the lowest possible skew when the splitters are used by the PARTITION BY RANGE component.

The BROADCAST component writes a copy of the splitters to each of its output flow partitions and sends them to the **split** port of the PARTITION BY RANGE component.

3. Partition the data by the splitter values and sort each partition (page 60).

   The PARTITION BY RANGE component uses the splitters to direct the records from the full input dataset to the appropriate partitions, with each partition getting approximately equal numbers of records with successively later keys in the sort order. The SORT component then sorts the records in each partition to produce the final output.

Sampling the data

To produce a relatively small random sample of the input data for use in determining the splitters:

- Connect a SAMPLE component to the input file and set the following parameters:
  - **sample_size** (number of records)
  - **max_skew** (optional: default is **1**)

You will probably get acceptable results simply by choosing a **sample_size** that results in a sample of about 10 MB, and using the default **max_skew** value (**1.0**). However, if you want to take a more careful approach, follow the guidelines described in the sidebar.

Calculating and broadcasting the splitters

To calculate and broadcast the splitters:

1. To sort the records according to a key specifier and find the ranges of key values that divide the records approximately evenly into a specified number of partitions, connect a FIND SPLITTERS component to the SAMPLE component and configure it as follows:

   a. Set its key specifier to the field on which to sort. (In **global_sort.mp**, this is the **id** field.)

ABOUT SAMPLE SIZE AND SKEW

If simply calculating a sample size of about 10 MB is not acceptable, and you want to have more precise control over the skew in the sorted partitions, choose a value for **sample_size** by balancing these considerations:

- The **sample_size** parameter, multiplied by the size of the records in the input, should be less than the amount of memory to be used by the FIND SPLITTERS component. FIND SPLITTERS must run in a serial layout (see the next section) so that it can bring records from all partitions together. If it runs out of memory, the graph will fail.
- The sample size should be large enough to produce splitters that are reasonably close to the real splitters and thus ensure a low skew. A large sample is a more accurate representation of the input data; the larger the sample, the smaller the skew.

For more information, see the SAMPLE component in Ab Initio Help.

b. Use the following command to set the **num_partitions** parameter to be dynamically calculated at runtime using the value of **$AI_MFS**:

```
$(m_expand -n $AI_MFS/global_sort_in.dat)
```

The FIND SPLITTERS component needs to be given the degree of parallelism of the layout it is partitioning to. However, you should not hardcode **num_partitions**, as you might want it to be different in different environments (for example, development versus production). It should also not be a separate parameter, because that would make it too easy to choose one depth for **$AI_MFS** and another depth for this parameter.

c. Set the **Interpretation** of **num_partitions** to **shell**.

d. On the **Layout** tab of the **Properties** dialog, set the **URL** to **$AI_SERIAL**.

2. To spread the splitters back into the parallel layout of the rest of the graph, connect a BROADCAST component to the FIND SPLITTERS component.

Partitioning by range and sorting the partitions

To divide the data into partitions according to the key field ranges calculated by the FIND SPLITTERS component:

1. Add a PARTITION BY RANGE component and configure it as follows:

   a. Set its key to the same field as the one used in the FIND SPLITTERS component, or use a more specific key if necessary. (In **global_sort.mp**, this is the **id** field.)

   b. Connect its **split** port to the BROADCAST component.

   c. Connect its **in** port to the input file in the main graph.

2. To sort the newly partitioned data by **id** within each partition, attach a SORT component to the PARTITION BY RANGE component and connect it to an output file.

3. Configure the SORT component as follows:

   a. Set the key to the same field as the one used in the FIND SPLITTERS and PARTITION BY RANGE components. (In **global_sort.mp**, this is the **id** field.)

   b. Set **max-core** as desired or leave the default (**100** MB).

Checking the results

To test the results of running global_sort.mp:

- Run **global_sort_check.mp**.

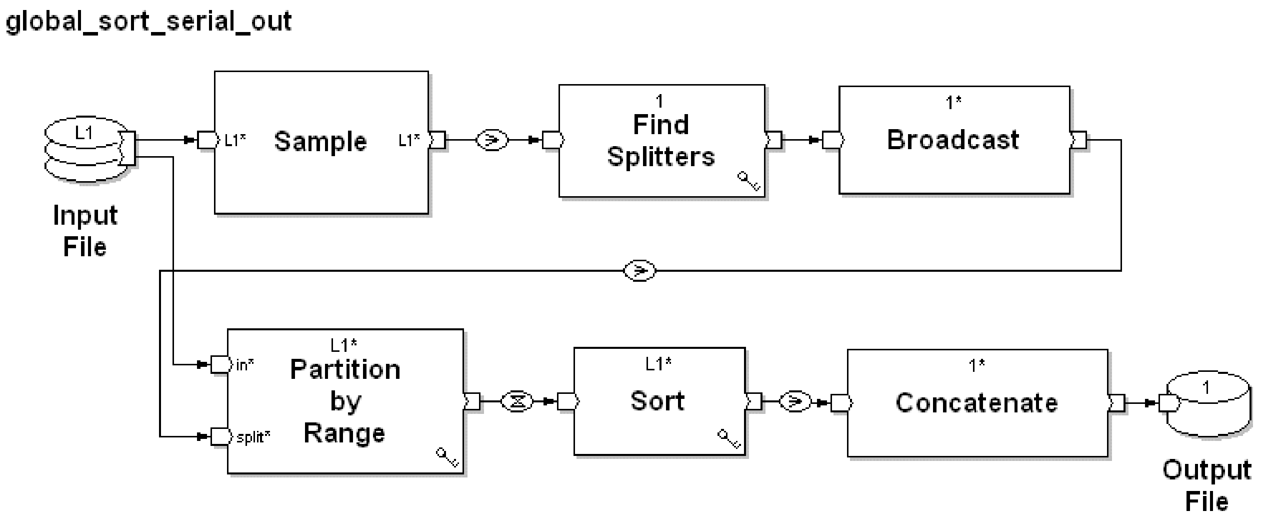
If this graph finishes without error, the output is correctly sorted.

PRESORTED INPUT

If the input file is already sorted within each partition, but not between partitions, the performance of **global_sort.mp** is poorer than you might expect. This is because the graph partially serializes: at first, all the partitions of the PARTITION BY RANGE component send records to the first partition of the SORT component. The node on which this partition is running is working hard to sort, while the other nodes are mostly idle. After a while, all the partitions of the PARTITION BY RANGE component switch to sending records to the second partition of the SORT component. Now the second node is working hard, while the others are idle. This pattern continues, with all partitions of the PARTITION BY RANGE component switching at about the same time. The overall performance of the graph is only slightly better than if it were running without any data parallelism.

Producing serial output

If your goal is to produce a serial file, you might be tempted simply to add a CONCATENATE component after the SORT, as in **global_sort_serial_out.mp**:

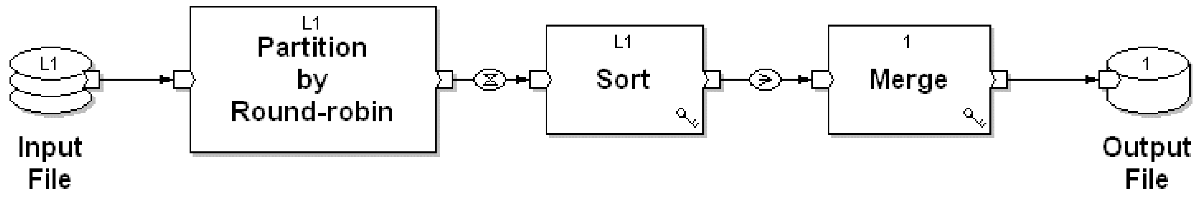


However, that will badly affect parallel performance, as successive partitions of the SORT component will have to wait for the CONCATENATE to consume data from preceding partitions.

Instead, it would be better to address the problem *ab initio* — freshly, from the beginning. Rather than finding splitters and partitioning by range, you can simply partition in round-robin fashion (evenly distributing the data), sort the partitioned data, and then collect the data using a MERGE component (as in **global_sort_serial_simple.mp**).

NOTE: If you create these graphs by modifying **global_sort.mp**, do not forget to change the URL for the output file to **file:$AI_SERIAL/***out_file***.dat**.

global_sort_serial_simple



Finding differences

Finding differences (also known as *differencing* or *diffing*) is useful in various situations:

- **When constructing a graph to reimplement an existing process**, you should test the graph by running it on the same data as the old process, and comparing the results to make sure they are the same.
- **When updating or maintaining a graph**, you should compare the outputs of the new and old versions of the graph.
- **When you have two versions of a dataset that were generated at different times**, you may want to compare them to see which records have changed. For example, you might want to see which addresses have changed between two monthly versions of your master customer file. You could use the results of the graph to determine which rows need to be inserted, deleted, and updated in a database.

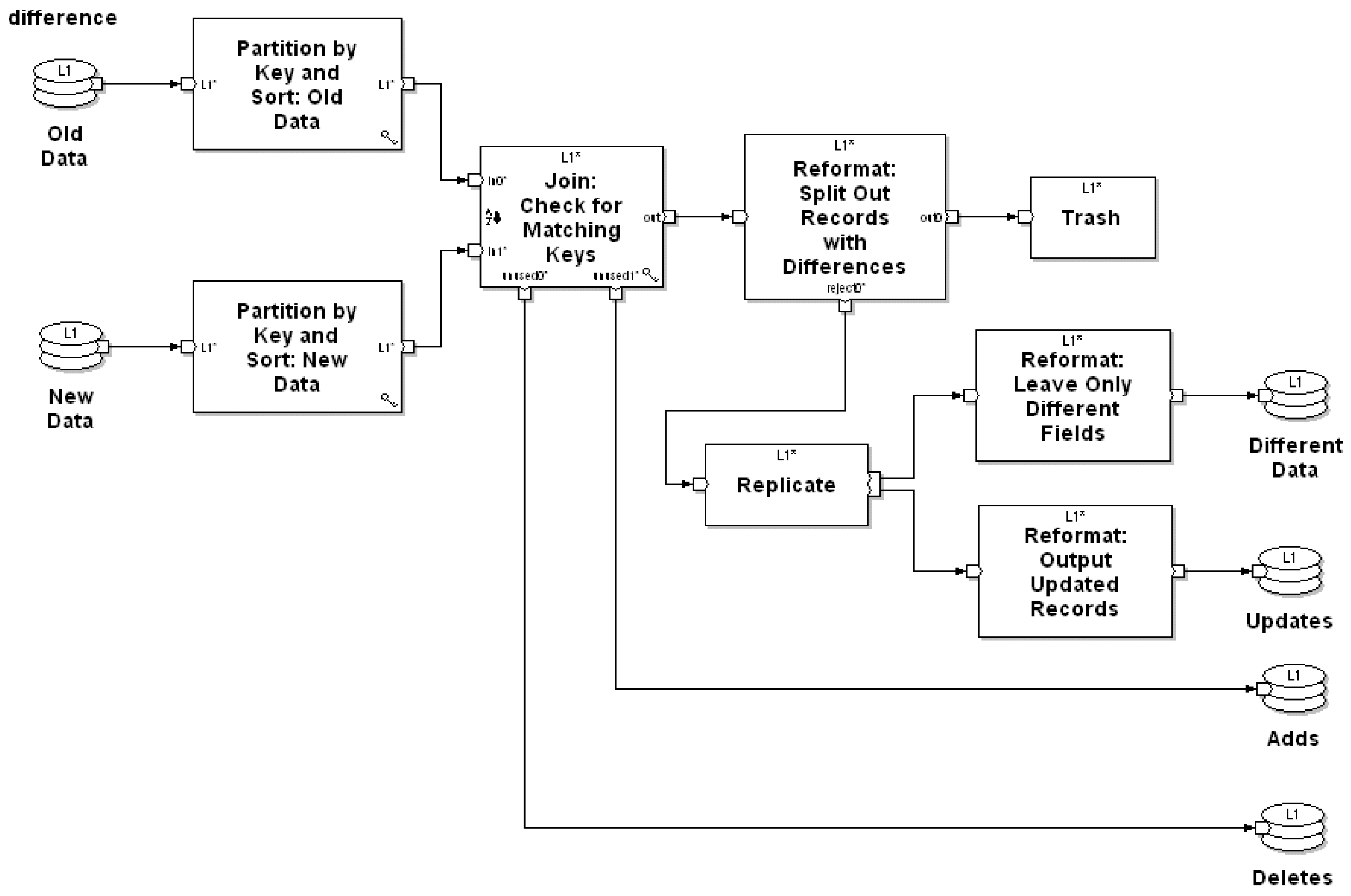
The recipe in this chapter describes how to create a graph that compares two datasets and finds the differences between them. The graph outputs separate datasets containing the following:

- Added records
- Deleted records
- Updated records
- Changed fields

The graph **difference.mp**, shown on page 67 and provided in the **difference** sandbox, finds the differences between two input files. It makes the following assumptions about the input datasets:

- Order and partitioning are unimportant. The two datasets are considered the same if they contain the same records in any order and in any partition.
- Each record has one or more key fields, and each key is unique within the dataset.

For a generic version of a differencing graph in which transform functions and an output record format are automatically generated from input parameters, see “Automated generation of record formats and transforms” (page 79).


difference
L1
Old Data
Partition by Key and Sort: Old Data
L1*
L1*
L1
New Data
Partition by Key and Sort: New Data
L1*
L1*
L1*
Join: Check for Matching Keys
in0*
in1*
out
unused0*
unused1*
L1*
Reformat: Split Out Records with Differences
out0
reject0*
L1*
Trash
L1*
Replicate
L1*
Reformat: Leave Only Different Fields
L1
Different Data
L1*
Reformat: Output Updated Records
L1
Updates
L1
Adds
L1
Deletes

The graph processes the input files and sends records to the outputs as described in the following table:

OUTPUT	RECORDS SENT TO OUTPUT
Trash	Unchanged records (that is, records that are identical in both input files).
Different Data file	Records showing changes between the **Old Data** and **New Data** datasets. Each record includes the key field(s), and an **old**_*field* and **new**_*field* field for each non-key field: • For non-key fields that have *not* changed, these fields are blank. • For non-key fields that *have* changed, the **old**_ *field* and **new**_ *field* fields contain the old and new values of the field.
Updates file	Records from the **New Data** file that have keys matching those of records in the **Old Data** file but have one or more differing non-key fields; in other words, these are updated versions of existing records.
Adds file	Records from the **New Data** file whose keys do not appear in a record in the **Old Data** file.
Deletes file	Records from the **Old Data** file whose keys do not appear in a record in the **New Data** file.

Creating a differencing graph

To create a graph like difference.mp:

1. For the input files containing the old and the new data, make sure that all records with the same key are in the same partition and are sorted within their partitions.

   To do this, connect a PARTITION BY KEY AND SORT component to each input file, and set its key appropriately.

For **difference.mp**, the record format of the input files is as follows:

```
record
  date("YYYYMMDD") date_key;
  string(8) str_key;
  decimal(8) dec_field;
  little endian integer(8) int_field;
  string("\n") str_field;
end
```

The key is formed by the first two fields:

```
{date_key; str_key};
```

2. To determine which records have been added or deleted, and to collect the records that exist in both the old and new datasets, use a JOIN component.

   Connect the JOIN component to the two PARTITION BY KEY AND SORT components, name it appropriately (in **difference.mp**, this is **Check for Matching Keys**), and configure it as follows:

   a. Set the **join_type** parameter to **Inner Join (matching records required on all inputs)**.

   b. Set the **key** parameter to the same value as the **key** for the PARTITION BY KEY AND SORT components you used in Step **1**.

   c. To collect records that have been deleted (records that are present in the old dataset but not in the new one), connect the **unused0** port to an output dataset, and name the dataset appropriately (in **difference.mp**, this is the **Deletes** dataset).

   d. To collect records that have been added (records that are present in the new dataset but not in the old one), connect the **unused1** port to an output dataset, and name the dataset appropriately (in **difference.mp**, this is the **Adds** dataset).

      If necessary, choose **View > Optional Ports** to show all optional ports. Then connect the **unused0** and **unused1** ports to the output datasets, and deselect **Optional Ports** to hide the unneeded ports.

e. For the **out** port, create an output record format that includes a field for both the old and the new values of each field, so that you can later compare the values to see which ones have changed. In the **difference.mp** graph, this is **difference_both.dml**:

```
record
  date("YYYYMMDD") date_key;
  string(8)str_key;
  decimal('')  old_dec_field;
  decimal('')  new_dec_field;
  decimal('')  old_int_field;
  decimal('')  new_int_field;
  string("\n") old_str_field;
  string("\n") new_str_field;
  string(1)    newline = "\n";
end;
```

f. To collect records that exist in both the old and the new datasets, edit the **transform** parameter such that it writes all matching old and new fields from each input file to the **out** port. The **difference.mp** graph does this by using the file **combine.xfr**, which contains the following transform:

```
out :: join(old, new) =
begin
  out.date_key   :: old.date_key;
  out.str_key    :: old.str_key;
  out.old_dec_field :: old.dec_field;
  out.new_dec_field :: new.dec_field;
  out.old_int_field :: old.int_field;
  out.new_int_field :: new.int_field;
  out.old_str_field :: old.str_field;
  out.new_str_field :: new.str_field;
end;
```

3. To find the differences among records that exist in both the old and the new datasets, use a REFORMAT component.

Connect the REFORMAT component to the JOIN from Step **2**, name it appropriately (in **difference.mp**, this is **Split Out Records with Differences**), and configure it as follows:

a. Set **reject_threshold** to **Never abort**.

b. Create a transform that compares the new and the old value for each field, sends records that do not have differences to a TRASH component, and rejects records that do have differences. (You will use the "rejected" records to generate two of the output datasets.)

The **difference.mp** graph does this by using the transform **diff_records.xfr**:

```
out :: reformat(in) =
begin
  out.date_key :: in.date_key;
  out.str_key  :: in.str_key;
  out.dec_field :: if (in.old_dec_field == in.new_dec_field)
                     in.old_dec_field;
  out.int_field :: if (in.old_int_field == in.new_int_field)
                     in.old_int_field;
  out.str_field :: if (in.old_str_field == in.new_str_field)
                     in.old_str_field;
end;
```

NOTE: This transform works because the fields in the output record format (specified in **difference_input.dml**) do not have default values. For fields *with* default values, you would have to use something like the following to ensure that records with differences would be sent to the **reject** port:

```
out.dec_field ::
  if (in.old_dec_field == in.new_dec_field)
    in.old_dec_field
  else force_error("Old and new values are different");
```

4. Replicate the flow from the **reject** port of the **Split Out Records with Differences** component (these are the records with differences).
5. Use one REFORMAT component to output a dataset containing the differences between the old and the new data.

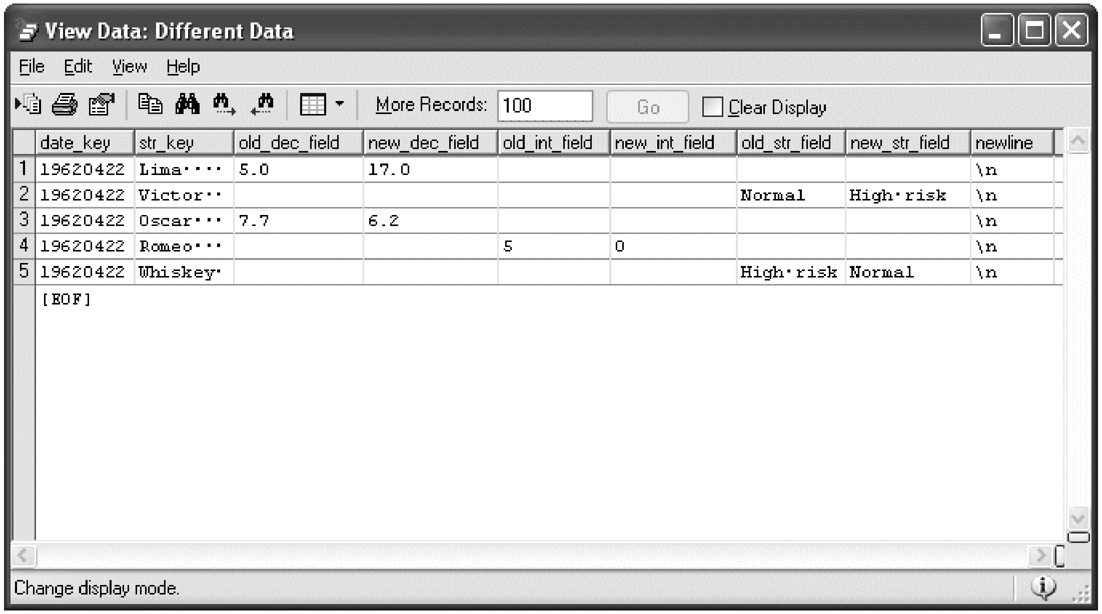
   Name the REFORMAT and the output dataset appropriately and code the transform of the REFORMAT as shown below. (In **difference.mp**, the REFORMAT is **Leave Only Different Fields**, the output dataset is **Different data**, and the transform is **diff_fields.xfr**.)

   a. Output the old and new values of the key fields.

   b. Compare the old and new values of the other key fields. If they are different, output each value. If they are the same, output a blank field.

```
out :: reformat(in) =
begin
  out.date_key  :: in.date_key;
  out.str_key   :: in.str_key;
  out.old_dec_field :: if (in.old_dec_field != in.new_dec_field)
                          in.old_dec_field else "";
  out.new_dec_field :: if (in.old_dec_field != in.new_dec_field)
                          in.new_dec_field else "";
  out.old_int_field :: if (in.old_int_field != in.new_int_field)
                          in.old_int_field else "";
  out.new_int_field :: if (in.old_int_field != in.new_int_field)
                          in.new_int_field else "";
  out.old_str_field :: if (in.old_str_field != in.new_str_field)
                          in.old_str_field else "";
  out.new_str_field :: if (in.old_str_field != in.new_str_field)
                          in.new_str_field else "";
end;
```

   The above transform sends the following data to the output file **Different Data**:



	date_key	str_key	old_dec_field	new_dec_field	old_int_field	new_int_field	old_str_field	new_str_field	newline
1	19620422	Lima····	5.0	17.0					\n
2	19620422	Victor··					Normal	High·risk	\n
3	19620422	Oscar···	7.7	6.2					\n
4	19620422	Romeo···			5	0			\n
5	19620422	Whiskey·					High·risk	Normal	\n
	[EOF]								

6. Use another REFORMAT component to output a dataset containing the updated records. Name the REFORMAT and the output dataset appropriately (in **differences.mp**, they are named **Output Updated Records** and **Updates**) and code the transform of the REFORMAT to assign the new value of each field to the updated records. In **differences.mp**, the transform is **diff_only_new_fields.xfr**:

```
out :: reformat(in) =
begin
  out.date_key  :: in.date_key;
  out.str_key   :: in.str_key;
  out.dec_field :: in.new_dec_field;
  out.int_field :: in.new_int_field;
  out.str_field :: in.new_str_field;
end;
```

Variations on simple differencing

The graph **difference.mp** makes certain assumptions about your goals in comparing two datasets. If your goals are different, you can change the graph as needed. For example, you can:

- **Omit the Different Data output dataset**. Do this if you merely want to find the set of rows to be added, dropped, and updated in a dataset. In this case you can also eliminate the REPLICATE and **Leave Only Different Fields** components, and connect **Output Updated Records** directly to **Split Out Records with Differences** via a straight flow.
- **Use an Output file component instead of a** TRASH **component**. Do this if you want to keep unchanged records rather than discarding them. Simply attach the output dataset to the **Split Out Records with Differences** component.

More complex differencing: Dealing with duplicate key values

The graph **difference.mp** works only for data with unique key values. To use a similar graph to find differences between datasets when one or both of them have duplicate keys, you must add logic to specify which new records correspond to old records. One way to do this is by using a SCAN component to add a uniquely numbered field to each record in a key group, thereby creating a two-field key for each record. For details on how to do this, see “Making unique keys” (page 21).

Advanced differencing: Equality checks

Data-checking graphs must be able to process all records in a dataset without causing errors that stop the graph. This means that they must handle potentially corrupted data, no matter how mangled the contents of a field, and be able to detect the beginning and end of each record. To ensure that your graphs can do this, follow the guidelines in this section.

Safe equality checks

The rules for safe comparison depend on the types of the values being compared.

string and integer types Since all combinations of characters are valid strings, and all sequences of bytes are valid binary integers, you can safely use the non-equal operator (**!=**) to compare **string** and **integer** types. The comparison expression for a field named **f** is simply:

```
in0.f != in1.f
```

decimal, date, and datetime types If you think the quality of your data is questionable, check the validity of **decimal**, **date**, and **datetime** values before comparing them. Do *not* simply cast them to strings before comparing them, as values that are equal in their original types may have different representations as strings. For example, **5** and **5.0** are equal as decimals, and **22APR1962** and **22apr1962** are equal as dates, but neither of these pairs are equivalent as strings. Then do one of the following:

- If they are valid, compare them as their actual types.
- If they are invalid, cast them to strings of the appropriate length, and then compare them as strings.

For a field **f** of type **decimal(",")**, use the following comparison expression:

```
if (is_valid(in0.f) && is_valid(in1.f))
  in0.f != in1.f
else
  (string(","))in0.f != (string(","))in1.f
```

real types You can safely compare the values of **real** types with the non-equal operator, in the sense that this operation never produces an error. The expression **in0.f != in1.f**, for example, never rejects a record.

Note, however, that the IEEE representation accommodates a value that is not a number (**NaN**). Such values always compare as unequal. If you think your dataset may contain **NaN**s, you should take special care to treat all **NaN**s as equal to one another by using the **math_isnan** function in a comparison expression like this:

```
if (math_isnan(in0.f) && math_isnan(in1.f))
  0
else
  in0.f != in1.f
```

void types You cannot directly compare the values of **void** types. Instead, reinterpret them as strings and then compare the strings. To compare two fields of type **void(100)**, use an expression like this:

```
reinterpret_as(string(100), in0.f)
  != reinterpret_as(string(100), in1.f)
```

vector types You must compare values of type **vector** element by element, using the appropriate rule for the vector's element type. You may want to define a function like the one shown below for comparing vectors. It first declares variables to hold the result of the comparison and the iterator for the **for** loop. Then it compares the lengths of the vectors, and finally the individual elements.

```
out::vectors_unequal(in0, in1) =
begin
  let int result = 0;
  let int i;
  if (length_of(in0) != length_of(in1))
    result = 1;
  else
    for (i, i < length_of(in0) and result == 0)
      if (in0[i] != in1[i])
        result = 1;
  out :: result;
end;
```

This function works for vectors of data types that can be compared using the non-equal operator (**!=**). For vectors of other types — such as **real**s, subrecords, and vectors — you may need to replace this expression:

```
in0[i] != in1[i]
```

with a better test, as described elsewhere in this section.

Then use a comparison expression like this:

```
vectors_unequal(in0.f, in1.f)
```

record types Compare the values of **record** types field by field, using the appropriate rules for the type.

union types Compare the values of **union** types by comparing their largest members.

Fuzzy equality checks

Because floating-point numbers are an imprecise representation of real numbers, some operations can produce surprising results that do not always agree with the usual rules of arithmetic. For example, the result of adding floating-point numbers depends on the order in which the numbers are added: **X+Y+Z** may be slightly different from **Z+X+Y**.

Typically the differences are small enough to ignore, and using the floating-point representation is acceptable. However, when comparing the output of two processes that use floating-point numbers, you should be prepared to accept a certain level of difference: the output of the two processes may differ slightly, without either being "wrong."

To compare records produced by processes that use floating-point arithmetic, replace this comparison expression:

```
in0.f != in1.f
```

with this:

```
math_abs(in0.f - in1.f) > 1e-10
```

Here the tolerance has been arbitrarily chosen to be **.0000000001** (or 10-10). Setting the tolerance for a comparison requires consideration of business rules (what is too small to matter?) and the details of floating-point arithmetic (how close can you expect the values to be?). You may also find it useful to create a function that defines the tolerance for a large collection of graphs or projects.

Automated generation of record formats and transforms

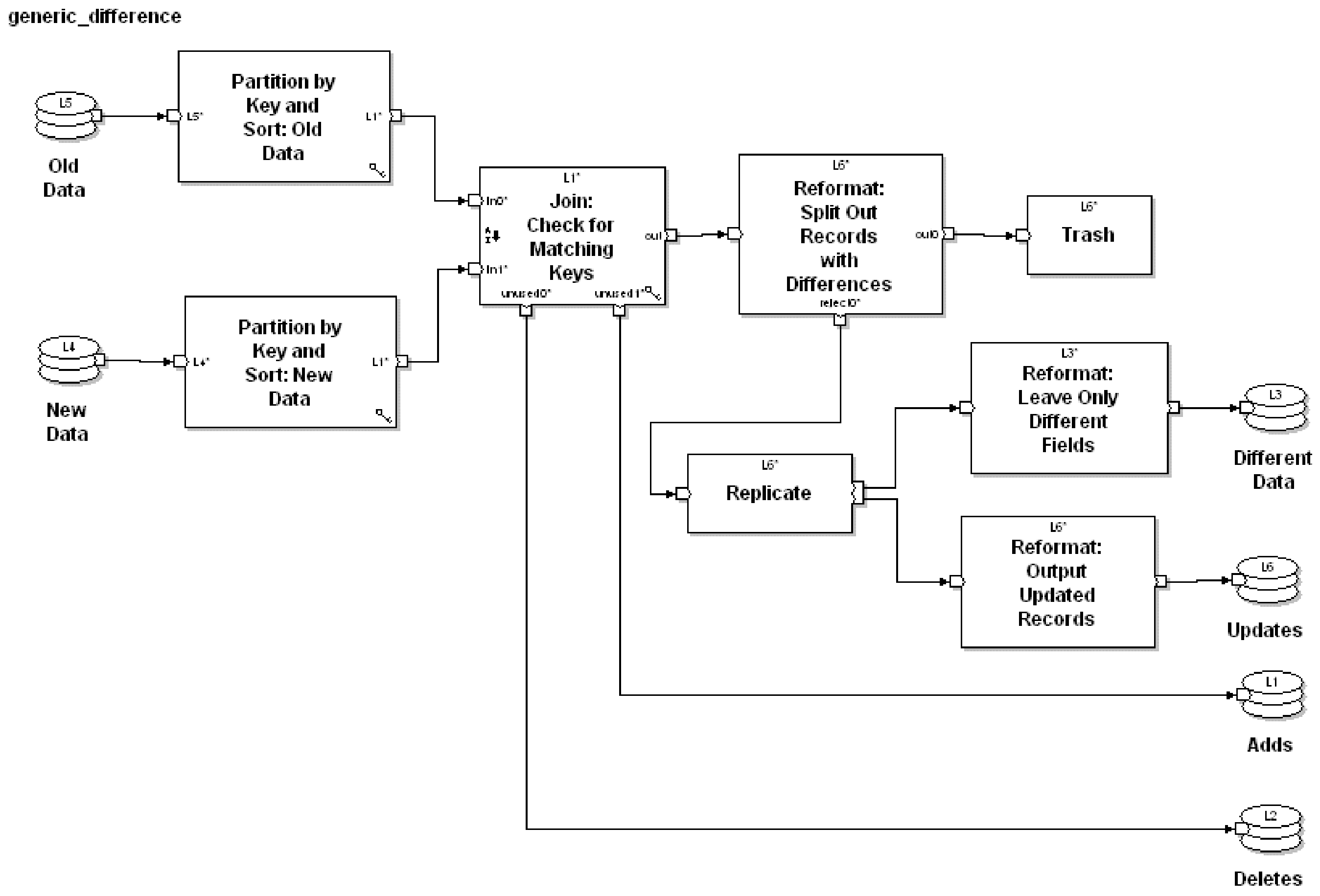
The recipes in this chapter describe metaprogramming techniques for automating the creation of record formats and transforms. They present a parameterized “metadata generation” graph (**make_difference_metadata.mp**, in the **gen-diff** sandbox) that generates transforms and a record format for finding and showing the differences between two input datasets. The transforms and record format it generates serve the same purpose as those in **difference.mp** (described in “Finding differences” on page 65) but are built automatically from a specified input record format. The outputs of **make_difference_metadata.mp** are used by **generic_difference.mp**.

Generated record formats and transforms can be useful in all sorts of other cases. For example:

- Data validation frequently requires applying one or more tests to all fields in each record and attaching to each unacceptable field an identification of which test(s) failed. To automate the generation of a transform and an output record format for such validation, you could create a graph that produces them from an input record format and a list of rules.
- Developers who receive field names and types in formats other than DML files (for example, data dictionaries in the form of spreadsheets) could create a graph to convert such information into DML files.

Generic differencing graph

Following is a review of the most important components in **difference.mp** (presented in "Finding differences" on page 65) and **generic_difference.mp** (page 81), and a summary of how the metadata generation graph (page 84) creates generic transforms like the ones used in those components.


generic_difference
Old Data
Partition by Key and Sort: Old Data
New Data
Partition by Key and Sort: New Data
Join: Check for Matching Keys
Reformat: Split Out Records with Differences
Trash
Replicate
Reformat: Leave Only Different Fields
Different Data
Reformat: Output Updated Records
Updates
Adds
Deletes

The bulk of the metadata generation graph's work is done by its REFORMAT component, which also contains a transform that creates the DML file needed for the output of the differencing graph. (For details, see "Finding differences" on page 65.)

- **Join: Check for Matching Keys** — Determines which records have been deleted or added, and collects the records that exist in both the old and the new datasets. For its **out** port, it uses a record format that includes a field for both the old and new values, each prefixed by **old_** or **new_** (except key fields and "ignored" fields).

  The REFORMAT component in the metadata generation graph creates a transform that checks for key fields and ignored fields and leaves their names unchanged, while for all other fields, it adds the prefix **old_** or **new_**, based on whether the record came from the old or new dataset.

- **Reformat: Split Out Records with Differences** — Finds records that are different between the old and new datasets by comparing the old and new value for each field.

  The REFORMAT component in the metadata generation graph creates a transform that checks for key fields and ignored fields and leaves their names unchanged, while for all other fields it checks whether the old value and the new value are the same, and if they are, it assigns the **old_** value to each **out** field. This causes records with differences to be sent to the **reject** port.

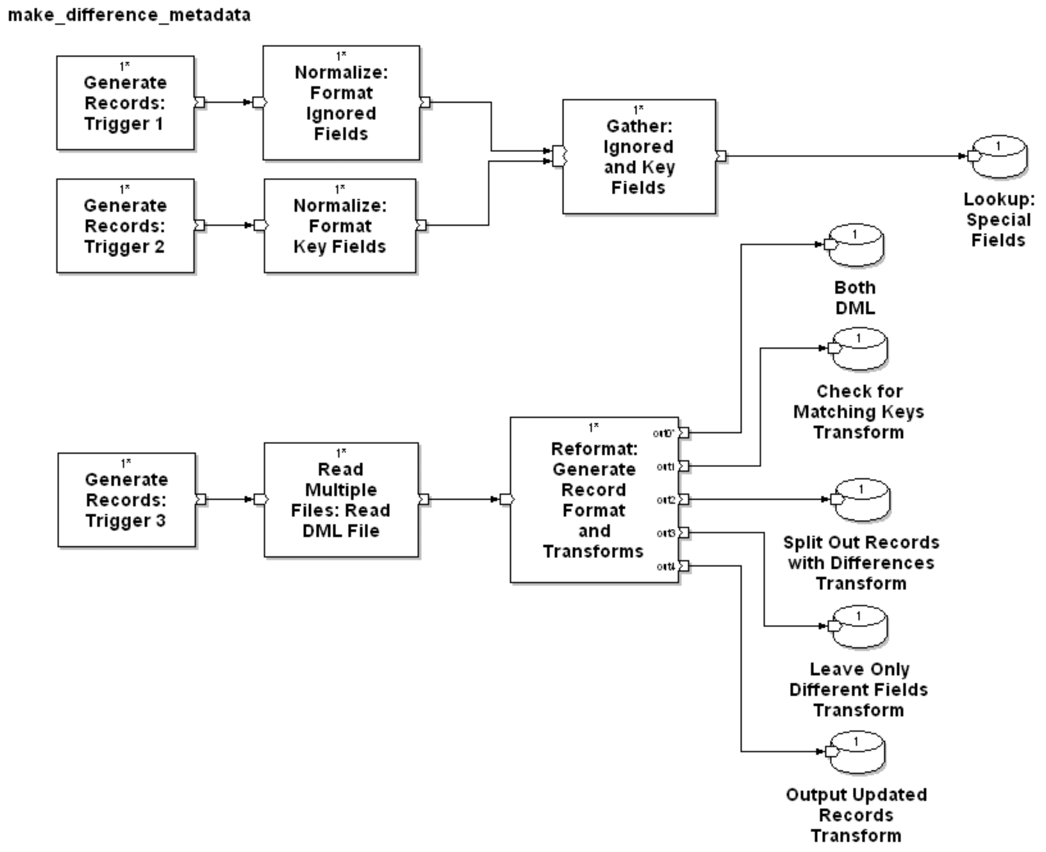
- **Reformat: Leave Only Different Fields** — Outputs a dataset containing the differences between the old and the new data. For records that are different, each value is in a field whose name is appropriately prefixed with **old_** or **new_**. For records that are the same, those fields are blank. Key fields and the **newline** field are left unchanged, while for any other ignored fields, an empty string is output.

  The REFORMAT component in the metadata generation graph creates a transform that first checks for key fields and ignored fields. For key fields and **newline** fields, it assigns the **in** value to the **out** value, while for any other ignored fields, it outputs an empty string. For all other fields, it adds the prefixes **old_** and **new_** to the field names appropriately, compares the old and the new values, and assigns the appropriate strings to the output fields.

- **Reformat: Output Updated Records** — Outputs a dataset containing the updated records, prefixing each **out** field (except key fields and ignored fields) with **new_**.

  The REFORMAT component in the metadata generation graph creates a transform that checks for key fields and ignored fields, leaving their names unchanged. For records that have changed, it outputs a dataset containing the updated records, assigning the new value to each field and adding the prefix **new_** to the field name.

Metadata generation graph


make_difference_metadata
Generate Records: Trigger 1
Normalize: Format Ignored Fields
Generate Records: Trigger 2
Normalize: Format Key Fields
Gather: Ignored and Key Fields
Lookup: Special Fields
Both DML
Check for Matching Keys Transform
Generate Records: Trigger 3
Read Multiple Files: Read DML File
Reformat: Generate Record Format and Transforms
Split Out Records with Differences Transform
Leave Only Different Fields Transform
Output Updated Records Transform

To parse the passed-in lists of key fields and ignored fields (including the **newline** field) into vectors of strings to be stored in the **Special Fields** lookup file, the metadata generation graph uses two NORMALIZE components that call the user-defined function **parse_field_list**. A GATHER component then combines the two flows and outputs them to the lookup file.

DML metaprogramming

The transforms in the REFORMAT component of **make_difference_metadata.mp** make frequent use of several DML metaprogramming functions and the built-in types that support them. The functions produce output strings in the form of DML record formats (type definitions) and transform rules that can then be used in other graphs. The following sections describe how the REFORMAT transforms use string manipulation and more complex logic to produce the record formats and transforms.

Using DML to generate record formats

The first transform in the REFORMAT component takes a string containing information about the record format of the input files (**Old Data** and **New Data**) and generates from it a DML file with fields for both the old and the new values of each record — as follows:

1. Declares variables to hold the attributes of each field in a record (**info** vector), the number of elements in **info**, the differences between fields, the name and DML type of each field in the record, and a loop counter:

```
out :: reformat(in) =
begin
  let dml_field_info_vec info = record_info(in);
  let int n_fields = length_of(info);
  let dml_field_info_vec diff_info = [vector];
  let string('') name;
  let string('') dml_type;
  let int i;
```

2. Loops through each element in the **dml_field_info_vec** and does the following:

   a. Gets the name of each field in the vector.

   b. Sets **dml_type** appropriately.

record_info, length_of, AND RELATED TYPES

The DML function **record_info** returns a vector of the attributes of each field from the passed-in record information. The built-in DML type **dml_field_info_vec** contains a **dml_field_info** record for each field, as shown below. (For more information, see Ab Initio Help.)

```
type dml_field_info =
  record
    string(integer(4))
      name;
    string(integer(4))
      dml_type;
    string(integer(4))
      default;
    decimal(1)
      nullable;
    decimal(1)
      nullflag;
    string(integer(4))
      condition;
    string(integer(4))
      form;
  end;
```

The **length_of** function, when passed a value of type **dml_field_info_vec**, returns the number of elements in the vector.

make_field AND add_fields

The DML function **make_field** creates a **dml_field_info** record from strings containing field information.

The DML function **add_fields** adds a new field definition to the text of a **record** or **union** definition. In the transforms of the REFORMAT component of **make_difference_metadata.mp**, it adds the content of the **diff_info** vector to a record definition. (For more information, see Ab Initio Help.)

c. Checks whether **name** is in the **Special Fields** lookup file, which contains (parameterized) key fields and ignored fields.

- If it is, the transform simply appends the record info for that field to the **diff_info** vector.
- If it is not, the transform creates a **diff_info** vector of differences by adding the strings **new_** and **old_** to the **name** field, using the **dml_type** that was set earlier, and taking the **default** and **nullflag** attributes from the **dml_field_info_vec** unchanged.

```
for (i, i < n_fields)
  begin
    name = info[i].name;

    if (info[i].form member [vector 'integer', 'real',
          'decimal'])
      dml_type = 'decimal("")';
    else
      dml_type = info[i].dml_type;

    if (lookup_match("Lookup: Special Fields", name))
      diff_info = vector_append(diff_info, info[i]);
    else
      begin
        diff_info = vector_append(diff_info,
                    make_field(string_concat('new_', name),
                    dml_type, info[i].default, '',
                    info[i].nullflag));
        diff_info = vector_append(diff_info,
                    make_field(string_concat('old_', name),
                    dml_type, info[i].default, '',
                    info[i].nullflag));
      end
  end
```

NOTE: As the assignment of **dml_type** above shows, you can write code that does more than simple string manipulation to generate a record format; here the type is set to **decimal** if the **form** attribute of the input **dml_field_info** record is **integer**, **real**, or **decimal**.

3. Calls **add_fields** to add the vector of differences to the type definition:

```
    out :: add_fields('record end', diff_info);
  end;
```

Using DML to generate transforms

The last four transforms in the REFORMAT component use the **dml_field_info_vec** returned by **record_info** in a fashion very similar to the first one (**transform0**), but they generate transforms rather than a record format.

Three of these transforms (**transform1**, **transform2**, and **transform4**) differ from **transform0** in the following ways:

- They declare a variable of type **dml_rule_vec** to hold transform rule definitions (rather than a variable of type **dml_field_info_vec** to hold field attributes):

  ```
  let dml_rule_vec rules = [vector];
  ```

- They call **make_rule** to create a transform rule definition (rather than **make_field** to create a **dml_field_info** record):

  ```
  rules = vector_append(rules, make_rule(string_concat(...
  ```

- They call **add_rules** to add a vector of rule definitions to the transform definition (rather than **add_fields** to add a vector of differences to a type definition):

  ```
  out :: add_rules('out :: ...= begin end;', rules);
  ```

For more information on these functions, see Ab Initio Help.

dml_rule_vec, make_rule, AND add_rules

The DML type **dml_rule_vec** is a vector of DML rule definitions of the following form:

```
type dml_rule =
  record
    string(integer(4))
      lhs;
    string(integer(4))
      rhs;
    decimal(',')
      priority;
  end;
```

where **lhs** is the left-hand side of the rule, **rhs** is the right-hand side, and **priority** is the rule's priority.

The three REFORMAT transforms described above use very simple logic to handle fields in the **Special Fields** lookup file whose **field_kind** is **key** or **ignored**. They call **lookup_match** for each field in the **dml_field_info_vec** to check whether the field name exists in the lookup file. If it does, each transform takes the same action in all cases, regardless of the **field_kind** attribute or the **name** of the field.

However, **transform3** demonstrates that you can automate more complex operations: it calls **lookup** to get the **field_kind** attribute for each field from the **Special Fields** lookup table and to check whether the field's **name** is **newline**. For fields whose **field_kind** is **key** or whose **name** is **newline**, it adds the prefix **in**, while for any other ignored fields, it outputs a blank string.

```
if (lookup_match("Lookup: Special Fields", name))
  rules = vector_append(rules,
                          make_rule(string_concat('out.', name),
                                    if (lookup("Lookup: Special Fields",
                                        name).field_kind == 'key' or
                                        name == 'newline')
                                      string_concat('in.', name)
                                    else
                                      '""'));
```

Quantiling

Quantiling can be defined as determining the ranges of a value that most evenly divide a set of records into a specified number of groups, and assigning each record to the appropriate group. Business applications frequently use quantiling to rank customers or otherwise divide them into groups for further statistical analysis. *Deciling* is a specific form of quantiling in which the records are divided into 10 groups.

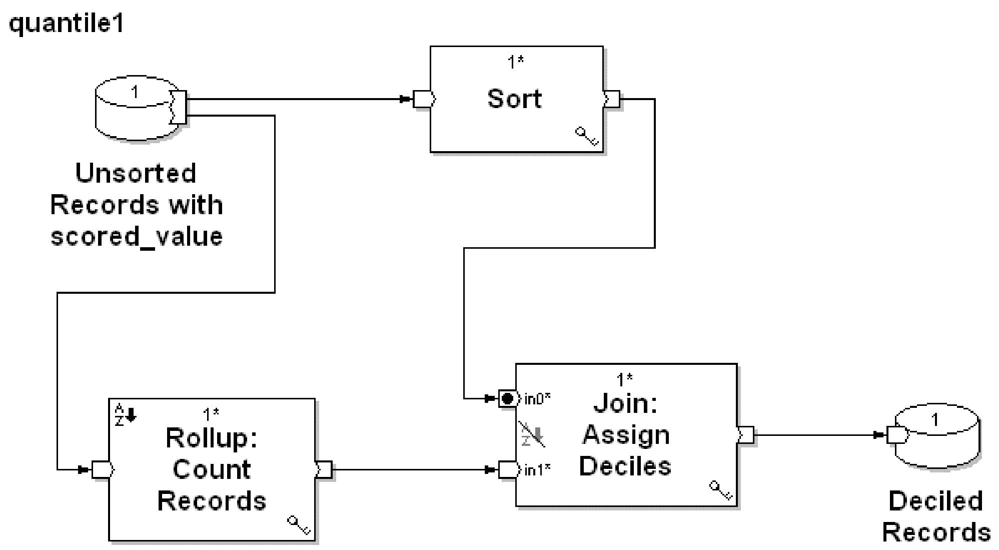
The recipes in this chapter describe how to create graphs that do the following:

- **Simple serial deciling** — Divide records evenly into 10 groups (page 90).
- **Fix decile boundaries** — Ensure that all records with the same scored value are assigned to the same group (page 92).
- **Parallel deciling** — Divide records that are distributed across partitions into 10 groups (page 95).
- **Parameterized parallel quantiling** — Allow users to customize a parallel quantiling graph by specifying values for various parameters (page 103).

The input file used for these graphs consists of 1,000 records that have a **scored_value** field for calculating the quantiles.

Simple serial deciling

This recipe describes how to create a serial graph (**quantile1.mp** in the **quantile** sandbox) that uses the **scored_value** field of the input records to divide them into 10 groups of roughly equal size.



NOTE: This recipe demonstrates one of the key ideas behind the basics of quantiling but is flawed, as described later. The recipe "Fixing decile boundaries" (page 92) addresses the flaw.

To create a serial graph like quantile1.mp:

1. Connect a ROLLUP component to your input file and use it to count the records, as follows:

   a. Set **key** to the empty key: **{}**

b. In the ROLLUP's transform, call **count(1)** to count the records:

```
out::rollup(in) =
begin
  out.count:: count(1);
end;
```

2. In a separate flow, add a SORT component that sorts the input file by **scored_value**.

3. Connect the ROLLUP and the SORT to a JOIN component, and do the following in the JOIN:

a. Set **key** to the empty key: **{}** to match all records from the ROLLUP (in this case only one) with all records from the SORT.

b. Set the parameter **sorted_input** to **In-memory: Inputs need not be sorted**.

c. Calculate the decile for each input record as shown in the following code, and output each **scored_value** with its **decile**:

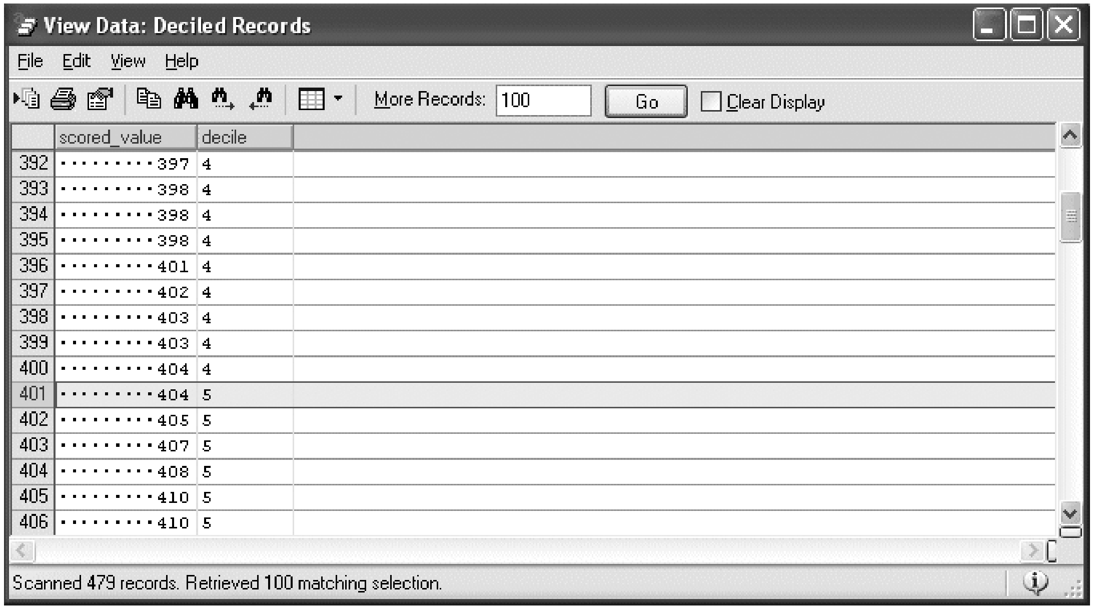
```
let integer(8) record_number = 0;

out::join(in0, in1) =
begin
  record_number = record_number + 1;

  out.* :: in0.*;
  out.decile :: ((record_number * 10 - 1) / in1.count) + 1;
end;
```

This code assigns the correct decile to each record in sequence: **(record_number * 10 – 1)** results in a number from **9** (for the first record) to just under 10 times the number of records (for the last record). That quantity divided by the number of records yields a value between **0** (for the early records) and **9** (for the last records). To label the deciles **1** through **10**, the code adds **1** to that value.

4. Run the graph and examine the output data to find the flaw in this graph:

EMPTY KEY ON JOIN

An empty key on a JOIN causes all records on each port to be matched with all records on all other ports. In this case it is a simple way to match up the single output record from the ROLLUP with every record from the SORT. Another way to do this would be to write the output of the ROLLUP into a LOOKUP FILE and read it with a REFORMAT.



	scored_value	decile
392	·········397	4
393	·········398	4
394	·········398	4
395	·········398	4
396	·········401	4
397	·········402	4
398	·········403	4
399	·········403	4
400	·········404	4
401	·········404	5
402	·········405	5
403	·········407	5
404	·········408	5
405	·········410	5
406	·········410	5

Scanned 479 records. Retrieved 100 matching selection.

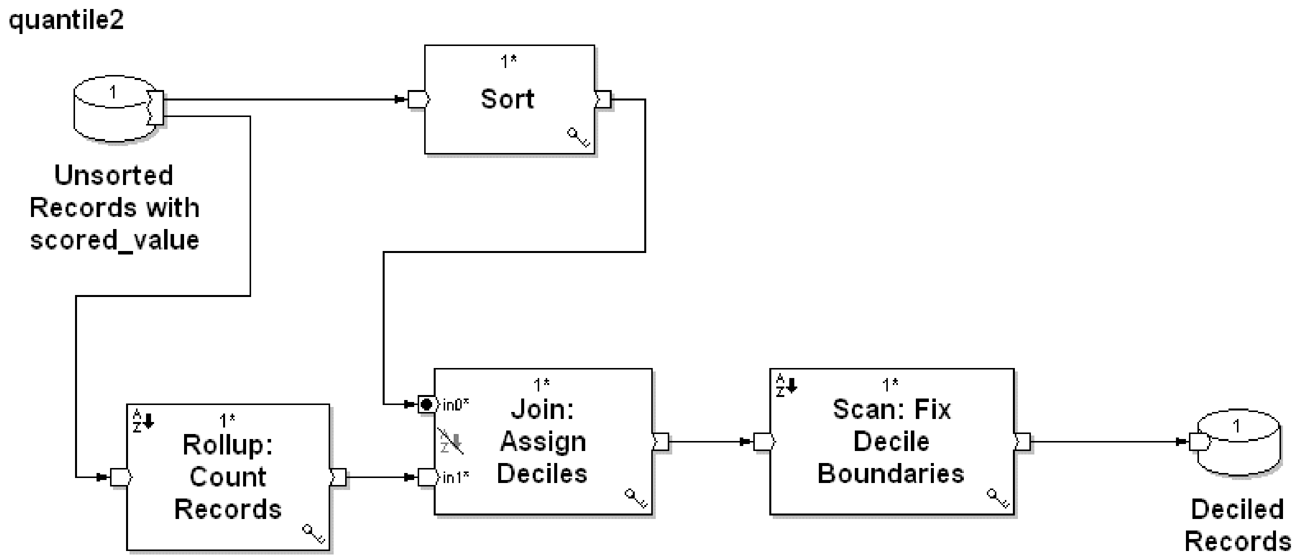
The records are divided into exactly 10 groups of 100 each, but in at least one case, two records with the same **scored_value** fall into two different deciles: Records 400 and 401 both have **scored_value** 404, but the first one is marked as belonging to decile 4 and second one as belonging to decile 5. To fix this problem, follow the steps in the next section.

Fixing decile boundaries

To ensure that all records with the same **scored_value** are assigned to the same decile, the graph needs to “remember” the value of **decile** for records that share the same **scored_value**. The best way to do this is to use a SCAN component as shown in the next figure (**quantile2.mp**). If you code the SCAN component’s functions as described after the figure, they will behave as follows:

- **initialize** — Is called at the beginning of each group of records that have the same key (**scored_value**), and stores (in its **temporary_type** variable) the decile for the first record in each group
- **scan** — "Remembers" the decile value as it processes each record in the group
- **finalize** — Puts the decile value in the **decile** field of every record in the group

By assigning the same decile value to all records in the same key group (specifically, the decile value of the first record in the group), the SCAN fixes any "bad" boundaries caused by the JOIN.



To use a SCAN to fix decile boundaries:

1. In a copy of the graph described in "Simple serial deciling" (page 90) (**quantile1.mp**), insert and connect a SCAN component between the JOIN and **Deciled Records**.

2. Configure the SCAN component as follows:

   a. Specify **scored_value** as the key.

   b. Set the **sorted_input** parameter to **Input must sorted or grouped**.

   c. On the **Ports** tab, select the **Propagate through** checkbox.

3. Double-click the SCAN component and allow the GDE to generate a default transform.

4. In the **Record Format Editor** or in **Text View**, define the **temporary_type** as **integer(4) decile**:

```
type temporary_type =
record
  integer(4) decile;
end
```

5. In the **Transform Editor** or in **Text View**, do the following:

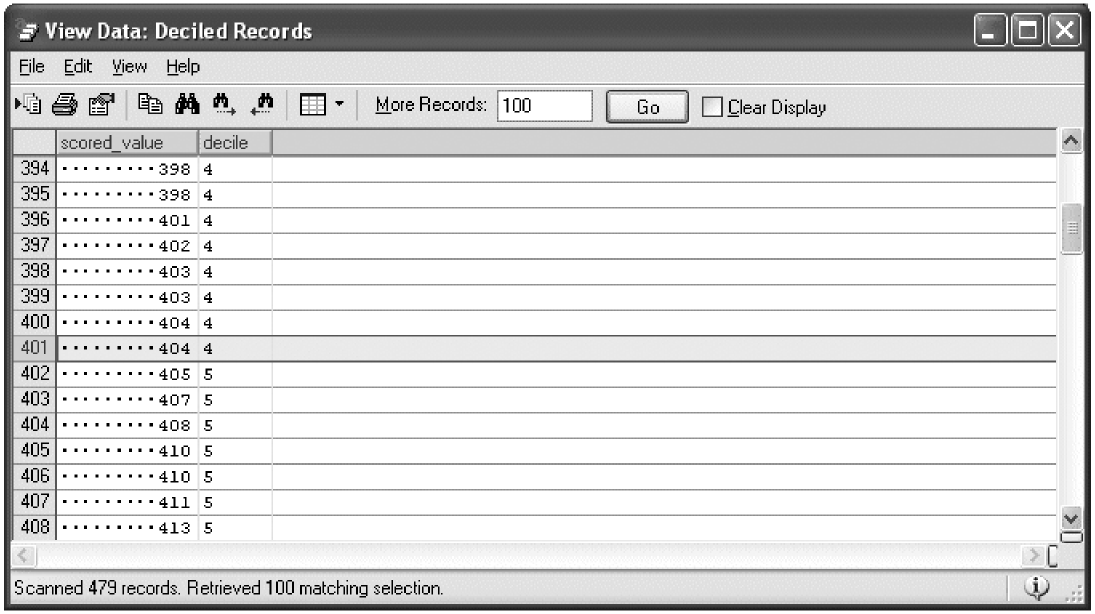
   a. In the **initialize** function, set **temp.decile** to **in.decile**.

   b. In the **scan** function, set **temp.decile** to **temp.decile**.

   c. In the **finalize** function, set **out.decile** to **temp.decile** and **out.scored_value** to **in.scored_value**:

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

temp::scan(temp, in) =
begin
  temp.decile :: temp.decile;
end;

out::finalize(temp, in) =
begin
  out.decile :: temp.decile;
  out.* :: in.*;
end;
```

6. Run the graph and examine the output data:



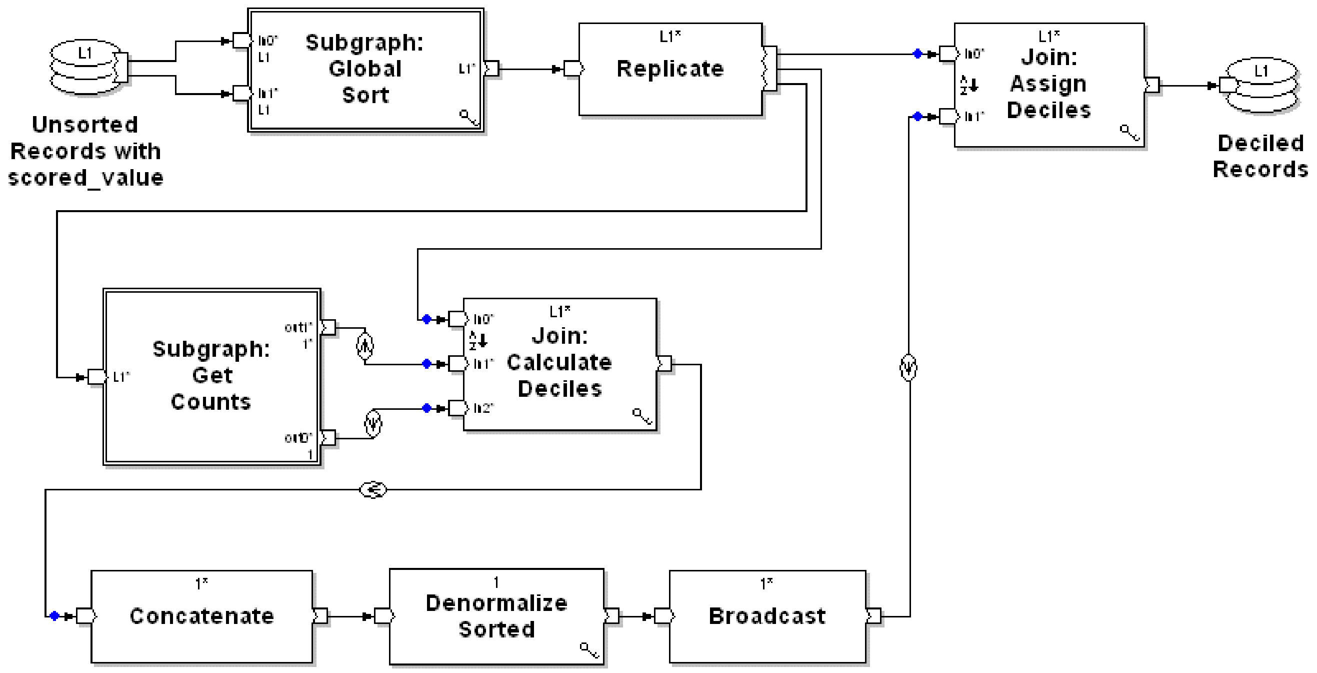
	scored_value	decile
394	398	4
395	398	4
396	401	4
397	402	4
398	403	4
399	403	4
400	404	4
401	404	4
402	405	5
403	407	5
404	408	5
405	410	5
406	410	5
407	411	5
408	413	5

Scanned 479 records. Retrieved 100 matching selection.

Parallel deciling

The above descriptions of deciling are sufficient for serial data. However, if you are processing large amounts of data in parallel, the records that belong in each decile will probably be distributed across partitions, so you must take the steps summarized on page 96 to find and fix the decile boundaries in parallel (see **quantile3.mp**).

quantile3



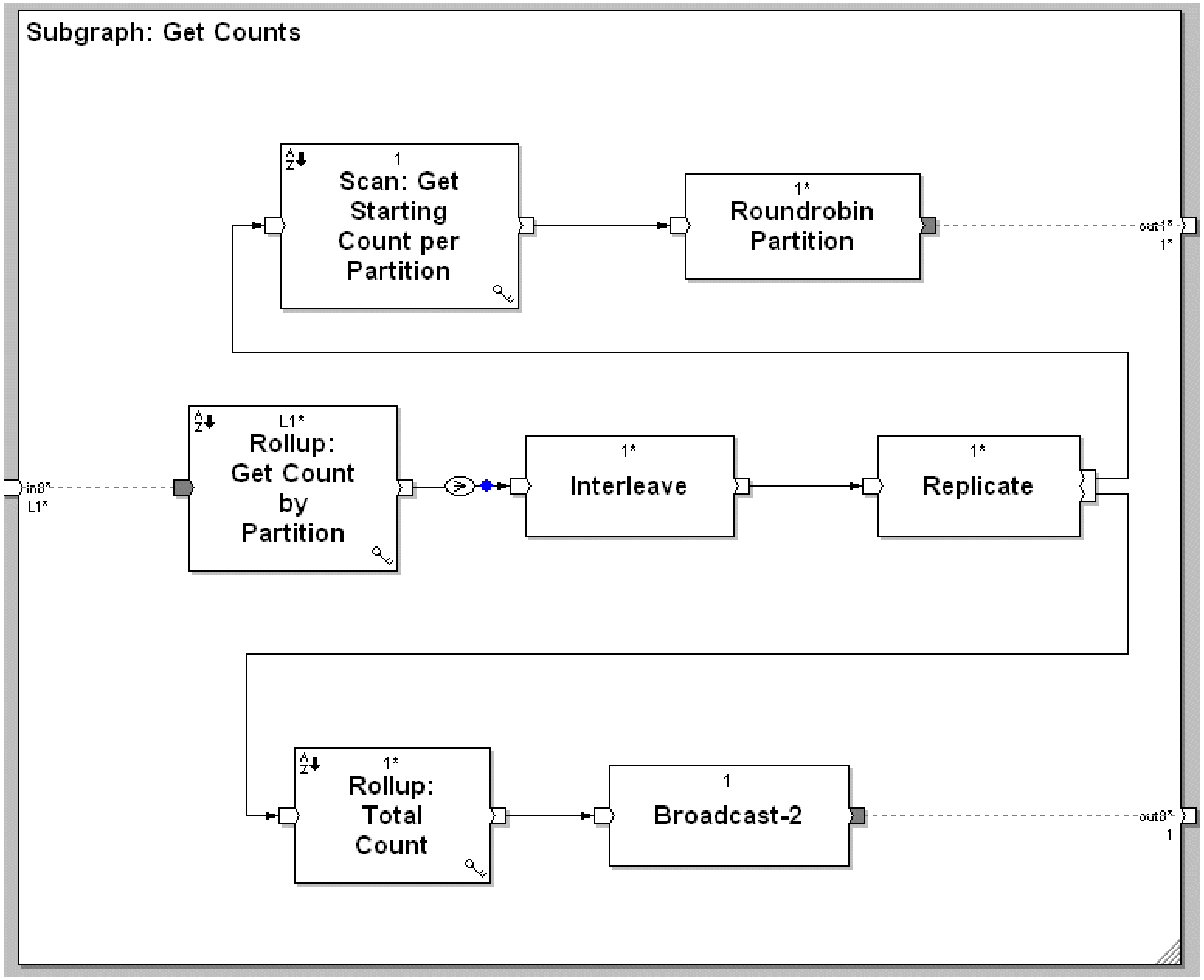
▶ **To find and fix the decile boundaries in parallel:**

1. Sort the data globally, so that the data is sorted *within* each partition and *across* partitions, with the smallest values in the first partition and the largest values in the last. (For details on how to do this, see "Global sort" on page 57.)
2. Replicate the resulting data so that you can use it for further processing and in the final JOIN.
3. Get the total record counts and starting record counts for each partition — so that when you calculate and assign the deciles, you know the relative position of any given record in the entire set (page 97).

4. Calculate the splitters (the ending decile values) and fix the boundaries — that is, ensure that all records with the same **scored_value** will be assigned to the same decile, even though they could be in different partitions (page 101).
5. Prepare the decile splitters for the final JOIN — concatenate them, create a vector of their values, and broadcast the vector (page 102).
6. Join the globally sorted data with the vector of splitters to assign the deciles and output the data (page 102).

Getting the total record counts and starting record counts for each partition

To calculate the deciles properly, you must calculate the total count of records across all partitions, as well as the starting count for each partition (that is, the total count of records in the earlier partitions). You can do this by creating a subgraph similar to **Get Counts** in **quantile3.mp** as described after the following figure, and connecting it to one of the **out** ports of the REPLICATE.


Subgraph: Get Counts
1
Scan: Get Starting Count per Partition
1*
Roundrobin Partition
out1*
1*
in0*
L1*
L1*
Rollup: Get Count by Partition
1*
Interleave
1*
Replicate
1*
Rollup: Total Count
1
Broadcast-2
out0*
1

To create a subgraph similar to Get Counts in quantile3.mp:

1. Calculate the total count for each partition by using a ROLLUP component with its **key** set to the empty key: **{}**
2. Reduce the counts from each partition to a single, ordered, serial flow and make a copy of it by using an INTERLEAVE and a REPLICATE.
3. Compute a grand total of the counts from the partitions by using a ROLLUP with its **key** set to the empty key: **{}** and broadcast it to all partitions.
4. Compute the starting count for each partition and send the counts to their proper partitions by using a SCAN and then a PARTITION BY ROUND-ROBIN. Code the SCAN transform as follows (see the code in Step **c** below):
   a. Initialize its **temporary_type** to zero.
   b. Read the stream of total counts for each partition (from the INTERLEAVE) and add each one to the **temporary_type** to get the ending count of each partition.
   c. Subtract each total count from the **temporary_type** to get the starting count of each partition.

```
type temporary_type = decimal(12);

temp::initialize(in) =
begin
  temp :: 0;
end;

temp::scan(temp, in) =
begin
  temp :: temp + in.count;
end;

out::finalize(temp, in) =
begin
  out.starting_count :: temp - in.count;
end;
```

USING WATCHERS

Watchers are saved with a graph unless you explicitly delete them by right-clicking the watcher and deselecting **Watcher on Flow**, or by choosing **Debugger > Remove All Watchers**. To see the watchers saved with a graph, enable the debugger by choosing **Debugger > Enable Debugger** or clicking the Enable icon on the Debugging toolbar. When the debugger is disabled, the watchers will not affect execution of the graph in any way.

This transform calculates a running count minus the current count, and yields the following values:

in.count	temp	out.starting_count
42	42	0
221	263	42
95	358	263
194	552	358
448	1000	552

5. Optionally (if you would like to be able to view the data generated by the INTERLEAVE and SCAN components), choose **Debugger > Enable Debugger**, right-click the **out** flows for those components, and click **Watcher on Flow**. After running the graph, right-click each Watcher icon and choose **View Data** to see the values.

- INTERLEAVE watcher values for **count**:

```
 42
221
 95
194
448
```

- SCAN watcher values for **count** (starting count per partition):

```
  0
 42
263
358
552
```

Calculating the splitters and fixing the boundaries across partitions

To calculate the deciles to be used as splitters in the final JOIN, you can use the same logic as in "Simple serial deciling" (page 90), with minor modifications. Instead of simply using **record_number**, you must add in the starting count for each partition (**in1.count + record_number**). Note also that the total count is now on **in2**.

A more complex task is fixing the boundaries to ensure that all records with the same **scored_value** will be assigned to the same decile, even though they could be in different partitions. You must write a JOIN transform that outputs a record *only* when it is the last record assigned a given decile value. The transform shown below does this by comparing the decile of the current record to that of the next, and putting out a record only when they differ. Here the inputs are the following:

- **in0** is the partitioned, sorted records (from the **Global Sort** subgraph).
- **in1** is the starting counts for each partition (from the **Get Counts** subgraph).
- **in2** is the total record count (from the **Get Counts** subgraph).

```
let integer(8) record_number =0;
let integer(4) this_decile = -1;
let integer(4) next_decile =-1;

out::join(in0, in1, in2) =
begin
record_number = record_number + 1;
this_decile = if (record_number != 1)
                next_decile
              else
                (((in1.starting_count + record_number) * 10 - 1)
                 / in2.count) + 1;
next_decile = (((in1.starting_count + record_number) * 10)
               / in2.count) + 1;
```

```
    out.* :: in0.*;
    out.decile :: if (this_decile != next_decile) this_decile;
end;
```

Preparing the decile splitters for the final JOIN

To present the set of decile splitters to the final JOIN as a vector of values, pass the calculated values to the following components:

- CONCATENATE (to collect them in order)
- DENORMALIZE SORTED with its key set to the empty key: **{ }** (to build a vector out of them)
- BROADCAST (to send the vector to each partition of the JOIN)

Assigning the deciles and outputting the data

Finally, you need to assign a decile to each of the original 1,000 records and output the records with their deciles. To do this, add a JOIN component and attach the following to its **in** ports:

- **in0** — a REPLICATE component containing the globally sorted records
- **in1** — a BROADCAST component containing the vector of splitter values for assigning the deciles

Write a transform such as the following to use the **scored_value** in each record to determine the corresponding decile:

```
let integer(4) current_decile = 0;
let decimal(12) next_decile_boundary = 0;

out::join(in0, in1) =
begin
  if (current_decile == 0)
    begin
      next_decile_boundary = in1.splitters[current_decile];
      current_decile = 1;
    end
  while (in0.scored_value > next_decile_boundary)
    begin
      next_decile_boundary = in1.splitters[current_decile];
      current_decile = current_decile + 1;
    end

  out.decile :: current_decile;
  out.* :: in0.*;
end;
```

Parameterized parallel quantiling

This recipe describes how to create a generic version of **quantile3.mp** that uses parameters instead of hardcoded values for the following:

- The name of the field used to calculate the quantiles
- The data type of the field used to calculate the quantiles
- The name of the field to write the quantiles to
- The number of quantiles to use

WHY YOU SHOULD CONNECT FLOWS DIRECTLY TO INPUT FILES INSTEAD OF USING REPLICATE

When multiple components are connected to an INPUT FILE, they each read the data independently, at their own pace. When multiple components are connected to the output of a REPLICATE, there are two potential problems:

- The downstream components may be slowed to the processing rate of the slowest component among them.
- If the downstream components send their outputs, directly or indirectly, to a component in common, deadlock flow buffering will be introduced, possibly causing data to spill to disk.

Here the PARTITION BY RANGE component in the **Global Sort** subgraph cannot start partitioning the records until it reads the splitters from the BROADCAST component. If you use a REPLICATE to pass the input data, the PARTITION BY RANGE will have to buffer the data while waiting for the splitters.

To create this parameterized version of the deciling graph:

NOTE: These steps are described in detail in subsequent sections.

1. Create a quantiling subgraph from the original graph and insert it into a test graph.
2. In the subgraph, add and export parameters for the values you do not want to be hardcoded (page 106).
3. In the subgraph, replace previously hardcoded values with parameters, and change their interpretation appropriately (page 107).
4. Test your quantiling subgraph (page 109).

Creating a quantiling subgraph and inserting it into a test graph

To create a quantiling subgraph from the original graph and insert it into a test graph:

1. Save copies of **quantile3.mp** under two new names such as the following:
   - **my-Subgraph_Parallel_Quantile.mp** — To become the quantiling subgraph
   - **my-quantile4.mp** — To be the test graph for the quantiling subgraph
2. Delete the input and output files and make subgraph inputs and outputs from what were previously their flows, as follows:

   a. Drag two flows from the inputs of the **Global Sort** subgraph component to the left edge of the graph.

   This allows the user of the subgraph to replicate the data explicitly if it is the result of some earlier processing or to hook the two inputs directly to an INPUT FILE, which is more efficient (see the sidebar).

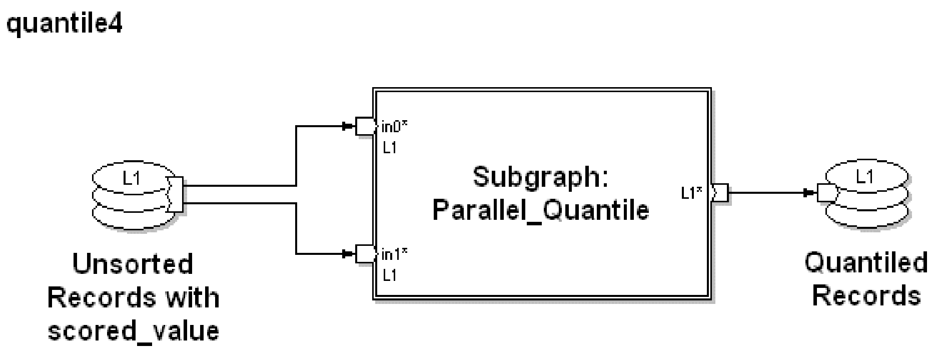
   b. Drag the flow from the output port of the JOIN to the right edge of the graph.
3. Save the graph, ignoring the compilation error message.
4. Switch to **my-quantile4.mp**.

5. Delete everything except the input and output files.
6. Save the graph, ignoring the compilation error message.
7. Choose **Insert > Program Component** and browse to the **mp** directory.

> **NOTE:** In the **Open** dialog, make sure the **Host** button is selected so that you are browsing in **Host** mode (not in **Local** mode).

8. Select **my-Subgraph_Parallel_Quantile.mp**, add **$AI_MP/** before the filename (to ensure that it is included with a sandbox path rather than hardcoded), and click **Open**.
9. Attach the flows from the subgraph to the input and output files.

   The graph should now look similar to **quantile4.mp**:


quantile4
L1
Unsorted Records with scored_value
in0*
L1
in1*
L1
Subgraph: Parallel_Quantile
L1*
L1
Quantiled Records

10. Save **my-quantile4.mp**.

    This graph should run, since you have not made any changes other than converting part of it to a subgraph.

LINKED SUBGRAPHS

The subgraph you inserted in Step **9** is a *linked* subgraph, as you can see from its (read-only) legend when you open it from within **my-quantile4.mp**. This means that the enclosing graph contains a local copy with a *link*, or pointer, to the location of the original subgraph (in this case in **$AI_MP**). This location is displayed after **Source:** on the **Description** tab of the **Properties** dialog for the subgraph.

To edit a linked subgraph:

1. Browse to the location of the subgraph, or right-click the copy in the containing graph and choose **Component Source**.
2. Edit and save the subgraph.

Changes you make to the original subgraph are not automatically updated in graphs that use them. This prevents errors in the subgraph from breaking every graph that uses it.

To update a containing graph with changes made to a linked subgraph:

- In the containing graph, right-click the linked subgraph and choose **Update**. (You can select multiple components and update all of them.)

Adding and exporting parameters

The current version of the shared subgraph contains hardcoded values that reflect the dataset used to test the original deciling graph.

To change the hardcoded values into parameters:

1. Open **my-Subgraph_Parallel_Quantile.mp** (*not* the linked copy of the subgraph contained in the main graph) and choose **Edit > Properties**.
2. Create the four parameters listed below. To do this, click **Create** on the **Parameters** tab of the **Properties** dialog, enter the names, and click **OK**. (The defaults for **Type**, **Scope**, and **Kind** are acceptable.)
   - **SCORED_VALUE** — To represent the name of the field used to calculate the quantiles. (In the test data provided in the sandbox, this is **scored_value**.)
   - **SCORED_VALUE_TYPE** — To represent the data type of the field used to calculate the quantiles. (In the test data provided in the sandbox, this is **decimal(12)**.)
   - **QUANTILE_FIELD_NAME** — To represent the field that the quantile values will be written to. (In the test data provided in the sandbox, this is **decile**.)
   - **MAX_QUANTILES** — To represent the number of quantiles into which the data is to be divided. (In the test data provided in the sandbox, this is **10**.)
3. To export all parameters (except **key**) from the **Global Sort** subgraph so that they will "inherit" their values from the containing graph, **my-Subgraph_Parallel_Quantile.mp**, do the following for each parameter:

   a. On the **Parameters** tab of the **Properties** dialog for the **Global Sort** subgraph, select the parameter and click **Export.**

   b. In the **Exported Name** box of the **Export as Parameter of graph** dialog, type the appropriate name from the list below:

UNEXPORTED PARAMETER	EXPORTED PARAMETER
serial_layout	SERIAL_LAYOUT
parallel_layout	PARALLEL_LAYOUT
SAMPLE_SIZE	SAMPLE_SIZE
MAX_CORE	MAX_CORE

The **my-Subgraph_Parallel_Quantile.mp** subgraph should now have eight parameters.

Replacing hardcoded values with parameters

To make the **my-Subgraph_Parallel_Quantile.mp** subgraph use the four parameters you created in Step **2** of "Adding and exporting parameters" above (rather than the hardcoded values), do the following:

1. Replace the hardcoded values as follows:

   a. Choose **Edit > Parameters**.

   b. On the left pane of the **Edit Parameters** dialog, click the plus signs to expand the component listings for **my-Subgraph_Parallel_Quantile**, **Subgraph_Get_Counts**, and **Subgraph_Global_Sort**.

   NOTE: The grid in the **Edit Parameters** dialog allows global, graph-wide search and replace operations, but only on items that are open in the view.

2. Choose **Edit > Replace**, click **Match Case** (to avoid making replacements in the parameters you just defined), and make the global replacements shown in the table.

   Start each search by selecting the top-level icon in the browser to ensure that no occurrences are missed.

GLOBALLY REPLACE THIS	WITH THIS
`scored_value`	`${SCORED_VALUE}`
`decimal(12)`	`${SCORED_VALUE_TYPE}`

3. Choose **Edit > Replace**, click **Match Case** (again to avoid making replacements in the parameters you just defined), and *where the values shown in the table below are values that you want to parameterize*, replace them as shown:
   - Do *not* replace the value **10** where it appears in the setting for **max_core**.
   - Do replace **decile** when it occurs in **out_metadata**; these are record formats used by the components.
   - Do *not* replace **decile** in instances of **[Same as...]**, **this_decile**, or **next_decile**.

WHERE APPROPRIATE, REPLACE THIS	WITH THIS
`10`	`${MAX_QUANTILES}`
`decile`	`${QUANTILE_FIELD_NAME}`

4. For all the above replacements that occurred in a transform or record format, change the interpretation from **constant** to **${} substitution** as follows:

   a. Choose **Edit > Parameters**.

   b. On the left pane of the **Edit Parameters** dialog, click the plus signs to expand the component listings for **my-Subgraph_Parallel_Quantile**, subgraph **Get_Counts**, and subgraph **Global_Sort**.

   c. Choose **Edit > Find** and search for **${** .

   d. Wherever you find **${** in a transform or record format (for record formats, the **Name** in the middle panel of the **Edit Parameters** dialog will end with **_metadata**), change the interpretation by expanding the **Advanced Attributes** list in the right pane of the **Edit Parameters** dialog (if necessary) and selecting **${} substitution** instead of **constant**.
   e. Save your changes.
5. Save your shared subgraph, ignoring the compilation error message (displayed because the graph has no inputs and therefore cannot propagate record formats and layouts).

Testing your quantiling subgraph

To test the quantiling subgraph:

1. Switch to your test graph (**my-quantile4.mp**).
2. Right-click the **my-Subgraph_Parallel_Quantile** subgraph and click **Update**.

   A yellow to-do cue appears because the four newly defined parameters have no values.
3. Provide the following values:

FOR THIS PARAMETER	SPECIFY THIS VALUE
`${SCORED_VALUE}`	`scored_value`
`${SCORED_VALUE_TYPE}`	`decimal(12)`
`${QUANTILE_FIELD_NAME`	`decile`
`${MAX_QUANTILES}`	`10`

4. Run the test graph.

Auditing

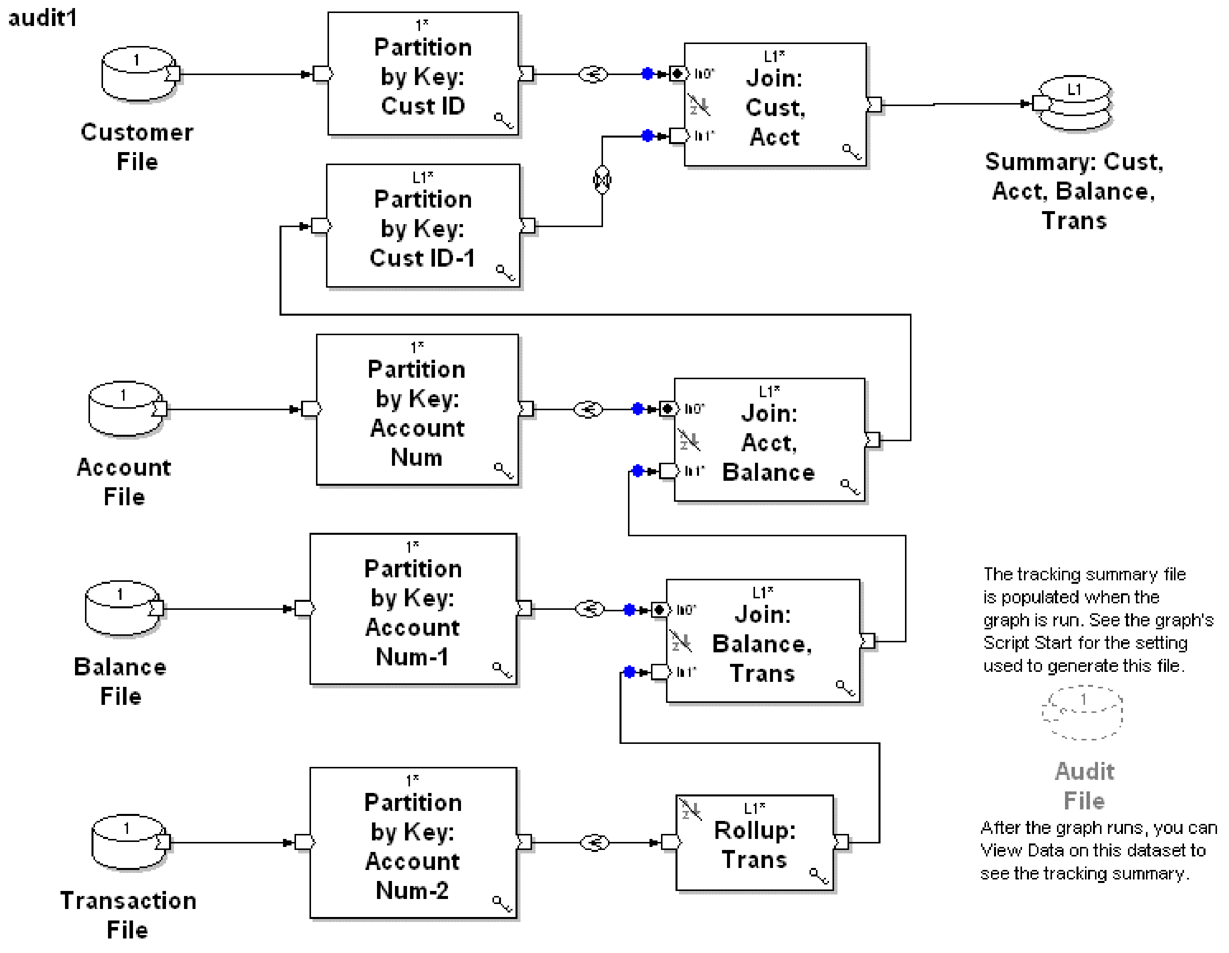
In many situations it is important to record and compare the number of records flowing into and out of a graph. In an accounting system, for example, you may need to ensure that a report contains information on all accounts.

The two recipes in this chapter show how to develop graphs that meet auditing requirements for financial applications. They present two different approaches to auditing the same sample application:

- Using a tracking summary file and a separate graph to ensure that no records have been lost between the inputs and outputs.
- Using log flows to ensure that no records have been lost and that the financial totals are unchanged between the inputs and outputs

Sample application to audit

The following figure shows the graph to be audited (**audit1.mp** in the **audit** sandbox). It is a simplified example of a banking application that creates a summary of the balances, debits, credits, and customer information for each account:


audit1
1
Customer File
1*
Partition by Key: Cust ID
L1*
Partition by Key: Cust ID-1
in0
in1
L1*
Join: Cust, Acct
L1
Summary: Cust, Acct, Balance, Trans
1
Account File
1*
Partition by Key: Account Num
L1*
Join: Acct, Balance
1
Balance File
1*
Partition by Key: Account Num-1
L1*
Join: Balance, Trans
1
Transaction File
1*
Partition by Key: Account Num-2
L1*
Rollup: Trans
1
Audit File

The tracking summary file is populated when the graph is run. See the graph's Script Start for the setting used to generate this file.

After the graph runs, you can View Data on this dataset to see the tracking summary.

Simple auditing: Counting records

This recipe shows how to audit a graph's record counts to make sure that no account records have been lost between the inputs and outputs.

To audit a graph's record counts:

1. Write tracking information to a summary file and examine the file to identify the flow(s) you will need for your audit (see the next section).
2. Create a separate graph that uses a ROLLUP to compare the input and output record counts from the summary file (page 115).

TRACKING DATA

Ab Initio software provides tracking data in four ways:

- Numbers on the flows in the GDE
- Text sent to standard output when you run a graph from the shell
- Values stored in the EME®
- Tracking summary file

The numbers in the first two locations are not readily parsed, and not everyone uses the EME to store tracking results. This recipe describes how to use the summary file.

Writing tracking information to a summary file and finding the flows needed for the audit

By default, the GDE uses the settings on the **Tracking** tab of the **Run > Settings** dialog to set the configuration variable AB_REPORT, which controls the ways a graph's tracking information is reported during a run. However, you can augment the AB_REPORT setting with options that are not visible in that dialog. To do this, choose **Edit > Script > Start** and add a line such as the following to the start script of the graph:

```
export AB_REPORT="$AB_REPORT other_settings"
```

By adding a setting of the form **summary=***path,* you can tell the graph to write the tracking information to the file specified in *path*, in a "summary" format that is easy to read from another graph. (For details, see **AB_REPORT** and **Generating a summary file** in Ab Initio Help.) The format of the summary file is specified in **$AB_HOME/include/summary.dml**, which contains a record format with conditional fields: the presence or absence of the fields in a specific summary file is determined by the kind of object (**flow**, **component**, **job-start**, or **phase-end**) being reported on. After running the graph, you can examine the summary file to find the flow(s) that are relevant for your audit.

FLOWS IN SUMMARY FILES

In a summary file, flows are represented by a **kind** value of **flow**. Their names consist of the name of the component and the port to which the flow is attached. Special characters in the flow name are replaced by underscores.

To use a summary file:

1. Choose **Edit > Script > Start**, and enter the following **ksh** command to specify where the GDE should create the summary file:

   ```
   export \
   AB_REPORT="$AB_REPORT summary=$RUN/$AB_GRAPH_NAME.summary.log"
   ```

   where *AB_GRAPH_NAME* is an environment variable that the GDE automatically sets to represent the name of your graph, without the **.mp** extension, and with any special characters (such as spaces) replaced by underscores.

2. Run the graph.

   The tracking summary is written to the **Audit File** output component of the graph.

3. Right-click the **Audit File** and choose **View Data** to compare the contents of the summary file with the original graph and find the flows you will need for your audit. In this example you need to make sure that no records were lost between these flows:

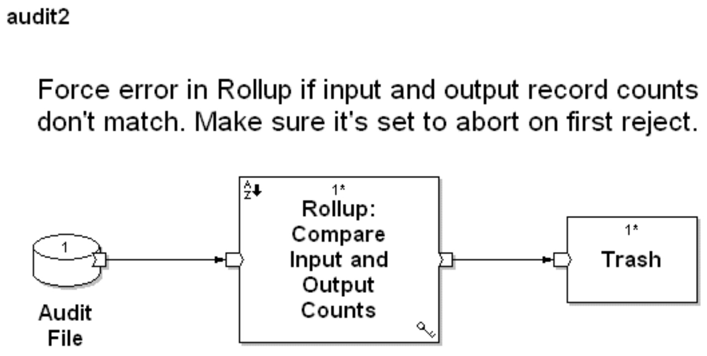
   - **Partition_by_Key_Account_Num.in**
   - **Join_Cust_Acct.out** flows (one for **partition 0** and one for **partition 1**)

   Specifically, you need to verify that the sum of the **num_records** fields (specified in **summary.dml**) for the **Partition_by_Key_Account_Num.in** flow is the same as the sum for the **Join_Cust_Acct.out** flows. The ROLLUP used in this graph and described in the next section accomplishes this.

Using a ROLLUP to check the record counts

To use a rollup to check the record counts:

1. Create a new graph consisting of an INPUT FILE component, a ROLLUP component, and a TRASH component, as shown in **audit2.mp**:


audit2
Force error in Rollup if input and output record counts don't match. Make sure it's set to abort on first reject.
1
Audit File
1*
Rollup: Compare Input and Output Counts
1*
Trash

2. In the INPUT FILE component, specify the path to the previously generated summary file as the URL:

```
file:$RUN/audit1.summary.log
```

3. Set the key on the ROLLUP to the empty key: **{ }**

   Here the output of the ROLLUP is unimportant. Its sole purpose is to force errors if the values from the summary file do not match.

4. In the ROLLUP component, create variables to store the values from the summary file:

```
let decimal("") input_accounts = sum(in.num_records,
  ((in.kind=="flow") and
   (in.name=="Partition_by_Key_Account_Num.in")));
let decimal("") output_accounts = sum(in.num_records,
  ((in.kind=="flow") and (in.name=="Join_Cust_Acct.out")));
```

   Because the summary file uses a record format with conditional fields, **in.name** will exist only for certain values of **in.kind**, so a condition such as **in.kind=="flow"** is required to avoid NULL values. For information about conditional fields, see the topic **Conditional fields** (under **DML Reference > Compound Types > Records**) in Ab Initio Help.

5. Check the input and output account variables for NULL values to catch unexpected errors:

```
if (is_null(input_accounts))
  force_error("No records found in input file");
if (is_null(output_accounts))
  force_error("No records found in output file");
```

   Because the next statement tests for inequality, this is not strictly necessary (a NULL value would fail anyway), but it provides a more informative error message.

6. Force an error and output an appropriate message if the counts of the input and output accounts do not match:

```
if ((input_accounts!=output_accounts))
  force_error(string_concat("Count of input accounts '",
    input_accounts, "' does not match count of output accounts '",
    output_accounts, "'"));
```

   You can surround the values in the error string with single quotes for clarity.

7. Write out the values being compared. This typically makes transforms like this easier to debug (and the ROLLUP will get errors if it does not assign an output):

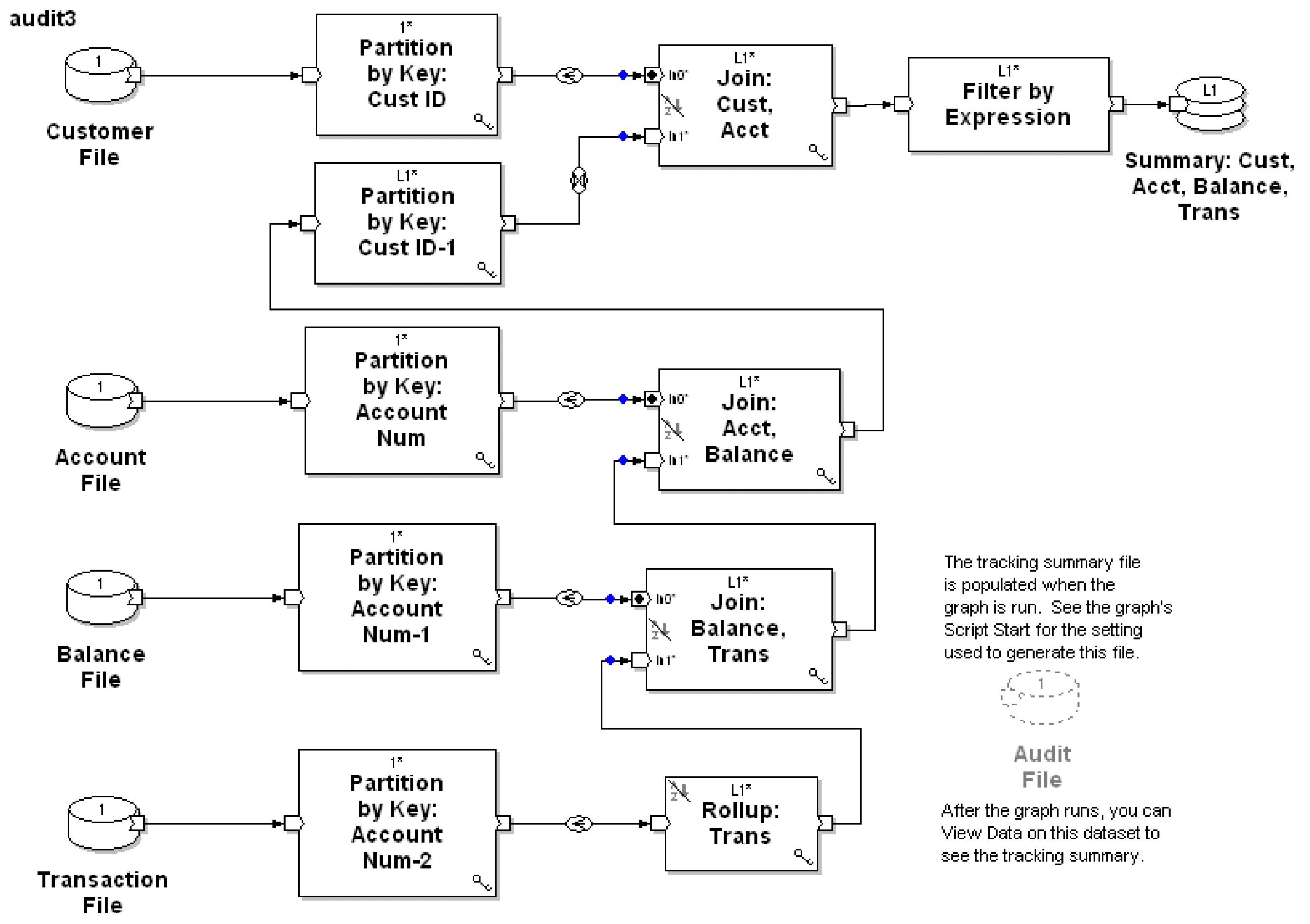
```
out.input_accounts :: input_accounts;
out.output_accounts :: output_accounts;
```

Limitations of using a summary file and a ROLLUP

Using a summary file for auditing has the advantage that you do not complicate your original graph with extra components to collect log information, but has the following limitations:

- A summary file can be used only to audit information written to the file, such as record counts. The summary file does not contain financial totals, for example.

- Using a summary file requires the audit graph to run after the original graph, so scheduling gets more complex and cluttered. Ideally, you would like the original graph to fail if it fails the audit checks.
- Using a summary file creates a possible dependency between the two graphs: changes to the first graph might necessitate changes to the auditing graph. In the best case, failure to update the names in the second graph will cause it to fail after changes to the first graph. The worst case is shown by the following graph (**audit3.mp**):


audit3
Customer File
1* Partition by Key: Cust ID
L1* Partition by Key: Cust ID-1
L1* Join: Cust, Acct
L1* Filter by Expression
L1
Summary: Cust, Acct, Balance, Trans
Account File
1* Partition by Key: Account Num
L1* Join: Acct, Balance
Balance File
1* Partition by Key: Account Num-1
L1* Join: Balance, Trans
Transaction File
1* Partition by Key: Account Num-2
L1* Rollup: Trans
Audit File

The tracking summary file is populated when the graph is run. See the graph's Script Start for the setting used to generate this file.

After the graph runs, you can View Data on this dataset to see the tracking summary.

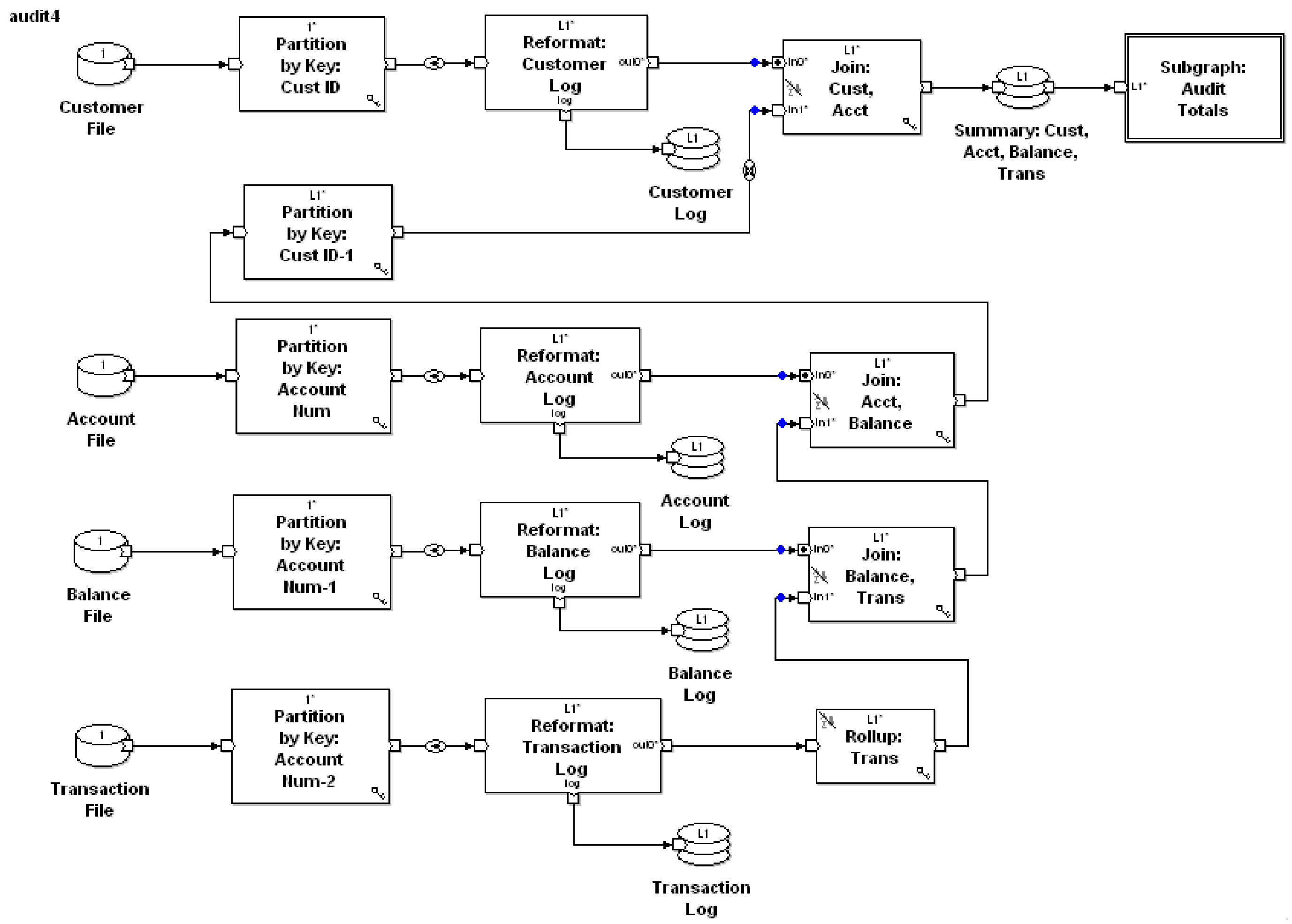
Here an intentional bug has been added: the **select_expr** parameter in the FILTER BY EXPRESSION component is set to zero, causing all the records to be dropped. But if you made this change and did nothing else, the audit graph would report success, because it counts the output from the JOIN. To see this, run this graph, change the input file in **audit3.mp** to point to **file:$RUN/ audit3.summary.log**, and then run **audit3.mp**. It still counts the output from the JOIN, so it reports success even though the output file is empty. To make this graph do a proper audit, you would need to change your **out** flow in **audit3.mp** to **Filter_by_Expression.out**, but that would be easy to forget.

NOTE: If you use summary files for auditing, make sure the original graph includes a legend documenting the need to change the audit graph if the original graph changes.

More complex auditing: Using log files

The more common, though more complex, method of auditing graphs is to add components to the graph to collect the necessary data and audit the results as required.

This recipe describes how to create a self-contained graph (**audit4.mp**, shown below) that not only produces output from its inputs, but also tests various assumptions about how the output should relate to the input. It uses components that are completely separate from the processing logic itself, and it handles odd conditions such as unexpected NULL values and the absence of any input records.


audit4
Customer File
Partition by Key: Cust ID
Reformat: Customer Log
Customer Log
Join: Cust, Acct
Summary: Cust, Acct, Balance, Trans
Subgraph: Audit Totals
Partition by Key: Cust ID-1
Account File
Partition by Key: Account Num
Reformat: Account Log
Account Log
Join: Acct, Balance
Balance File
Partition by Key: Account Num-1
Reformat: Balance Log
Balance Log
Join: Balance, Trans
Transaction File
Partition by Key: Account Num-2
Reformat: Transaction Log
Transaction Log
Rollup: Trans

Specifically, this audit graph checks that:

- No input records have been dropped.
- Total balances are the same in the input and the output.
- The total of all transaction amounts in the input equals the sum of the debits and credits in the output.

To accomplish this, the graph does the following:

1. Uses log files to collect the relevant data from the inputs (page 121).
2. Rolls up the graph's output (page 124).
3. Reads back and rolls up the values from the log files (page 125).
4. Compares the input and output values (page 127).

NOTE: If you follow these steps to audit a similar graph, you should test the audit graph as described in "Testing the audit graph" (page 129).

ANOTHER WAY TO COLLECT DATA ABOUT INPUTS

A straightforward but less efficient way to collect data about the inputs would be to add a REPLICATE component, make a second copy of the input flow, and use a ROLLUP component to count the inputs and/or sum the financial data.

Collecting log data about the inputs

To collect log data about the inputs:

1. After partitioning each input file by key, add a REFORMAT component after each partitioner.

   Adding the REFORMAT components *after* the partitioners (rather than attaching them directly to the input files) is safe, as partitioners never reject records. This allows the collection of log information to proceed in parallel.

2. Define a **log_type** as shown in the following code and save it in a DML file (for **audit4.mp** this is **log_type.dml**) to be included later when you send information to the **log** ports of the four REFORMATs:

   ```
   type log_type =
   record
     string("") event_type;
     string("") event_text;
   end;
   ```

3. Do the following in the REFORMAT components:

   a. For **Customer File** and **Account File**, create a transform that includes **log_type.dml**, creates a global variable *count* to accumulate the count of customers and accounts, and defines a function named **final_log_output** to send *count* to the **log** port, as shown in the following code. (The function *must* be named **final_log_output**. For details, see "About the final_log_output function" on page 124.) The graph **audit4.mp** does this in the transform file **audit_count.xfr**:

      ```
      include "/~$AI_DML/log_type.dml";

      let decimal("") count =0;

      log_type out::final_log_output() =
      begin
        out.event_type :: "audit";
        out.event_text :: count;
      end;

      out::reformat(in) =
      begin
        count = count + 1;
        out.* :: in.*;
      end;
      ```

b. For the **Balance** file, you need to accumulate the count *and* the sum, so write an embedded transform that includes **log_type.dml**, creates global variables *count* and *sum_balances*, and defines a function named **final_log_output** to send the values to the **log** port, as shown in the following code. (Again, the function *must* be named **final_log_output**. For details, see "About the final_log_output function" on page 124.)

```
include "/~$AI_DML/log_type.dml";

let decimal("") count =0;
let decimal("") sum_balances =0;

log_type out::final_log_output() =
begin
  out.event_type :: "audit";
  out.event_text :: string_concat(count, "|", sum_balances);
end;

out::reformat(in) =
begin
  count = count + 1;
  sum_balances = sum_balances + first_defined(in.bal, 0);
  out.* :: in.*;
end;
```

c. For the **Transaction** file, you also need to accumulate the count *and* the sum, so write an embedded transform that includes **log_type.dml**, creates global variables *count* and *sum_tran*, and defines a function named **final_log_output** to send them to the **log** port, as shown in the following code. (Again, the function *must* be named **final_log_output**. For details, see "About the final_log_output function" on page 124).

```
include "/~$AI_DML/log_type.dml";

let decimal("") count =0;
let decimal("") sum_tran =0;
```

ABOUT THE final_log_output FUNCTION

In any component with a transform, if a function named **final_log_output** is present, it is called *after* the transform has been called for *all* input records, and it writes its result to the **log** port, using any values accumulated in global variables. (For details on this function, search Ab Initio Help for **final_log_output function**.)

```
log_type out::final_log_output() =
begin
  out.event_type :: "audit";
  out.event_text :: string_concat(count, "|", sum_tran);
end;

out::reformat(in) =
begin
  count = count + 1;
  sum_tran = sum_tran + first_defined(in.amt, 0);
  out.* :: in.*;
end;
```

Rolling up the graph's output

To perform a meaningful audit, you must check the inputs recorded in your log data against the appropriate values in the graph's outputs. To accomplish this, you need to roll up the outputs to a single count and (for balances and transactions) a single total, as demonstrated in the **Audit Totals** subgraph in **audit4.mp** and described in this section. These ROLLUPs yield the needed data as follows:

- They count and sum the non-NULL balances in the output, which should equal the counts and sums of the input balance records.
- Since each transaction contributed to either a credit count and sum or a debit count and sum, the rolled-up counts and sums of the debits and credits on the output should equal the transaction counts and sums on the input.

To roll up the graph's output:

1. Roll up the graph's output on **cust_id** to find distinct customers.

2. Roll up the output of the first ROLLUP in parallel on the empty key: **{ }** to generate a single grand total for each input.

   Doing these two ROLLUPs is more efficient than rolling up directly from the account level, because many customers have multiple accounts, so there are fewer records after the ROLLUP to the customer level. Also, because the input to the first ROLLUP is at the account level and the second is at the customer level, and both inputs might be a lot of records, these ROLLUPs should be done in parallel.

3. Roll up the output of the second ROLLUP in serial to generate a single grand total from the totals of the individual partitions.

   Since the data is already partitioned by **cust_id**, there is no double-counting. Note also that layouts of these last two ROLLUPs are set explicitly to prevent propagation from causing something to run in the wrong layout.

Reading back and rolling up the values from the log files

Before your subgraph can audit the results by comparing the log values with the output values from the main graph, you must generate the needed log values (those with an **event_type** of **audit**) in the appropriate format. Also, since the log files were attached to multiple components executing in parallel, you need to roll these up to get the total values for comparison.

To do this, write separate transforms to read back and roll up the log values for the following two cases:

- **Customer** and **Account** logs — **count** values
- **Balance** and **Transaction** logs — **count** and **sum** values

Customer and account logs (count values) Do this ROLLUP by putting the code shown below in your transform. In **audit4.mp**, this code is in **sum_log_counts.xfr**. Note that because the **event_text** field in log files is stored as a string delimited by **|\n**, you must cast it to a decimal so that you can use the data as a number.

```
out.sum_counts :: sum((decimal(""))in.event_text,
                      in.event_type == "audit");
```

Balance and transaction logs (count and sum values) Reading back the log values for the balances and transactions is more complex, because two values separated by **"|"** are stored in a single **in.event_text** field in the log file. You can use the **reinterpret_as** function to interpret this string as a record containing those two values. In **audit4.mp**, the code for this is in **sum_log_counts_values.xfr**.

To read back the log values for the balances and transactions:

- Create an expression with a record format that matches the format used for the **event_text** in the log file, and assign this expression to a variable as follows:

  a. On the **Variables** tab of the ROLLUP transform, add the following expression:

  ```
  reinterpret_as(record decimal("|") count; decimal("|\n") sum;
  end, in.event_text);
  ```

  b. Drag the expression across to the **Variables** output pane.

  The GDE creates a variable of type **record** with a default name such as **expr1** and the following format, and assigns the expression to it:

  ```
  record
    decimal("|") count;
    decimal("|\n") sum;
  end
  ```

c. Rename the variable appropriately (**log**, for example) by clicking it twice (not double-clicking) and typing a new name.

In **Text View**, the resulting code looks like this:

```
out::rollup(in) =
begin
  let record
    decimal("|") count;
    decimal("|\n") sum;
  end log =
    reinterpret_as(record
                     decimal("|") count;
                     decimal("|\n") sum;
                   end,
                   in.event_text);
```

d. Roll up **log.count** and **log.sum** and finish the transform as follows:

```
  out.sum_counts :: sum(log.count, in.event_type == "audit");
  out.sum_values :: sum(log.sum, in.event_type == "audit");
end;
```

MAKING YOUR GRAPHS AND CODE MORE READABLE

You can change the default names assigned to subgraph ports and numbered ports as follows:

1. On the **Ports** tab of the component or subgraph, right-click the name of the port.
2. Click **Rename**.
3. Type a name for the port and press Enter.

Comparing the input and output values

After rolling up the graph's output data and the generated log data, you have five flows, each rolled up to a single record containing totals: one for the graph's output and four for the log outputs.

To check the values (as shown in the Audit Totals subgraph in audit4.mp):

1. Do an explicit join of the five flows.

TIP: Change the names of the **in** ports on the JOIN component to make them more descriptive.

2. To ensure that the JOIN transform gets called (has something to join against) if some or all of these flows have no records, so that you can flag NULL flows as errors, send the **log** port of the second ROLLUP (parallel rollup to grand total) into the JOIN, and give the port an appropriate name. In the **Audit Totals** subgraph, this is the **rollup_log** port.

   NOTE: The **log** port of a transform always has one or more records, even if the transform did not receive any input records.

   TIP: Another way of ensuring that a transform gets called is to add a GENERATE RECORDS component to create a single dummy record.

3. Set the **record_required** parameter of the **log** port's flow into the JOIN to **True (record required on this input)**. (In the **Audit Totals** subgraph, this is **record_required1**.)

4. To ensure that the **log** port's flow has exactly one record, set the appropriate **dedup** parameter of the JOIN to **Dedup this input before joining.** (In the **Audit Totals** subgraph, this is **dedup1**.)

5. Test for NULL and force an error whenever the counts and totals on the inputs and outputs do not match as expected.

   In other words, write a sequence of conditions of the form **if (error condition) force_error("error message")**. For example:

```
if (is_null(totals))
  force_error("No records in output dataset");
...
if ((totals.debits + totals.credits) != transactions.sum_values)
  force_error(string_concat(
    "Sum of input transaction amounts '",
    (decimal("")) transactions.sum_values,
    "' does not match sum of output debits and credits '",
    (decimal("")) (totals.debits + totals.credits),
    "'"
    ));
```

6. To aid in debugging (and to provide outputs, which are required for every JOIN), assign the counts and sums from the grand total to the outputs:

```
    out.* :: totals.*;
```

Testing the audit graph

Graphs that involve this sort of auditing require extensive testing. It is not sufficient just to run some legitimate test data. You should also create test conditions to trigger every possible error message (even if you think some conditions cannot possibly occur) and make sure that the correct error is actually detected in each case. This may be impossible to do using just the original graph, since you coded it to handle all errors.

Often a useful technique is to add a component directly before your output file to cause various errors. For example, you could add a FILTER BY EXPRESSION to drop one (or all) records, or a REFORMAT to change the value of a single field:

```
out.foo :: if (next_in_sequence() == 500) 0 else in.foo;
```

Processing sets of files

In many situations, an input to a graph is not a single file, but several files. The files may hold data collected from different locations (such as daily cash register data from each of several stores), at different times (hourly logs of a Web server), or both.

The recipes in this chapter present two approaches to processing such data:

- Using ad hoc multifiles to process multiple files in data parallel fashion
- Using READ MULTIPLE FILES and WRITE MULTIPLE FILES components to read and write multiple files in a single processing stream

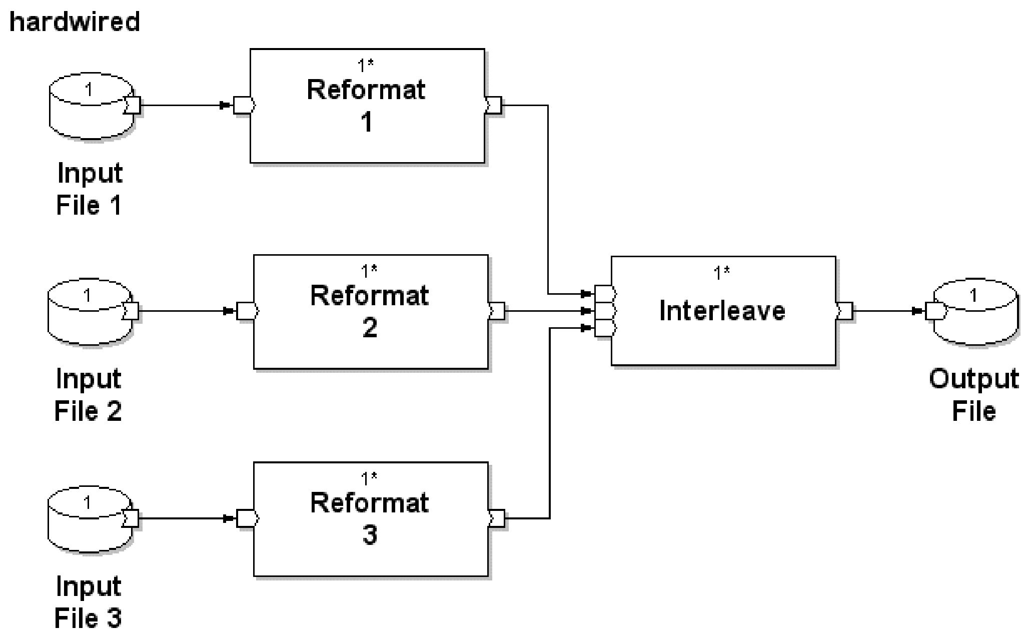
The two simplest approaches to processing sets of files both have disadvantages in certain situations:

- Running the same graph multiple times, once for each file

  If the files can be processed independently, this may be acceptable if there are not very many of them. If they are numerous, however, the overhead of starting up the graph once for every file can be excessive.

- Building a graph that processes a fixed number of files, with one INPUT FILE component for each

  This approach (shown in **hardwired.mp** below) may allow some parallelism, but it is inherently inflexible, as the graph can process only the number of files specified when the graph was developed.



Ad hoc multifiles

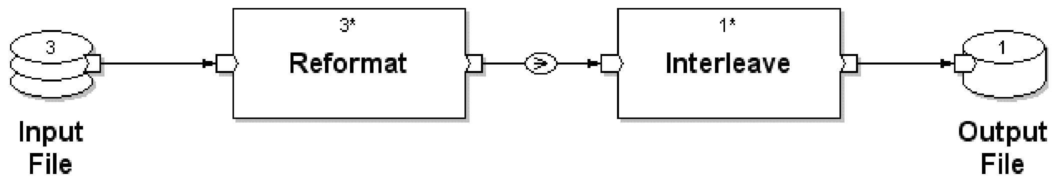
This section presents four graphs (provided in the **file-sets** sandbox) that show they all produce the same output. But each graph demonstrates a different way of specifying multiple files, as described in the following subsections:

- Listing files explicitly (**ad_hoc_explicit.mp**, page 134)
- Using wildcards to list files (**ad_hoc_wild_card.mp**, page 135)
- Using a command to list files (**ad_hoc_command.mp**, page 137)
- Using a parameter to list files (**ad_hoc_parameter.mp**, page 137)

NOTE: One potential downside to ad hoc multifiles is that if a very large number of partitions (hundreds or thousands) are being processed, the graph may end up running hundreds or thousands of ways parallel. This could stress or exhaust system resources.

Each of the following graphs treats three serial input files as a multifile, reformats the data in the **description** field into a one-character **trans-kind** field, interleaves the partitions, and sends the result to a single serial file. For example:

ad_hoc_explicit



All three input files have the same record format, which is based on the user-defined type specified in **transaction_type.dml**:

```
type transaction =
  record
    date("YYYY.MM.DD") trans_date;
    decimal(9,0) account_id;
    decimal(10.2) amount;
    string('\n') description;
  end;
```

The input files use this by including **transaction_type.dml** in the record format and identifying **transaction** as the top-level type:

```
include "/~$AI_DML/transaction_type.dml";
metadata type = transaction;
```

The input data consists of three files containing transaction records for three months:

- **trans-2006-01.dat**
- **trans-2006-02.dat**
- **trans-2006-03.dat**

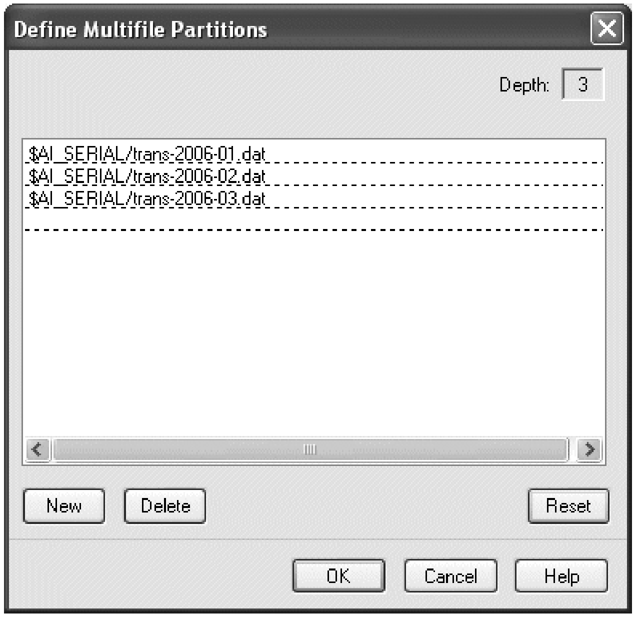
Listing files explicitly

To define an ad hoc multifile by listing files explicitly:

1. On the **Description** tab of the **Properties** dialog for the INPUT FILE, click **Edit**.
2. In the **Define Multifile Partitions** dialog, click **New** and enter the full path of the first file you want to process. For example:

   ```
   $AI_SERIAL/trans-2006-01.dat
   ```
3. Repeat Step **2** analogously for each additional file, and click **OK**.

   In **ad_hoc_explicit.mp**, the **Define Multifile Partitions** dialog looks like this:



Depth: 3

```
$AI_SERIAL/trans-2006-01.dat
$AI_SERIAL/trans-2006-02.dat
$AI_SERIAL/trans-2006-03.dat
```

4. Click **OK** twice to close the **Define Multifile Partitions** dialog and the **Properties** dialog.

When you run the graph, the INPUT FILE behaves like a three-way multifile whose data partitions are the three listed files; components can run in the layout of the INPUT FILE component.

Using wildcards to list files

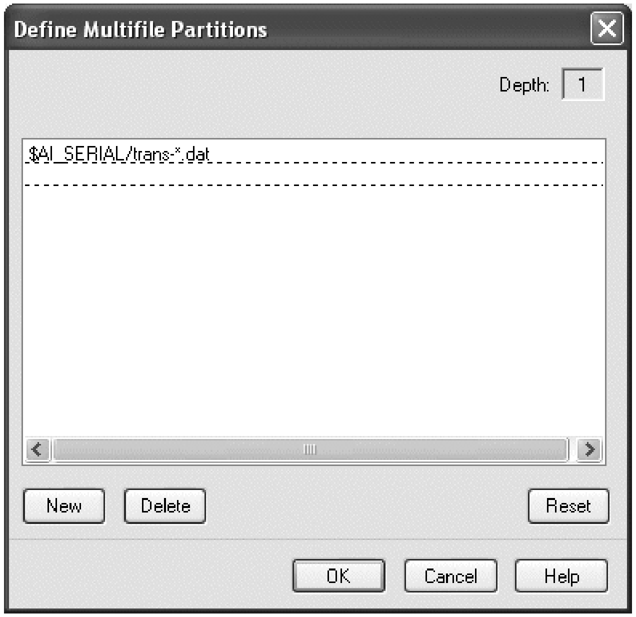
Listing files by using wildcards is one of the most common ways to specify an ad hoc multifile, because often the names of the files to be processed are similar but are not precisely known until the graph is run.

To define an ad hoc multifile by using wildcards:

1. On the **Description** tab of the **Properties** dialog for the INPUT FILE, click **Edit**.
2. In the **Define Multifile Partitions** dialog, click **New** and enter the full path of the files you want to process, using an asterisk (*) as a wildcard to represent any sequence of characters that is not identical in the names of all files you want to process. For example:

   ```
   $AI_SERIAL/trans*.dat
   ```

   In **ad_hoc_wild_card.mp**, the **Define Multifile Partitions** dialog looks like this:

   $AI_SERIAL/trans-*.dat

3. Click **OK** twice to close the **Define Multifile Partitions** dialog and the **Properties** dialog.

Using a command to list files

A shell command is useful if you want to:

- Generate an ad hoc multifile based on file properties other than their names
- Get a list of files from another program

The graph **ad_hoc_command.mp** shows the simple case of using the **ls** command to generate an ad hoc multifile. This approach is not recommended (using a wildcard as described on page 135 would be easier), but it demonstrates the technique.

To define an ad hoc multifile by using a command:

1. On the **Description** tab of the **Properties** dialog for the INPUT FILE, click **Edit**.

2. In the **Define Multifile Partitions** dialog, click **New** and enter the shell command that will generate a list of the files you want to process. For example:

```
$(ls $AI_SERIAL/ad_hoc_input_*.dat)
```

NOTE: You must enclose the command in parentheses and precede the opening parenthesis with a dollar sign: **$(...)** to indicate that what follows is a **ksh** command.

3. Click **OK** twice to close the **Define Multifile Partitions** dialog and the **Properties** dialog.

Using a parameter to list files

You can also define an ad hoc multifile by using a graph parameter. This is useful, for example, if an external process is being used to provide a list of the input files.

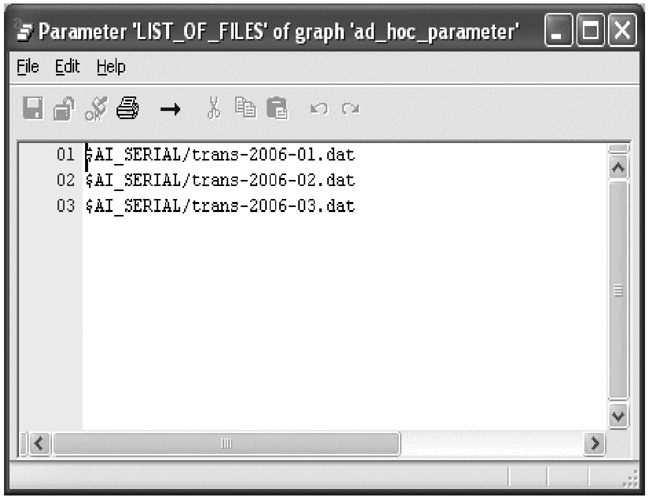
To define an ad hoc multifile by using a parameter:

1. Choose **Edit > Parameters**.

2. In the **Edit Parameters** dialog, choose **Edit > Insert**.

3. Add a graph parameter (for example, **LIST_OF_FILES**) and make sure that its interpretation is set to **$ substitution**.
4. Click the pencil icon to the right of the parameter name.
5. In the **Editing** *name* **parameter** dialog, type the full paths of the input files, separated by spaces or carriage returns, and save the parameter.

   In **ad_hoc_parameter.mp**, the **LIST_OF_FILES** parameter is defined as follows:

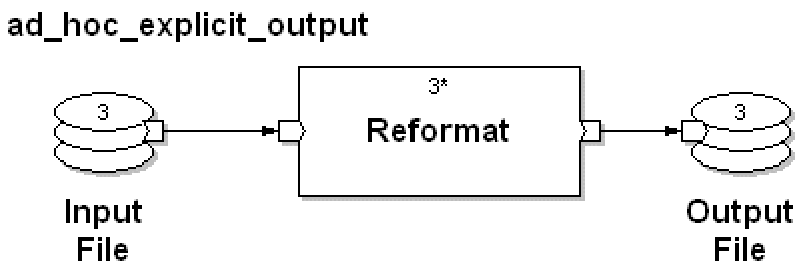


```
$AI_SERIAL/trans-2006-01.dat
$AI_SERIAL/trans-2006-02.dat
$AI_SERIAL/trans-2006-03.dat
```

6. Click **OK** to close the **Edit Parameters** dialog.

Ad hoc output multifiles

Ad hoc output multifiles are useful when a downstream processing system needs to receive a set of files rather than a single serial file or a multifile. The graph **ad_hoc_explicit_output.mp** demonstrates this.



NOTE: Because both the input and output are partitioned three ways, the REFORMAT's processing will run three-ways parallel — one way for each partition.

To use an explicitly defined ad hoc output multifile, follow the same steps you would for an input file (see page 134).

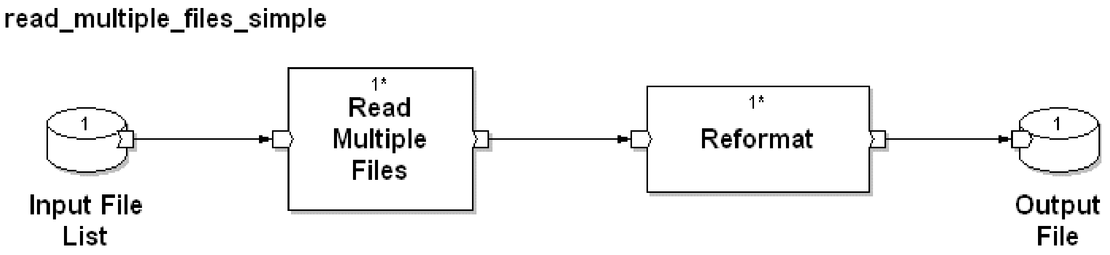
READ MULTIPLE FILES and WRITE MULTIPLE FILES components

The recipes in this section demonstrate and describe the following techniques:

- Reading multiple files (below) — **read_multiple_files_simple.mp** reads three input files, reformats them to use a different record type, and writes the file contents to a single serial file.
- Reading and writing multiple files (page 144) — **read_and_write_multiple_files.mp** reads three input files, reformats them to use a different record type and add a field containing the filename, and writes the file contents to three appropriately named serial files.

Reading multiple files

The graph **read_multiple_files_simple.mp** reads the files specified in a list and writes their contents to a single serial file:



To read multiple files from a list and write their contents to a single serial file:

1. Add an INPUT FILE component whose URL specifies a file containing a list of the files to read and whose record format specifies how the filenames are separated.

   In **read_multiple_files_simple.mp**, the URL of **Input File List** is **file:$AI_SERIAL/ list_of_files.dat**, and the referenced file has the following content:

   ```
   trans-2006-01.dat
   trans-2006-02.dat
   trans-2006-03.dat
   ```

   The record format for **list_of_files.dat** files is:

   ```
   record
     string('\n') filename;
   end
   ```

2. Connect a READ MULTIPLE FILES component to the INPUT FILE and configure it as follows:

   a. Hold down the Shift key and double-click the component to generate a default transform. (If the **Transform Editor** is displayed in **Package View**, switch to **Text View**, and you will be prompted to generate the transform.)

b. Delete all the transform code except the **get_filename** function.

c. Code the **get_filename** function such that it will generate the full path to each input file and then retrieve its content. For example:

```
filename :: get_filename(in) =
begin
  filename :: string_concat($AI_SERIAL, '/', in.filename);
end;
```

d. On the **Ports** tab of the **Properties** dialog for the READ MULTIPLE FILES component, specify the record format for the output data (which is the record format for each file in the list). To do this, select **Use file** and then browse to the location of a file containing the record format.

This graph uses **transaction.dml**, which includes **transaction_type.dml**. The definition of **transaction** in **transaction_type.dml** is as follows:

```
type transaction =
  record
    date("YYYY.MM.DD") trans_date;
    decimal(9,0) account_id;
    decimal(10.2) amount;
    string('\n') description;
  end;
```

The file **transaction.dml** declares the named type **transaction** as a top-level type for use as the output record format:

```
include "/~$AI_DML/transaction_type.dml";
metadata type = transaction;
```

The graph **read_and_write_multiple_files.mp**, described on page 144, also uses this type: its READ MULTIPLE FILES component uses it as the **input_type** and as part of the output record format.

USER-DEFINED TYPES

Creating a user-defined type such as **transaction** in a separate DML file allows you to reuse the type in different ways. You can use it as the top-level type in a record format, as in **transaction.dml**, or include it in a transform function, as described in "Reading and writing multiple files" (page 144).

3. Attach a REFORMAT component to the READ MULTIPLE FILES component and code its transform appropriately. To code it like the REFORMAT in **read_multiple_files_simple.mp**, do the following:

   a. Include the transform file **map_trans_kind.xfr**, which replaces the data in each **description** field with a single character as follows:

   ```
   out :: map_trans_kind(description) =
   begin
     out :1: if (string_index(description, "ATM"))    "A";
     out :2: if (string_index(description, "Teller")) "T";
     out :3: if (string_index(description, "Check"))  "C";
     out :: "O";
   end;
   ```

   b. Output all fields, mapping the reformatted **description** field to a **trans_kind** field in the output. The complete REFORMAT transform in **read_multiple_files_simple.mp** looks like this:

   ```
   include "/~$AI_XFR/map_trans_kind.xfr";

   out::reformat(in) =
   begin
     out.* :: in.*;
     out.trans_kind :: map_trans_kind(in.description);
   end;
   ```

4. Attach an OUTPUT FILE component to the REFORMAT component and configure it as follows:

   a. On the **Description** tab of the **Properties** dialog for the OUTPUT FILE, specify the URL for the serial file to which you want to write the output data. For example:

   ```
   file:$AI_SERIAL/processed.dat
   ```

   b. On the **Ports** tab of the **Properties** dialog for the OUTPUT FILE, specify the record format for the output data by selecting **Use file** and then browsing to the location of a file containing the record format.

This graph uses **processed.dml**, which includes **processed_type.dml**. The definition of **processed** in **processed_type.dml** is as follows:

```
type processed =
  record
    date("MMM DD, YYYY") trans_date;
    decimal(9,0) account_id;
    decimal(10.2) amount;
    string('\n') trans_kind;
  end;
```

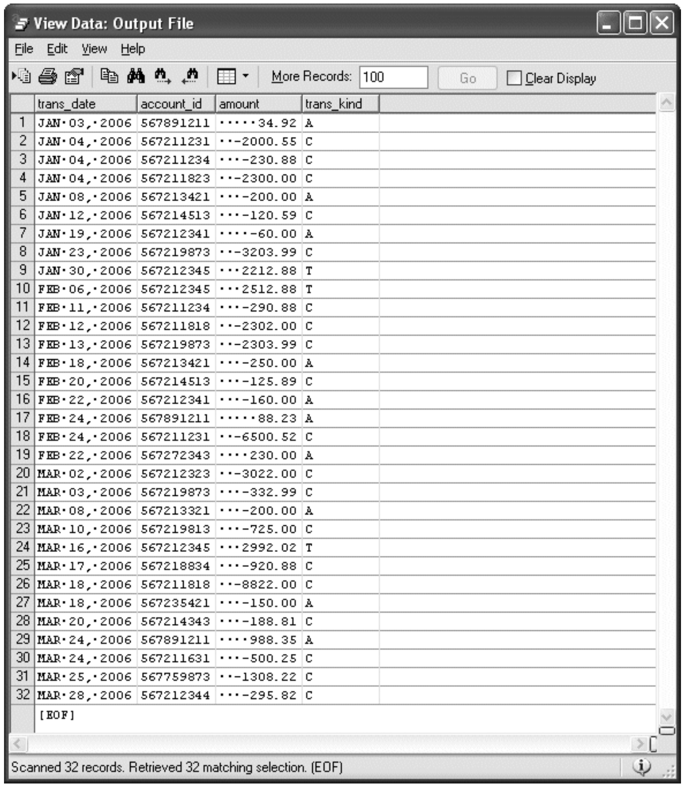
The file **processed.dml** declares the named type **processed** as a top-level type for use as the output record format:

```
include "/~$AI_DML/processed_type.dml";
metadata type = processed;
```

Again, **read_and_write_multiple_files.mp**, described on page 144, uses this type in a different way: its REFORMAT component uses the type as part of an embedded output format that also includes the filename for each record, and its WRITE MULTIPLE FILES component uses it as the **output_type**.

5. Run the graph and view the results.

   The output should look like this:



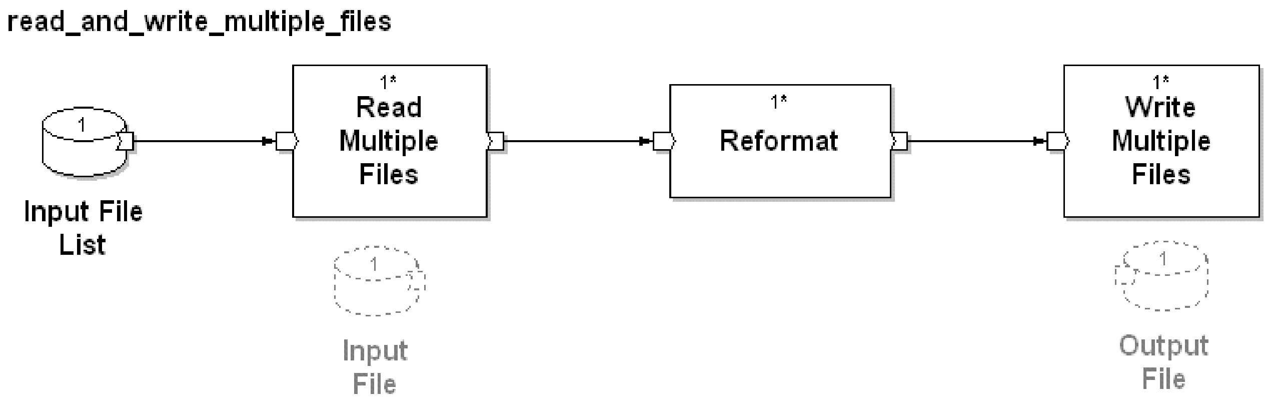
	trans_date	account_id	amount	trans_kind
1	JAN·03,·2006	567891211	·····34.92	A
2	JAN·04,·2006	567211231	··-2000.55	C
3	JAN·04,·2006	567211234	···-230.88	C
4	JAN·04,·2006	567211823	··-2300.00	C
5	JAN·08,·2006	567213421	···-200.00	A
6	JAN·12,·2006	567214513	···-120.59	C
7	JAN·19,·2006	567212341	····-60.00	A
8	JAN·23,·2006	567219873	··-3203.99	C
9	JAN·30,·2006	567212345	···2212.88	T
10	FEB·06,·2006	567212345	···2512.88	T
11	FEB·11,·2006	567211234	···-290.88	C
12	FEB·12,·2006	567211818	··-2302.00	C
13	FEB·13,·2006	567219873	··-2303.99	C
14	FEB·18,·2006	567213421	···-250.00	A
15	FEB·20,·2006	567214513	···-125.89	C
16	FEB·22,·2006	567212341	···-160.00	A
17	FEB·24,·2006	567891211	·····88.23	A
18	FEB·24,·2006	567211231	··-6500.52	C
19	FEB·22,·2006	567272343	····230.00	A
20	MAR·02,·2006	567212323	··-3022.00	C
21	MAR·03,·2006	567219873	···-332.99	C
22	MAR·08,·2006	567213321	···-200.00	A
23	MAR·10,·2006	567219813	···-725.00	C
24	MAR·16,·2006	567212345	···2992.02	T
25	MAR·17,·2006	567218834	···-920.88	C
26	MAR·18,·2006	567211818	··-8822.00	C
27	MAR·18,·2006	567235421	···-150.00	A
28	MAR·20,·2006	567214343	···-188.81	C
29	MAR·24,·2006	567891211	····988.35	A
30	MAR·24,·2006	567211631	···-500.25	C
31	MAR·25,·2006	567759873	··-1308.22	C
32	MAR·28,·2006	567212344	···-295.82	C
	[EOF]			

Scanned 32 records. Retrieved 32 matching selection. (EOF)

Reading and writing multiple files

The graph **read_and_write_multiple_files.mp** reads the contents of the same three input files as **read_multiple_files_simple.mp** (described on page 140). However, rather than simply writing the reformatted records to a single output file, it uses READ MULTIPLE FILES and WRITE MULTIPLE FILES components in combination to get the filenames and write the records to three appropriately renamed files. It shows how to:

- Include data from the input to a READ MULTIPLE FILES component in its output records. In this example, the filenames are added to each output record, and the output format is different from the record format of the files that the component reads.
- Write multiple files, using a different record format than that of the data flowing into the WRITE MULTIPLE FILES component. In this example, the output records of the REFORMAT include the filename, but the records written by the WRITE MULTIPLE FILES component do not. Instead, the component uses that information to name the files it writes to.



To read multiple files from a list and write reformatted data to renamed output files:

1. Add an INPUT FILE component whose URL specifies a file containing a list of the files to read and whose record format specifies how the filenames are separated.

   In **read_and_write_multiple_files.mp**, the URL of **Input File List** is **file:$AI_SERIAL/list_of_files.dat**, and the referenced file has the following content:

   ```
   trans-2006-01.dat
   trans-2006-02.dat
   trans-2006-03.dat
   ```

The record format for **list_of_files.dat** is:

```
record
  string('\n') filename;
end
```

2. Connect a READ MULTIPLE FILES component to the INPUT FILE and configure it as follows:

   a. Hold down the Shift key and double-click the READ MULTIPLE FILES component to generate a default transform. (If the **Transform Editor** is displayed in **Package View**, switch to **Text View**, and you will be prompted to generate the transform.)

   b. Specify the record format for reading the records from the input files:

   ```
   include "/~$AI_DML/transaction_type.dml";
   type input_type = transaction;
   ```

   c. Code the **get_filename** function such that it will generate the full path to each input file and then retrieve its content. For example:

   ```
   filename :: get_filename(in) =
   begin
     filename :: string_concat($AI_SERIAL, '/', in.filename);
   end;
   ```

   d. Code the **reformat** function to output the name of each input file and all desired fields.

   This graph outputs the fields defined by **transaction** in the output record format as described in Step **e**:

   ```
   out::reformat(in, filename) =
   begin
     out.filename :: filename;
     out.trans :: in;
   end;
   ```

e. On the **Ports** tab, select **Embed** and specify the record format for the output.

   This graph includes the file **transaction_type.dml**, which contains the type definition of **transaction** (page 141), and creates a metadata type consisting of fields for it and the filename:

```
include "/~$AI_DML/transaction_type.dml";
metadata type =
  record
    string(",") filename;
    transaction trans;
  end;
```

3. Connect a REFORMAT to the READ MULTIPLE FILES component and configure it as follows:

   a. On the **Ports** tab, select **Embed** and specify the record format for the output.

   This graph includes the file **processed_type.dml**, which contains the type definition of **processed** (page 143), and creates a metadata type consisting of fields for it and the filename:

```
include "/~$AI_DML/processed_type.dml";
metadata type =
  record
    string(",") filename;
    processed trans;
  end;
```

   b. Code the transform as follows to output the filenames and the **processed** fields.

   This graph replaces the **description** field of the **transaction** type with the **trans_kind** field of the **processed** type as follows:

```
include "/~$AI_XFR/map_trans_kind.xfr";
```

```
out::reformat(in) =
begin
  out.filename :: in.filename;
  out.trans.* :: in.trans.*;
  out.trans.trans_kind :: map_trans_kind(in.trans.description);
end;
```

4. Connect a WRITE MULTIPLE FILES to the REFORMAT component and code its transform as follows:

a. Specify the record format of the output type.

This graph includes the DML file containing the definition of the desired output type, **processed**:

```
include "/~$AI_DML/processed_type.dml";
type output_type = processed;
```

b. Name the output files appropriately:

This graph replaces the string '**trans**' from the input filenames with '**processed**' in the output filenames:

```
filename::get_filename(in) =
begin
  filename :: string_replace(in.filename, 'trans',
                             'processed');
end;
```

c. Write the reformatted transaction data to each output file:

```
write::reformat(in) =
begin
  write :: in.trans;
end;
```

The WRITE MULTIPLE FILES component uses the result of the **reformat** function as the data to write to the file.

5. If you want to be able to view the contents of one or more of the input files listed in **Input File List**, add an INPUT FILE component and configure it as follows:

    a. Set its URL to the location of the input file.

    In **read_and_write_multiple_files.mp**, this is **file:$AI_SERIAL/trans-2006-01.dat**.

    b. Disable it to prevent others from thinking it is an actual input.

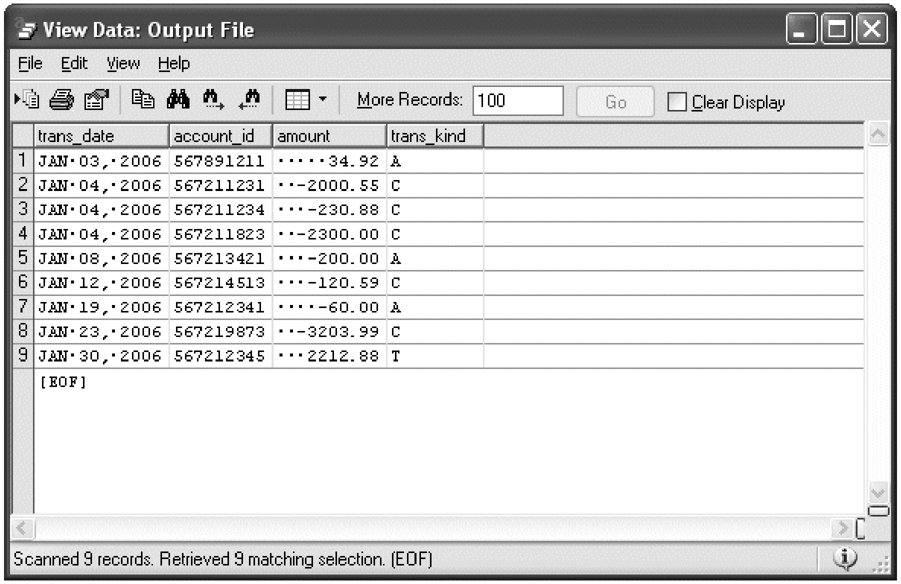
6. If you want to be able to view the contents of one or more of the output files written by the WRITE MULTIPLE FILES component, add an OUTPUT FILE component and configure it as follows:

    a. Set its URL to the location of the output file.

    In **read_and_write_multiple_files.mp**, this is **file:$AI_SERIAL/processed-2006-01.dat**.

    b. Disable it to prevent others from thinking it is an actual output.

    In **read_and_write_multiple_files.mp**, the data for the first output file looks like this:



	trans_date	account_id	amount	trans_kind
1	JAN·03,·2006	567891211	·····34.92	A
2	JAN·04,·2006	567211231	··-2000.55	C
3	JAN·04,·2006	567211234	···-230.88	C
4	JAN·04,·2006	567211823	··-2300.00	C
5	JAN·08,·2006	567213421	···-200.00	A
6	JAN·12,·2006	567214513	···-120.59	C
7	JAN·19,·2006	567212341	····-60.00	A
8	JAN·23,·2006	567219873	··-3203.99	C
9	JAN·30,·2006	567212345	···2212.88	T

[EOF]

Scanned 9 records. Retrieved 9 matching selection. [EOF]

Header and trailer record processing

While most datasets contain only one kind of record, some (particularly those from legacy systems) may contain several. The most common form of such data consists of groups of records that each have a **header** record, followed by some number of **body** records, followed by a **trailer** record. The header record usually contains information that applies to each of the body records, and the trailer often contains summary information. For example, a dataset could consist of groups of transactions, with each group made up of a header record containing customer information, body records for each transaction, and a trailer record containing totals.

This chapter begins by describing three different formats for representing data that consists of headers, bodies, and trailers. It then describes graphs that process data in those formats:

- Generating separate summaries of header and body subrecords (page 160)
- Distributing header information to body records (page 165)
- Normalizing vector data to header and body records (page 167)
- Validating flat input data (page 169)
- Validating vector input data (page 174)

Finally, an advanced topic (page 176) describes a graph that constructs header/body/trailer data from simple records.

Input data formats

With Ab Initio DML, the same data can often be described in many ways. The graph **ht_data.mp** (in the **header** sandbox) shows three ways of describing the same physical data:

- Flat record format (page 153)
- Subrecord format (page 155)
- Vector format (page 158)

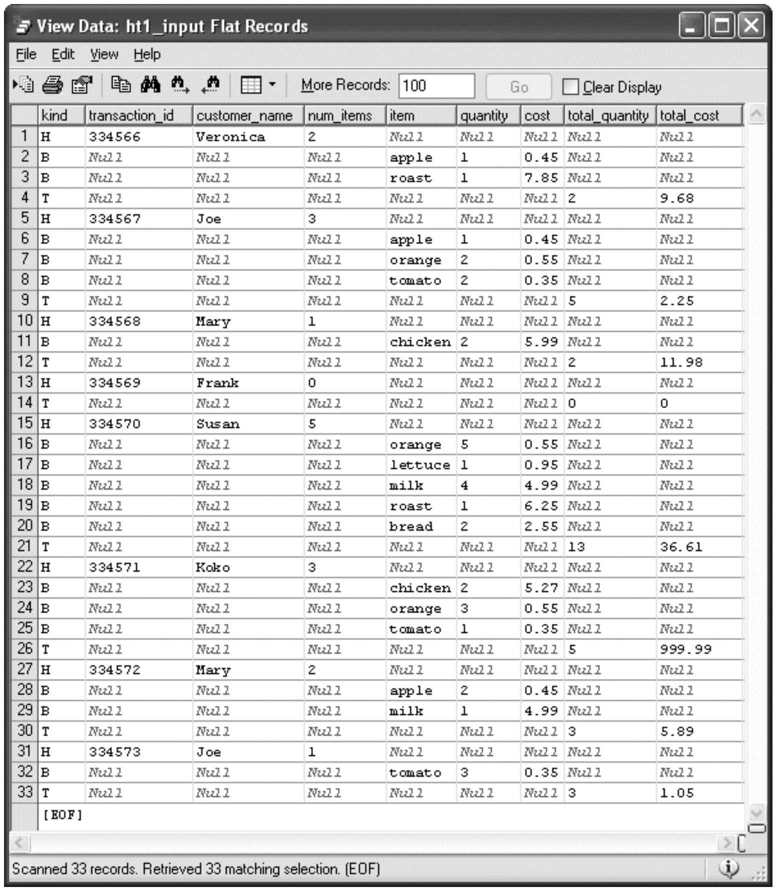
Rather than running **ht_data.mp** (it does not do anything), you should view the record format and the data for each INPUT FILE component in it to see the differences in how the records are described and displayed.

Flat records

The records in the dataset **ht1_input Flat Records** (in **ht_data.mp**) are represented in a flat format (defined in **ht1_input.dml**) that uses conditional fields. The presence or absence of a value in each conditional field is determined by the value of the **kind** field:

```
record
  string(1) kind; /* One of H, B or T */
  if (kind=="H")
    begin
      decimal(',') transaction_id;
      string(',') customer_name;
      decimal('\n') num_items;
    end;
  if (kind=="B")
    begin
      string(',') item;
      decimal(',') quantity;
      decimal('\n') cost;
    end;
  if (kind=="T")
    begin
      decimal(',') total_quantity;
      decimal('\n') total_cost;
    end;
end
```

For each customer transaction, this format results in at least two records (one header and one trailer) with all nine fields in each record, and NULL values for each empty field. To see this, view the data in **ht1_input Flat Records**:



	kind	transaction_id	customer_name	num_items	item	quantity	cost	total_quantity	total_cost
1	H	334566	Veronica	2	Null	Null	Null	Null	Null
2	B	Null	Null	Null	apple	1	0.45	Null	Null
3	B	Null	Null	Null	roast	1	7.85	Null	Null
4	T	Null	Null	Null	Null	Null	Null	2	9.68
5	H	334567	Joe	3	Null	Null	Null	Null	Null
6	B	Null	Null	Null	apple	1	0.45	Null	Null
7	B	Null	Null	Null	orange	2	0.55	Null	Null
8	B	Null	Null	Null	tomato	2	0.35	Null	Null
9	T	Null	Null	Null	Null	Null	Null	5	2.25
10	H	334568	Mary	1	Null	Null	Null	Null	Null
11	B	Null	Null	Null	chicken	2	5.99	Null	Null
12	T	Null	Null	Null	Null	Null	Null	2	11.98
13	H	334569	Frank	0	Null	Null	Null	Null	Null
14	T	Null	Null	Null	Null	Null	Null	0	0
15	H	334570	Susan	5	Null	Null	Null	Null	Null
16	B	Null	Null	Null	orange	5	0.55	Null	Null
17	B	Null	Null	Null	lettuce	1	0.95	Null	Null
18	B	Null	Null	Null	milk	4	4.99	Null	Null
19	B	Null	Null	Null	roast	1	6.25	Null	Null
20	B	Null	Null	Null	bread	2	2.55	Null	Null
21	T	Null	Null	Null	Null	Null	Null	13	36.61
22	H	334571	Koko	3	Null	Null	Null	Null	Null
23	B	Null	Null	Null	chicken	2	5.27	Null	Null
24	B	Null	Null	Null	orange	3	0.55	Null	Null
25	B	Null	Null	Null	tomato	1	0.35	Null	Null
26	T	Null	Null	Null	Null	Null	Null	5	999.99
27	H	334572	Mary	2	Null	Null	Null	Null	Null
28	B	Null	Null	Null	apple	2	0.45	Null	Null
29	B	Null	Null	Null	milk	1	4.99	Null	Null
30	T	Null	Null	Null	Null	Null	Null	3	5.89
31	H	334573	Joe	1	Null	Null	Null	Null	Null
32	B	Null	Null	Null	tomato	3	0.35	Null	Null
33	T	Null	Null	Null	Null	Null	Null	3	1.05
	[EOF]								

Scanned 33 records. Retrieved 33 matching selection. (EOF)

Subrecords

The records in the dataset **ht1_input Using Subrecords** (in **ht_data.mp**) are represented in a format that uses subrecords as follows:

- **ht1_input_subrec_types.dml** defines three types: **header_type**, **body_type**, and **trailer_type**
- **ht1_input_subrec.dml** includes **ht1_input_subrec_types.dml** and defines a record format with three fields, one of each of the three types defined in **ht1_input_subrec_types.dml**

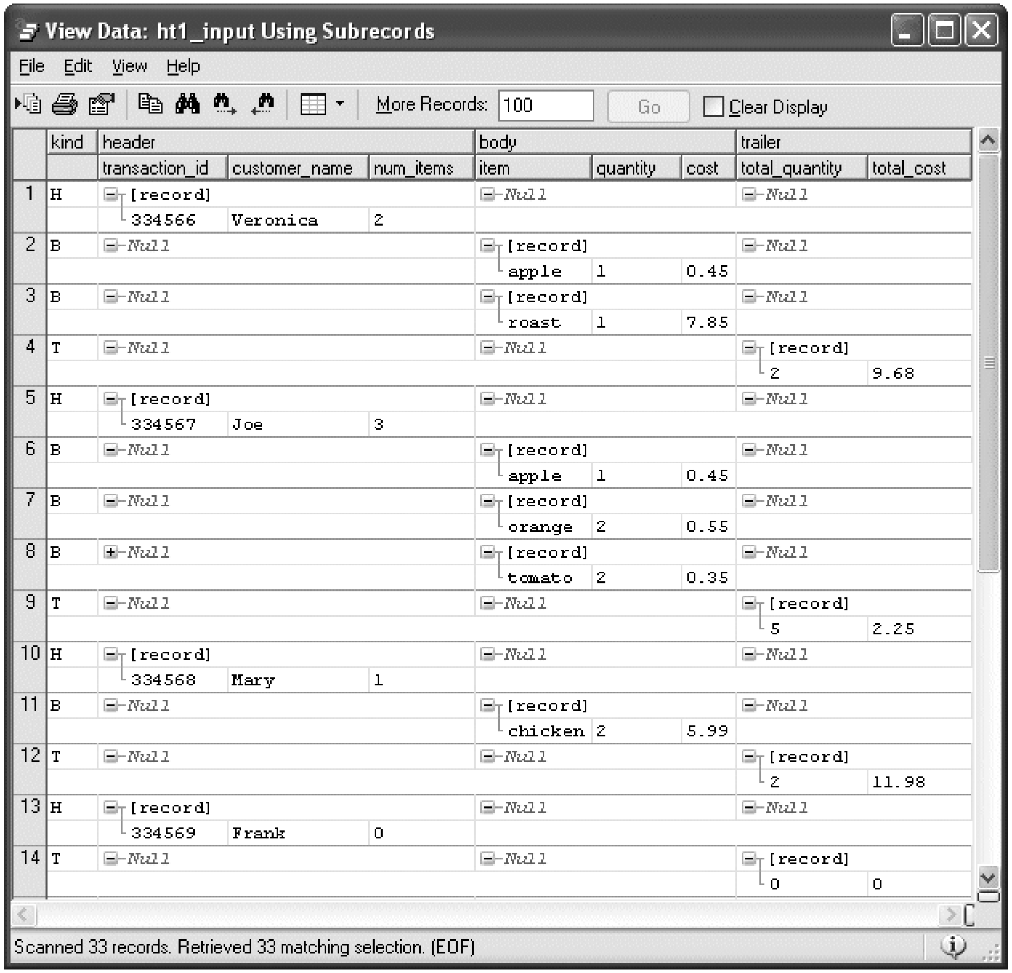
ht1_input_subrec_types.dml

```
type header_type =
  record
    decimal(',') transaction_id;
    string(',') customer_name;
    decimal('\n') num_items;
  end;

type body_type =
  record
    string(',') item;
    decimal(',') quantity;
    decimal('\n') cost;
  end;

type trailer_type =
  record
    decimal(',') total_quantity;
    decimal('\n') total_cost;
  end;
```

ht1_input_subrec.dml

```
include "/~$AI_DML/ht1_input_subrec_types.dml";
metadata type =
  record
    string(1) kind; /* One of H, B or T */
    if (kind=="H") header_type header;
    if (kind=="B") body_type body;
    if (kind=="T") trailer_type trailer;
  end;
```

For each customer transaction, this format also results in at least two records (one header and one trailer) with three subrecords in each record, and NULL values for each subrecord that has no content. To see this, view the data in **ht1_input Using Subrecords**:



View Data: ht1_input Using Subrecords

File Edit View Help

More Records: 100 Go Clear Display

	kind	header			body			trailer	
		transaction_id	customer_name	num_items	item	quantity	cost	total_quantity	total_cost
1	H	[record]			Null			Null	
		334566	Veronica	2					
2	B	Null			[record]			Null	
					apple	1	0.45		
3	B	Null			[record]			Null	
					roast	1	7.85		
4	T	Null			Null			[record]	
								2	9.68
5	H	[record]			Null			Null	
		334567	Joe	3					
6	B	Null			[record]			Null	
					apple	1	0.45		
7	B	Null			[record]			Null	
					orange	2	0.55		
8	B	Null			[record]			Null	
					tomato	2	0.35		
9	T	Null			Null			[record]	
								5	2.25
10	H	[record]			Null			Null	
		334568	Mary	1					
11	B	Null			[record]			Null	
					chicken	2	5.99		
12	T	Null			Null			[record]	
								2	11.98
13	H	[record]			Null			Null	
		334569	Frank	0					
14	T	Null			Null			[record]	
								0	0

Scanned 33 records. Retrieved 33 matching selection. (EOF)

Vectors

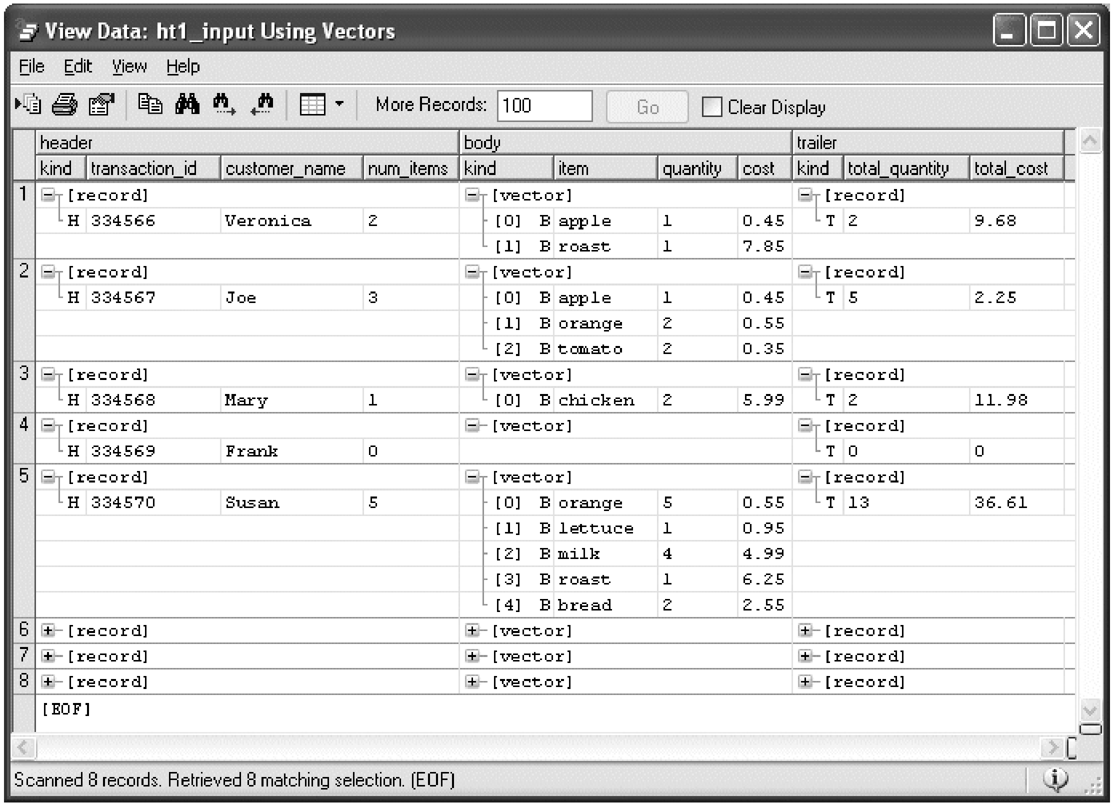
In some cases, a header/body/trailer group can be treated as a single record, with a vector used to hold the body (sub)records.

The records in **ht1_input Using Vectors** (in **ht_data.mp**) are represented in a vector format, in which the number of body records is determined by the value of the **num_items** field in the header record:

ht1_input_vector.dml

```
record
  record
    string(1) kind; /* Should be H */
    decimal(',') transaction_id;
    string(',') customer_name;
    decimal('\n') num_items;
  end header;
  record
    string(1) kind; /* Should be B */
    string(',') item;
    decimal(',') quantity;
    decimal('\n') cost;
  end body[header.num_items];
  record
    string(1) kind; /* Should be T */
    decimal(',') total_quantity;
    decimal('\n') total_cost;
  end trailer;
end
```

As you can see by viewing the data in **ht1_input Using Vectors**, this format results in just one record for each customer transaction: it consists of the header data, a vector of body data, and one trailer record, and it has no NULL fields:



View Data: ht1_input Using Vectors

File Edit View Help

More Records: 100 Go Clear Display

header				body					trailer		
kind	transaction_id	customer_name	num_items	kind		item	quantity	cost	kind	total_quantity	total_cost
1 [record]				[vector]					[record]		
H	334566	Veronica	2	[0]	B	apple	1	0.45	T	2	9.68
				[1]	B	roast	1	7.85			
2 [record]				[vector]					[record]		
H	334567	Joe	3	[0]	B	apple	1	0.45	T	5	2.25
				[1]	B	orange	2	0.55			
				[2]	B	tomato	2	0.35			
3 [record]				[vector]					[record]		
H	334568	Mary	1	[0]	B	chicken	2	5.99	T	2	11.98
4 [record]				[vector]					[record]		
H	334569	Frank	0						T	0	0
5 [record]				[vector]					[record]		
H	334570	Susan	5	[0]	B	orange	5	0.55	T	13	36.61
				[1]	B	lettuce	1	0.95			
				[2]	B	milk	4	4.99			
				[3]	B	roast	1	6.25			
				[4]	B	bread	2	2.55			
6 [record]				[vector]					[record]		
7 [record]				[vector]					[record]		
8 [record]				[vector]					[record]		
[EOF]											

Scanned 8 records. Retrieved 8 matching selection. (EOF)

Processing the data

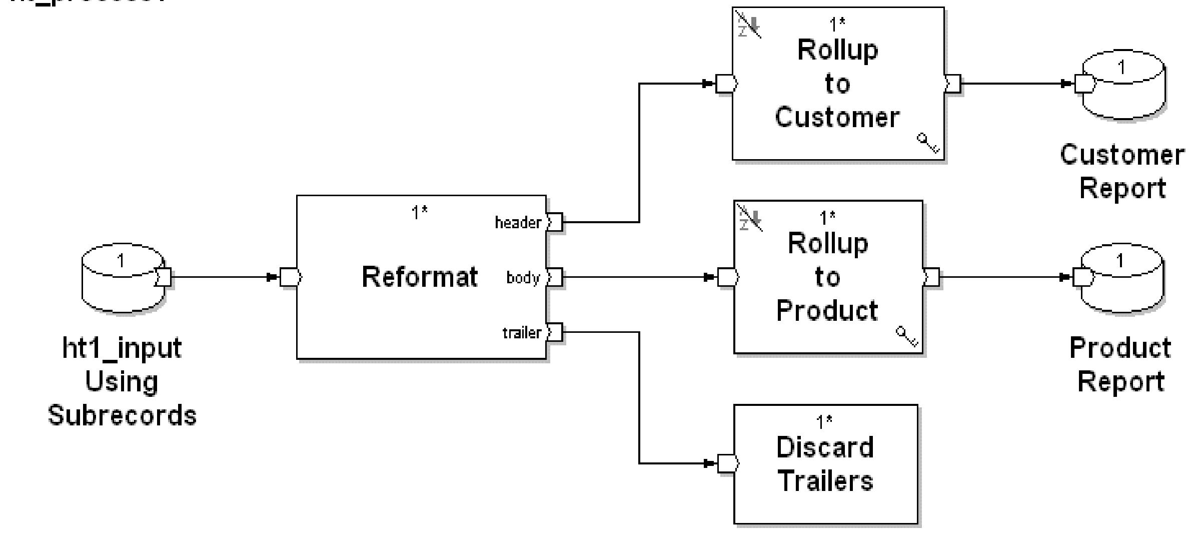
This section describes ways of generating summaries of data stored in subrecord or vector format:

- Generating separate summaries of header and body subrecords (next)
- Distributing header information to body records (page 165)
- Normalizing vector data to header and body records (page 167)

Generating separate summaries of header and 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**):

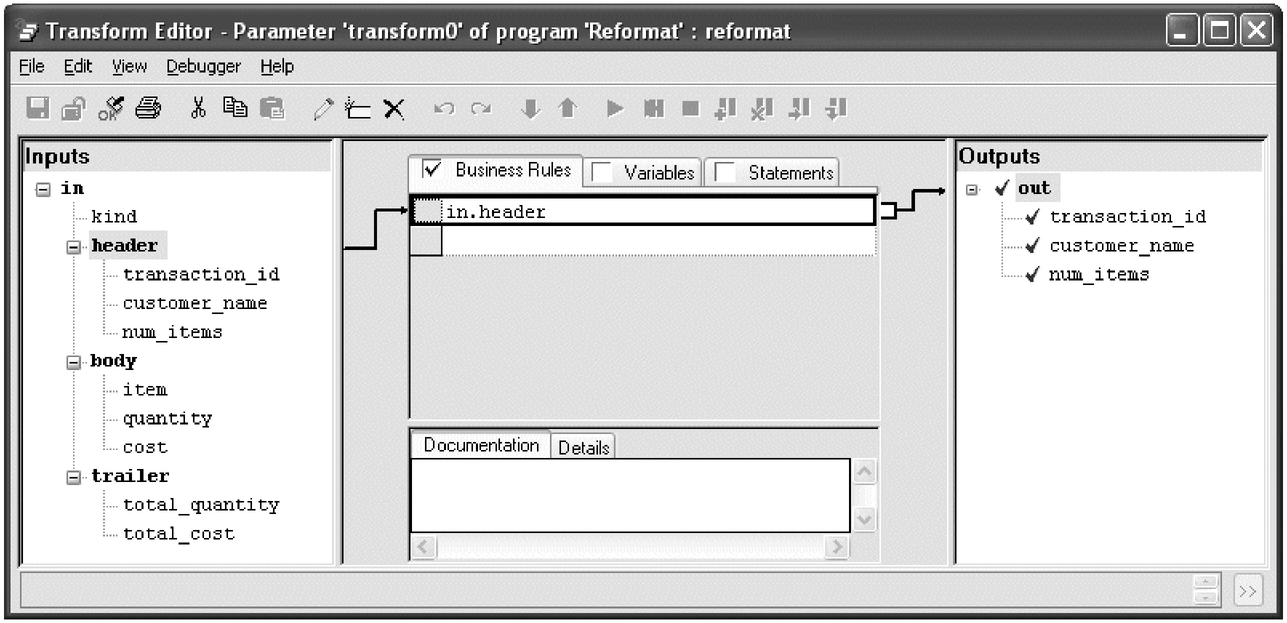
```
out :: output_index(in) =
begin
  out :: if (in.kind == 'H') 0 else if (in.kind == 'B') 1
         else 2;
end;
```

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

transform0	**transform1**	**transform2**

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

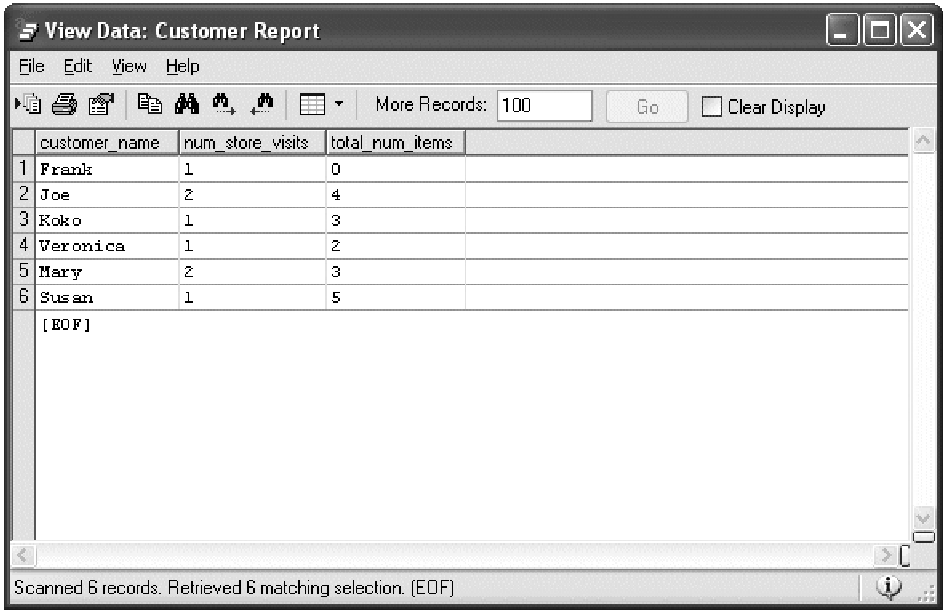
```
out::rollup(in) =
begin
  out.customer_name :: in.customer_name;
  out.num_store_visits :: count(1);
  out.total_num_items :: sum(in.num_items);
end;
```

- 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:



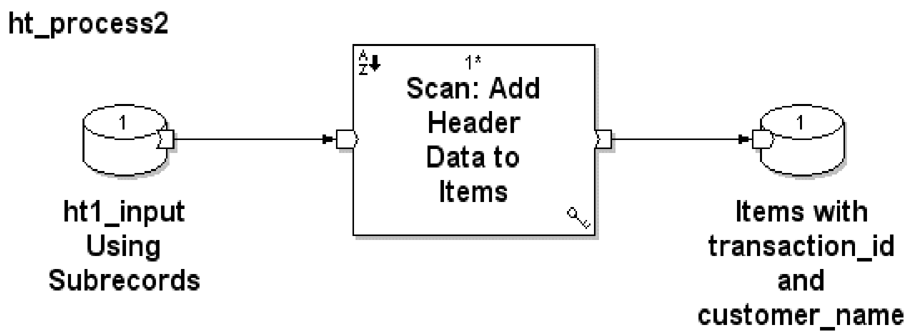
	customer_name	num_store_visits	total_num_items
1	Frank	1	0
2	Joe	2	4
3	Koko	1	3
4	Veronica	1	2
5	Mary	2	3
6	Susan	1	5

[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 and generate a dataset containing records of a single type that combines information from header and body records.


ht_process2
1
ht1_input Using Subrecords
1*
Scan: Add Header Data to Items
1
Items with transaction_id and customer_name

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;
   ```

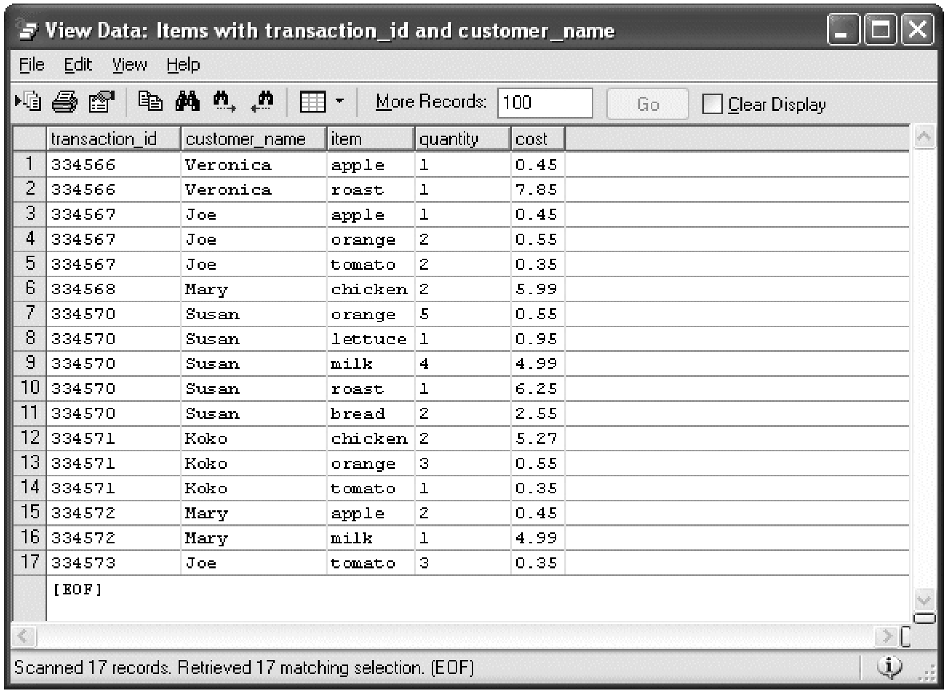
   This causes the current **transaction_id** and **customer_name** fields from the header record to be written to the temporary record for each subsequent body and trailer record in the key group initialized for that header record.

   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:

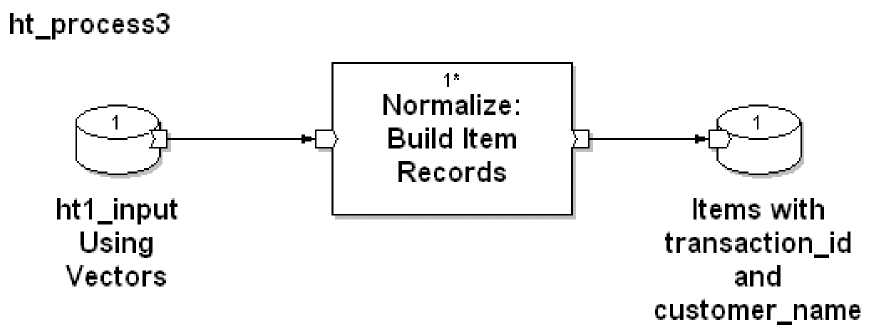


	transaction_id	customer_name	item	quantity	cost
1	334566	Veronica	apple	1	0.45
2	334566	Veronica	roast	1	7.85
3	334567	Joe	apple	1	0.45
4	334567	Joe	orange	2	0.55
5	334567	Joe	tomato	2	0.35
6	334568	Mary	chicken	2	5.99
7	334570	Susan	orange	5	0.55
8	334570	Susan	lettuce	1	0.95
9	334570	Susan	milk	4	4.99
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 the same output as **ht_process2.mp** (a dataset containing records of a single type that combines information from header and body records) but uses a NORMALIZE to process the vector view of the data:



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;
```

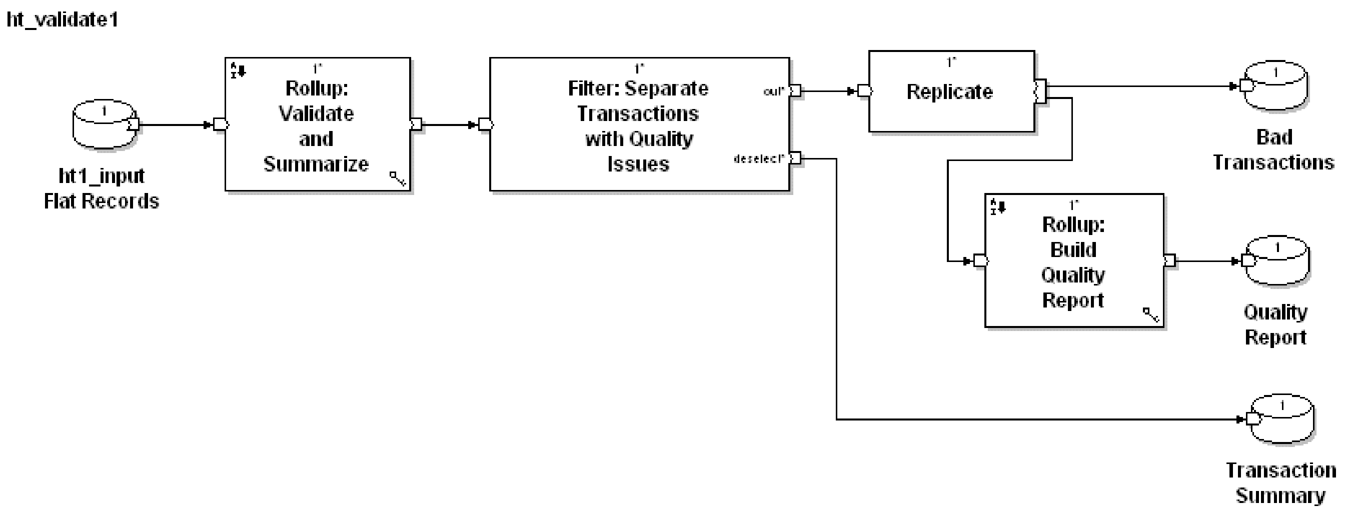
Validating the input data

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

- Validating flat input data (next)
- Validating vector input data (page 174)

Validating flat input data

The graph **ht_validate1.mp** shows how to validate the flat representation of the input data described on page 153:



ht_validate1
ht1_input Flat Records
Rollup: Validate and Summarize
Filter: Separate Transactions with Quality Issues
out*
deselect*
Replicate
Bad Transactions
Rollup: Build Quality Report
Quality Report
Transaction Summary

NULL VALUES IN ROLLUP AGGREGATION FUNCTIONS

In the conditional representation used for the flat data (see "Flat records" on page 153), fields that are not part of a given **kind** of record are NULL. (For example, the **quantity** field of all header and 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.

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.

The components of **ht_validate1.mp** do the following:

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

   a. 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;
```

   b. 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);
```

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

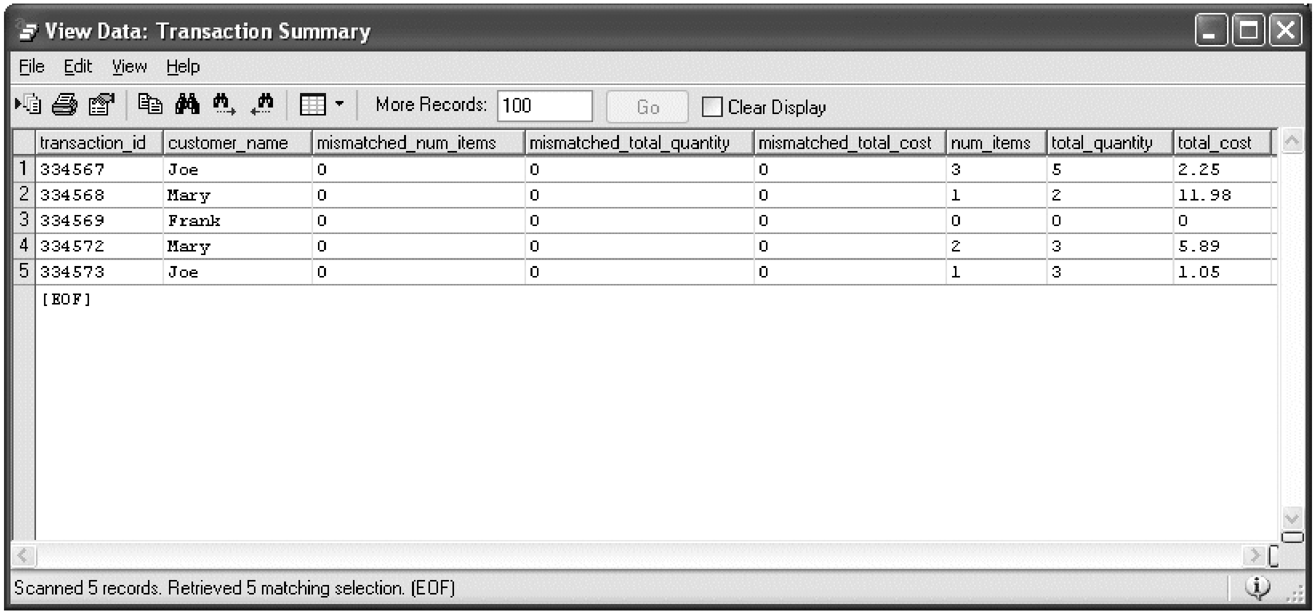
```
  out.transaction_id :: first(in.transaction_id);
  out.customer_name :: first(in.customer_name);
  out.mismatched_num_items :: actual_num_items !=
       first(in.num_items);
  out.mismatched_total_quantity :: actual_total_quantity !=
       last(in.total_quantity);
  out.mismatched_total_cost ::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

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:



View Data: Transaction Summary

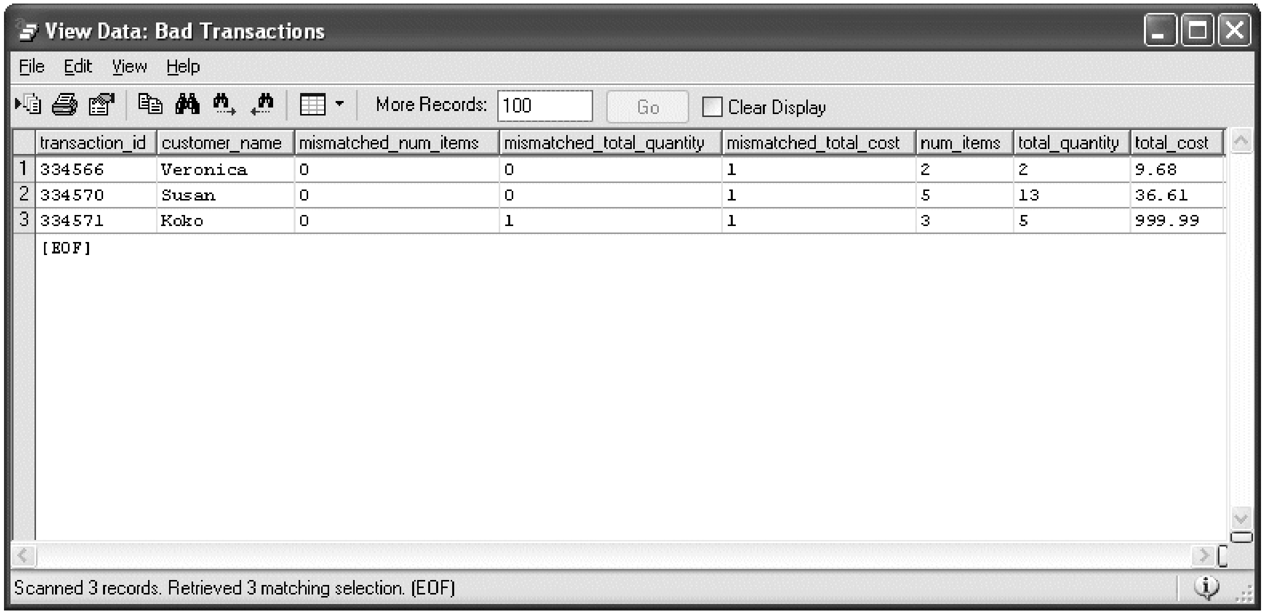
More Records: 100 | Go | Clear Display

	transaction_id	customer_name	mismatched_num_items	mismatched_total_quantity	mismatched_total_cost	num_items	total_quantity	total_cost
1	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**).



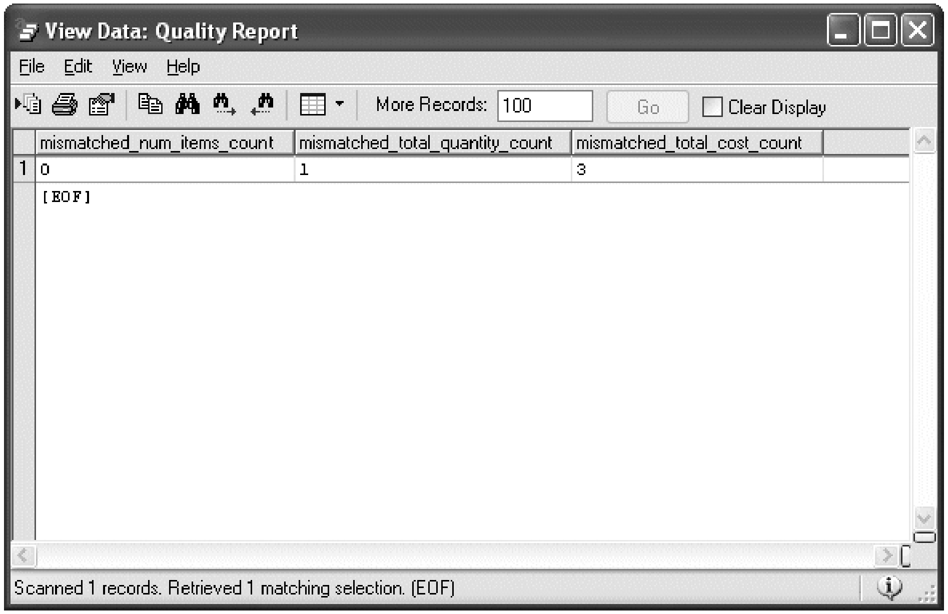
	transaction_id	customer_name	mismatched_num_items	mismatched_total_quantity	mismatched_total_cost	num_items	total_quantity	total_cost
1	334566	Veronica	0	0	1	2	2	9.68
2	334570	Susan	0	0	1	5	13	36.61
3	334571	Koko	0	1	1	3	5	999.99
	[EOF]							

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:

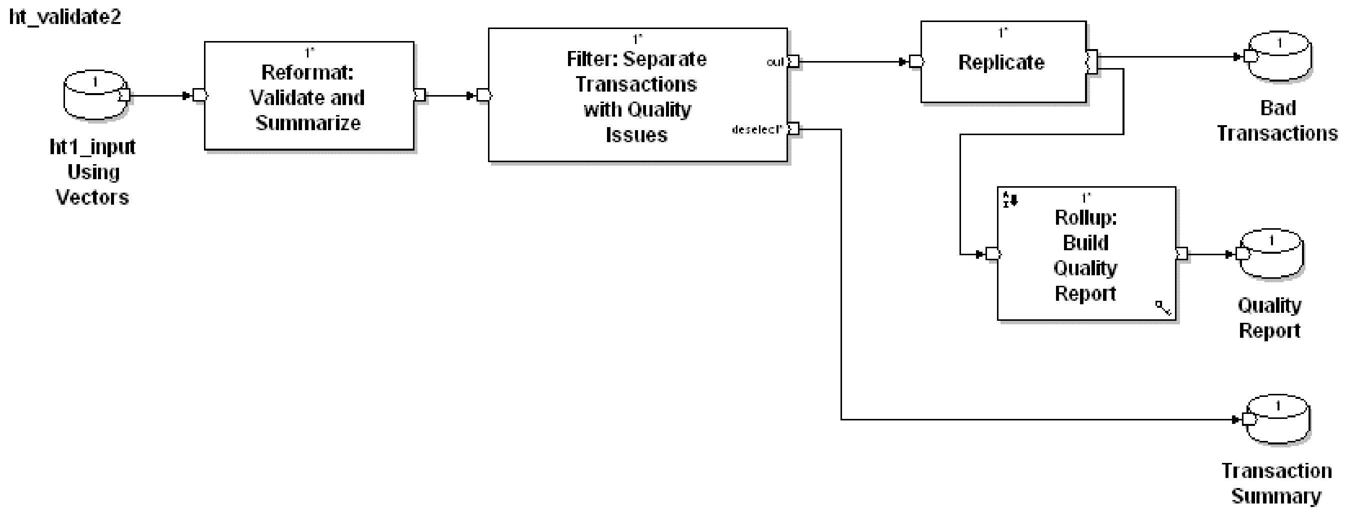


	mismatched_num_items_count	mismatched_total_quantity_count	mismatched_total_cost_count
1	0	1	3
	[EOF]		

Scanned 1 records. Retrieved 1 matching selection. (EOF)

Validating vector input data

The graph **ht_validate2.mp** shows how to validate the vector representation of the input data described on page 158:


ht_validate2
ht1_input Using Vectors
Reformat: Validate and Summarize
Filter: Separate Transactions with Quality Issues
out
deselect
Replicate
Bad Transactions
Rollup: Build Quality Report
Quality Report
Transaction Summary

This graph is identical to **ht_validate1.mp** (page 169) except that instead of using a ROLLUP (which processes groups of records), this graph uses a REFORMAT, because each group of header, body, trailer is in a single record.

The transform in the **Reformat: Validate and Summarize** component does the following:

1. Creates variables to hold the correct values for the total quantity of each item purchased and the total cost for each store visit (that is, actual values that the graph will compute) and initializes them to zero:

```
out :: reformat(in) =
begin
  let decimal('') actual_total_quantity = 0;
  let decimal('') actual_total_cost = 0;
```

2. Uses a **for** loop to iterate over the elements of each body vector and accumulate **actual_total_quantity** and **actual_total_cost**:

```
  let int i;

  for (i, i < length_of(in.body))
    begin
      actual_total_quantity = actual_total_quantity +
                              in.body[i].quantity;
      actual_total_cost = actual_total_cost + (in.body[i].quantity *
                              in.body[i].cost);
    end
```

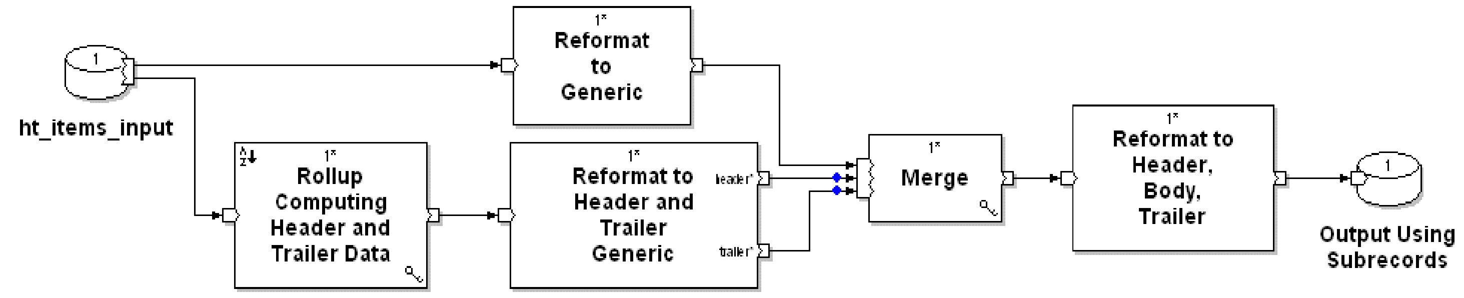
3. Outputs fields from the header and trailer records, along with boolean values indicating whether any of the calculated values in the original data are incorrect:

```
  out.transaction_id :: in.header.transaction_id;
  out.customer_name :: in.header.customer_name;
  out.mismatched_total_quantity :: actual_total_quantity !=
        in.trailer.total_quantity;
  out.mismatched_total_cost :: actual_total_cost !=
        in.trailer.total_cost;
  out.num_items :: in.header.num_items;
  out.total_quantity :: in.trailer.total_quantity;
  out.total_cost :: in.trailer.total_cost;
end;
```

Advanced topic: Building flat data into header/body/trailer data subrecord format

In some situations you may need to transform data from a homogeneous form into a header, body, trailer form — for example, when the results of a graph are to be fed back to a legacy system. This section describes **ht_build.mp**, a graph that converts data from flat format into subrecord format:

ht_build


ht_items_input
1
Reformat to Generic
Rollup Computing Header and Trailer Data
Reformat to Header and Trailer Generic
header
trailer
Merge
Reformat to Header, Body, Trailer
Output Using Subrecords

The record format of the input data is defined by **ht_items.dml** as follows:

```
record
  decimal(',') transaction_id;
  string(',') customer_name;
  string(',') item;
  decimal(',') quantity;
  decimal('\n') cost;
end
```

The preceding format does not include the following fields from the headers and trailers of the graphs described earlier in this chapter:

```
decimal(',') num_items; /* from the header */
decimal(',') total_quantity; /* from the trailer */
decimal('\n') total_cost; /* from the trailer */
```

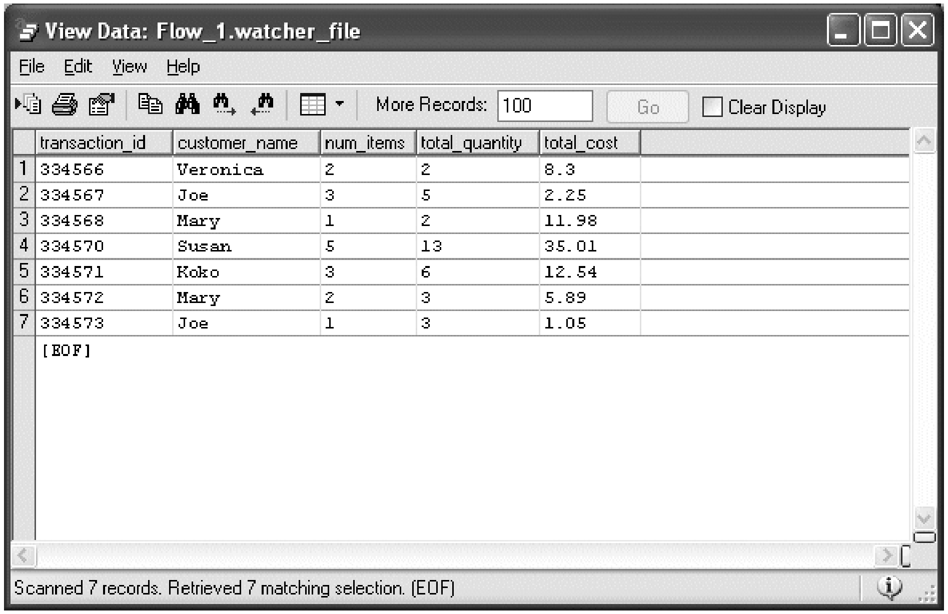
The graph **ht_build.mp** computes those values as part of its processing. Specifically, it does the following:

1. Rolls up the input records to compute the data for the above three fields (page 178).
2. Reformats the data to generic records that contain all eight fields, plus a **kind** field with a value of **H**, **B**, or **T** to designate each record as a header, body, or trailer; and an **ordering** field with a value of **1**, **2**, or **3** (page 179). The graph does this in two steps:
    a. Reformats the body records
    b. Reformats the rolled-up header records and trailer records
3. Merges the reformatted records, assembling them in header, body, trailer order (page 183).
4. Reformats the merged records, using a conditional record format to populate the header, body, and trailer fields appropriately in the output data (page 184).

Rolling up to compute header and trailer data

The **Rollup Computing Header and Trailer Data** component calculates (for each group of records with the same **{transaction_id}**) the number of different types of items purchased, the total quantity, and the total cost. Following are its transform and output:

```
out::rollup(in) =
begin
  out.transaction_id :: in.transaction_id;
  out.customer_name :: in.customer_name;
  out.num_items :: count(1);
  out.total_quantity :: sum(in.quantity);
  out.total_cost :: sum(in.quantity * in.cost);
end;
```



	transaction_id	customer_name	num_items	total_quantity	total_cost
1	334566	Veronica	2	2	8.3
2	334567	Joe	3	5	2.25
3	334568	Mary	1	2	11.98
4	334570	Susan	5	13	35.01
5	334571	Koko	3	6	12.54
6	334572	Mary	2	3	5.89
7	334573	Joe	1	3	1.05

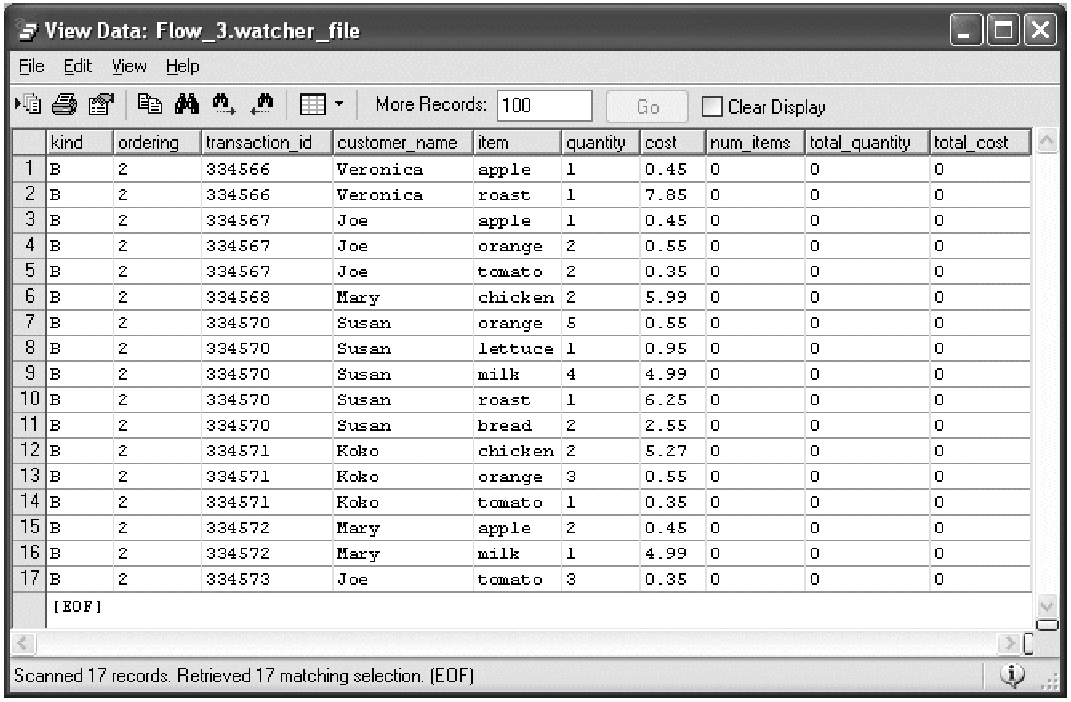
```
[EOF]
```

Scanned 7 records. Retrieved 7 matching selection. (EOF)

Reformatting to generic records

The **Reformat to Generic** component reformats each body record, adding a **kind** field with a value of **B** and an **ordering** field with a value of **2**. Following are the component's transform and output:

```
body::reformat(in) =
begin
  body.kind :: 'B';
  body.ordering :: 2;
  body.* :: in.*;
end;
```



	kind	ordering	transaction_id	customer_name	item	quantity	cost	num_items	total_quantity	total_cost
1	B	2	334566	Veronica	apple	1	0.45	0	0	0
2	B	2	334566	Veronica	roast	1	7.85	0	0	0
3	B	2	334567	Joe	apple	1	0.45	0	0	0
4	B	2	334567	Joe	orange	2	0.55	0	0	0
5	B	2	334567	Joe	tomato	2	0.35	0	0	0
6	B	2	334568	Mary	chicken	2	5.99	0	0	0
7	B	2	334570	Susan	orange	5	0.55	0	0	0
8	B	2	334570	Susan	lettuce	1	0.95	0	0	0
9	B	2	334570	Susan	milk	4	4.99	0	0	0
10	B	2	334570	Susan	roast	1	6.25	0	0	0
11	B	2	334570	Susan	bread	2	2.55	0	0	0
12	B	2	334571	Koko	chicken	2	5.27	0	0	0
13	B	2	334571	Koko	orange	3	0.55	0	0	0
14	B	2	334571	Koko	tomato	1	0.35	0	0	0
15	B	2	334572	Mary	apple	2	0.45	0	0	0
16	B	2	334572	Mary	milk	1	4.99	0	0	0
17	B	2	334573	Joe	tomato	3	0.35	0	0	0
	[EOF]									

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

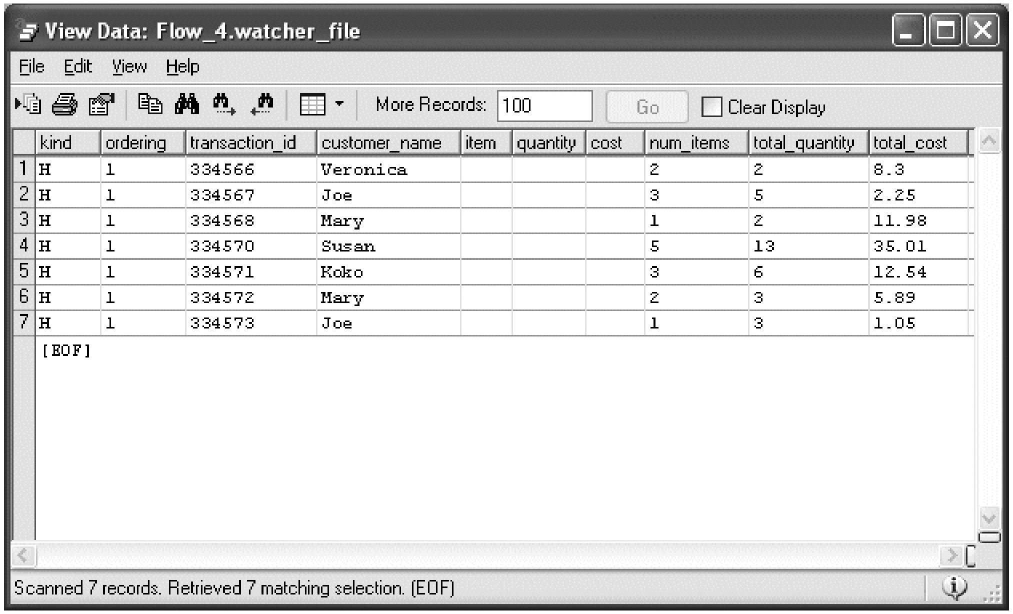
The **Reformat to Header and Trailer Generic** component has two transforms: one each to reformat the header and trailer records:

- **transform0** — Adds a **kind** field with a value of **H** and an **ordering** field with a value of **1**
- **transform1** — Adds a **kind** field with a value of **T** and an **ordering** field with a value of **3**

Following are the two transforms and their output:

transform0 (headers)

```
header::reformat(in) =
begin
  header.kind :: 'H';
  header.ordering :: 1;
  header.* :: in.*;
end;
```


View Data: Flow_4.watcher_file

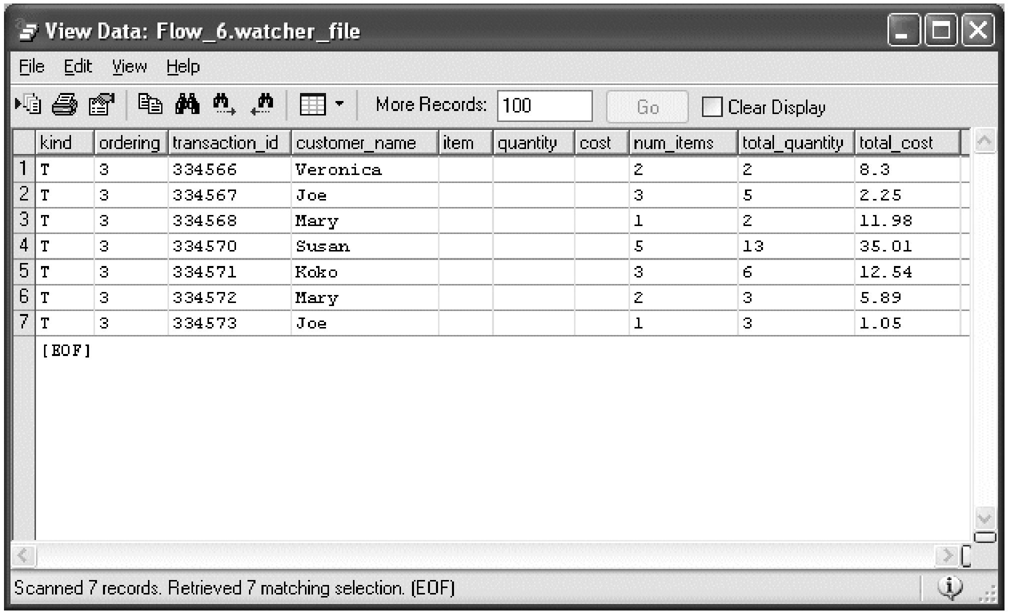
	kind	ordering	transaction_id	customer_name	item	quantity	cost	num_items	total_quantity	total_cost
1	H	1	334566	Veronica				2	2	8.3
2	H	1	334567	Joe				3	5	2.25
3	H	1	334568	Mary				1	2	11.98
4	H	1	334570	Susan				5	13	35.01
5	H	1	334571	Koko				3	6	12.54
6	H	1	334572	Mary				2	3	5.89
7	H	1	334573	Joe				1	3	1.05

[EOF]

Scanned 7 records. Retrieved 7 matching selection. [EOF]

transform1 (trailers)

```
trailer::reformat(in) =
begin
  trailer.kind :: 'T';
  trailer.ordering :: 3;
  trailer.* :: in.*;
end;
```



View Data: Flow_6.watcher_file

More Records: 100

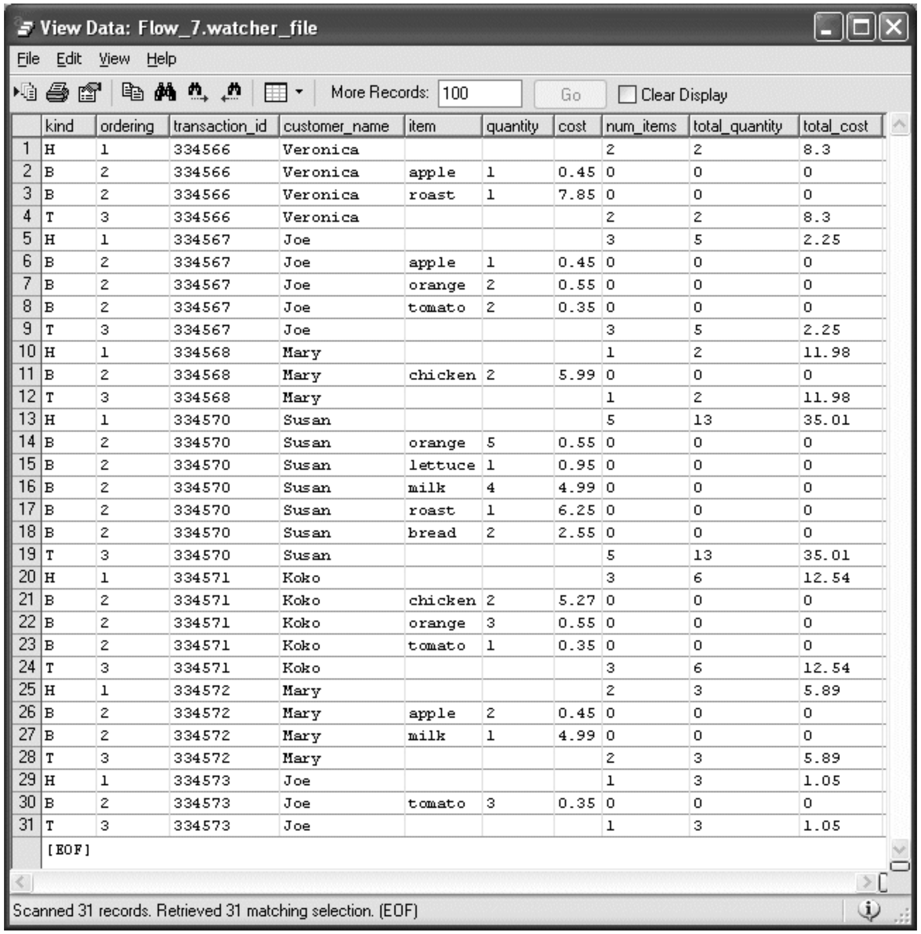
	kind	ordering	transaction_id	customer_name	item	quantity	cost	num_items	total_quantity	total_cost
1	T	3	334566	Veronica				2	2	8.3
2	T	3	334567	Joe				3	5	2.25
3	T	3	334568	Mary				1	2	11.98
4	T	3	334570	Susan				5	13	35.01
5	T	3	334571	Koko				3	6	12.54
6	T	3	334572	Mary				2	3	5.89
7	T	3	334573	Joe				1	3	1.05

[EOF]

Scanned 7 records. Retrieved 7 matching selection. [EOF]

Merging to header, body, trailer order

The MERGE component merges the records by using a two-part key — **{transaction_id; ordering}** — to assemble the records in the right order:



	kind	ordering	transaction_id	customer_name	item	quantity	cost	num_items	total_quantity	total_cost
1	H	1	334566	Veronica				2	2	8.3
2	B	2	334566	Veronica	apple	1	0.45	0	0	0
3	B	2	334566	Veronica	roast	1	7.85	0	0	0
4	T	3	334566	Veronica				2	2	8.3
5	H	1	334567	Joe				3	5	2.25
6	B	2	334567	Joe	apple	1	0.45	0	0	0
7	B	2	334567	Joe	orange	2	0.55	0	0	0
8	B	2	334567	Joe	tomato	2	0.35	0	0	0
9	T	3	334567	Joe				3	5	2.25
10	H	1	334568	Mary				1	2	11.98
11	B	2	334568	Mary	chicken	2	5.99	0	0	0
12	T	3	334568	Mary				1	2	11.98
13	H	1	334570	Susan				5	13	35.01
14	B	2	334570	Susan	orange	5	0.55	0	0	0
15	B	2	334570	Susan	lettuce	1	0.95	0	0	0
16	B	2	334570	Susan	milk	4	4.99	0	0	0
17	B	2	334570	Susan	roast	1	6.25	0	0	0
18	B	2	334570	Susan	bread	2	2.55	0	0	0
19	T	3	334570	Susan				5	13	35.01
20	H	1	334571	Koko				3	6	12.54
21	B	2	334571	Koko	chicken	2	5.27	0	0	0
22	B	2	334571	Koko	orange	3	0.55	0	0	0
23	B	2	334571	Koko	tomato	1	0.35	0	0	0
24	T	3	334571	Koko				3	6	12.54
25	H	1	334572	Mary				2	3	5.89
26	B	2	334572	Mary	apple	2	0.45	0	0	0
27	B	2	334572	Mary	milk	1	4.99	0	0	0
28	T	3	334572	Mary				2	3	5.89
29	H	1	334573	Joe				1	3	1.05
30	B	2	334573	Joe	tomato	3	0.35	0	0	0
31	T	3	334573	Joe				1	3	1.05
	[EOF]									

Scanned 31 records. Retrieved 31 matching selection. (EOF)

Reformatting to subrecord format

The **Reformat to Header, Body, Trailer** component reformats the data into the format defined in **ht1_input_subrec.dml**.

Geographic calculations

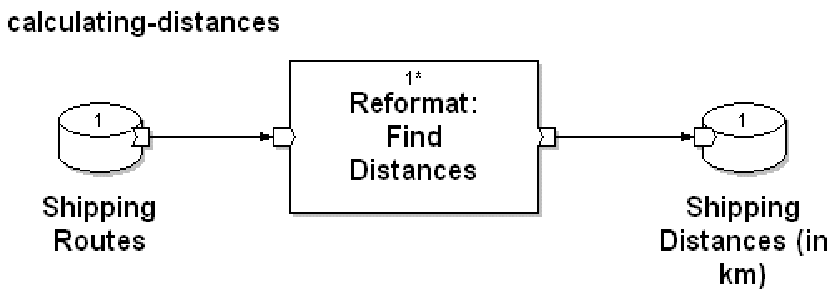
This chapter presents three graphs (provided in the **distances** sandbox) that show how to calculate and use distances between geographic locations. Distance calculation is of obvious importance in transportation and shipping, but it can also be useful in industries such as banking — for example, to find the bank branch nearest to a given customer's location.

The chapter starts with a graph that uses trigonometric functions in DML to calculate the distance between two points on the globe (page 186). Then it presents two graphs that solve the "nearest branch" problem mentioned above:

- The first graph does this in a simple, "brute-force" manner, calculating and comparing the distance between every bank branch and every customer location (page 193).
- The second graph uses a more sophisticated and more efficient algorithm, which results in much higher performance (page 197).

Calculating distances between locations

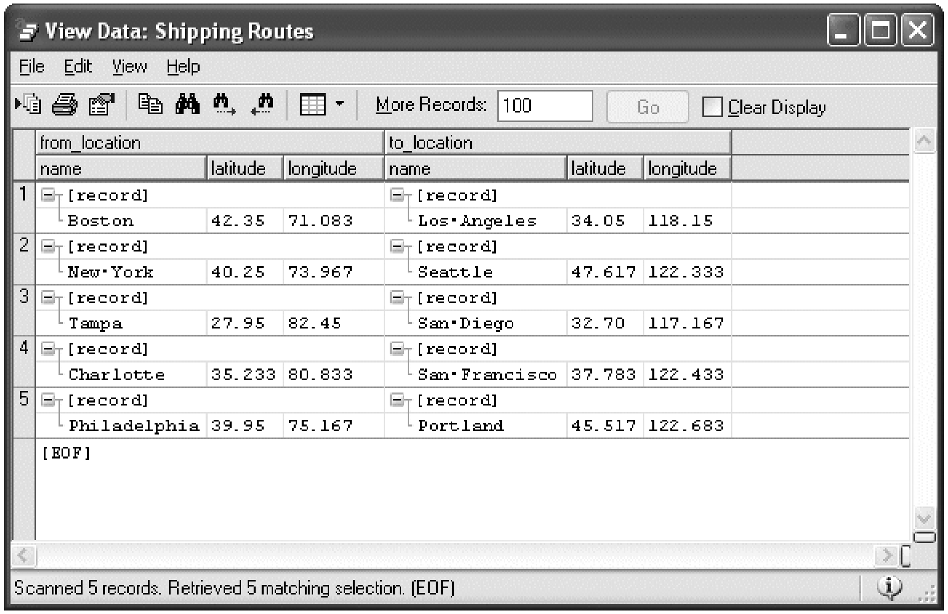
The graph **calculating-distances.mp** uses DML to calculate the distance between two geographic locations (also known as the *great circle* distance), given the latitude and longitude of each location:



The graph also shows how to:

- Incorporate a package of reusable functions by using an **include** statement
- Use built-in trigonometric functions
- Use a fixed-point decimal type in the output record format to eliminate the need for explicit formatting of numeric results

The INPUT FILE component contains the latitude and longitude of five pairs of cities in the following format:


View Data: Shipping Routes
File Edit View Help
More Records: 100 Go Clear Display

	from_location			to_location		
	name	latitude	longitude	name	latitude	longitude
1	[record]			[record]		
	Boston	42.35	71.083	Los·Angeles	34.05	118.15
2	[record]			[record]		
	New·York	40.25	73.967	Seattle	47.617	122.333
3	[record]			[record]		
	Tampa	27.95	82.45	San·Diego	32.70	117.167
4	[record]			[record]		
	Charlotte	35.233	80.833	San·Francisco	37.783	122.433
5	[record]			[record]		
	Philadelphia	39.95	75.167	Portland	45.517	122.683
	[EOF]					

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

The record format is specified in **shipping-routes.dml**:

```
record
  record
    string(',') name; /* from_location name*/
    decimal(',') latitude; /* Degrees latitude*/
    decimal(',') longitude; /* Degrees longitude*/
  end from_location;
```

```
    record
      string(',') name; /* to_location name*/
      decimal(',') latitude; /* Degrees latitude*/
      decimal('\n') longitude; /* Degrees longitude*/
    end to_location;
  end;
```

The OUTPUT FILE contains the names of each pair of cities and the distances between them. The record format for this data is in **route-lengths.dml**:

```
record
  string(',') route;
  decimal(9.2) distance;
end;
```

Haversine formula and great circle distance

The REFORMAT component in **calculating-distances.mp** uses the Haversine formula to calculate the great circle distance (in kilometers) between the two cities in each of the five pairs. The great circle distance — the shortest distance between two points on a sphere — takes the curvature of the earth into account. Since the earth is not a perfect sphere, the calculation is approximate (but accurate to within 0.18%).

The REFORMAT calculates the distances as follows:

1. Incorporates the transform file **great-circle-distance.xfr** by using an **include** statement.

```
include "/~$AI_XFR/great-circle-distance.xfr";
```

The included transform file uses built-in trigonometric functions to calculate the distances:

```
constant real(8) radius = 6367.0;
                          /* Earth radius in kilometers */

constant real(8) to_radians = 0.0174532925;
                         /* Degrees per radian */
```

```
out::haversine(x) =
begin
  out :: 0.5 * (1.0 - math_cos(x));
end;

out::great_circle_distance(a_lat, a_long, z_lat, z_long) =
begin
  /* Convert inputs in degrees to radians: */
  let real(8) a_lat_r = to_radians * a_lat;
  let real(8) z_lat_r = to_radians * z_lat;
  let real(8) a_long_r = to_radians * a_long;
  let real(8) z_long_r = to_radians * z_long;

  /* x should always be between 0 and 1, except for
     floating-point roundoff problems that might knock it
     out of that range: */

  let real(8) x = haversine(a_lat_r - z_lat_r) +
      math_cos(a_lat_r) * math_cos(z_lat_r) *
      haversine(a_long_r - z_long_r);

  /* Compute result: */
  out :1: if (x <= 1.0 and x >= 0)
          radius * 2 * math_asin(math_sqrt(x));
  out :2: if (x > 1)
          radius * 2 * math_asin(1.0);
  out :3: 0;
end;
```

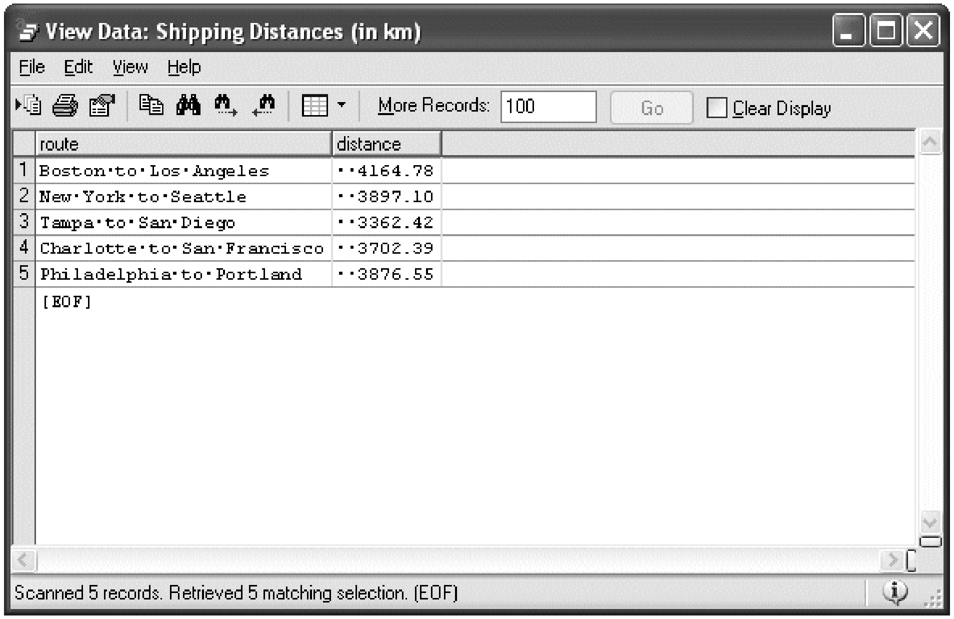
2. Outputs strings containing the names of each pair of cities, followed by the calculated distance between them:

```
include "/~$AI_XFR/great-circle-distance.xfr";

out::reformat(in) =
begin
  out.route :: string_concat(in.from_location.name,  " to ",
                             in.to_location.name);
  out.distance ::
    great_circle_distance(in.from_location.latitude,
                          in.from_location.longitude,
                          in.to_location.latitude,
                          in.to_location.longitude);
end;
```

Simplifying the code and controlling the output by using the right data type

Because the graph uses the fixed-point decimal type **decimal(9.2)** for **distance** in the output record format rather than a floating-point decimal type (such as **decimal(9)** or **decimal('\n')**), it generates easily readable output data with no additional code to round the results of the floating-point arithmetic.

If you run **calculating-distances.mp**, you should see the following output:



	route	distance
1	Boston·to·Los·Angeles	··4164.78
2	New·York·to·Seattle	··3897.10
3	Tampa·to·San·Diego	··3362.42
4	Charlotte·to·San·Francisco	··3702.39
5	Philadelphia·to·Portland	··3876.55
	[EOF]	

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

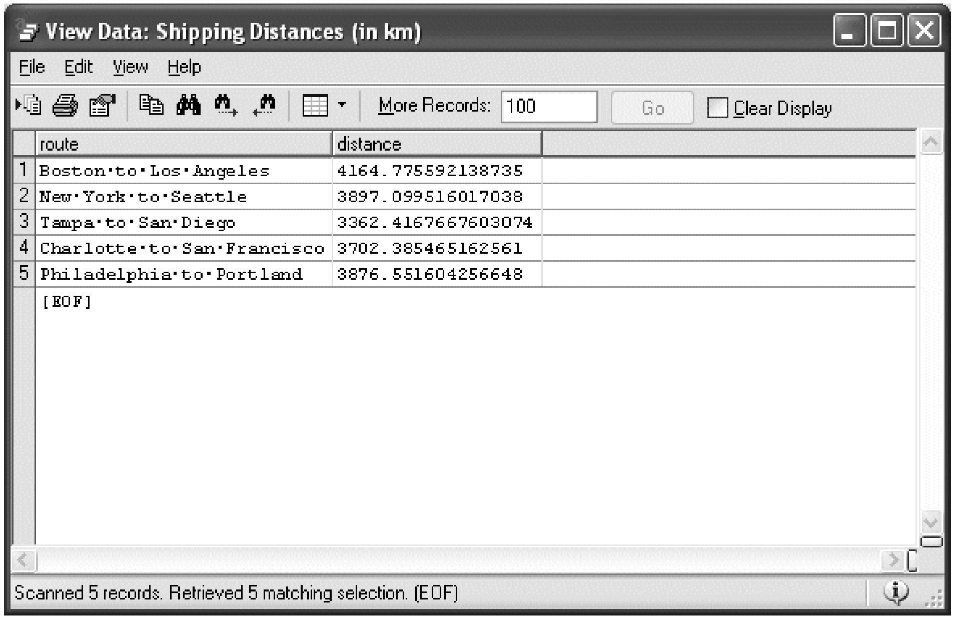
To see the effect of using a floating-point decimal type in the output record format for this graph:

1. Save **calculating-distances.mp** under a different name (such as **my-calculating-distances.mp**).
2. On the **Ports** tab of the **Properties** dialog of the **Shipping Distances (in km)** output file, select **Embed** and click **Yes** when asked whether you would like to use the contents of the file **route_lengths.dml** as the embedded value. (By doing this, you prevent this experiment from overwriting the record format for the original graph.)
3. Change the record format of the **distance** field in the output file to the following and click **OK**:

```
    decimal('\n') distance;
```

4. Run the graph and view the results. You should see the output below.

For delimited decimal types, the transform outputs all significant digits, in this case 12 or 13 after the decimal point, which is not very useful in this application; the distances would be specified to the level of a tiny fraction of a millimeter. To avoid this, use a fixed-point decimal type to explicitly specify an appropriate number of digits after the decimal point.

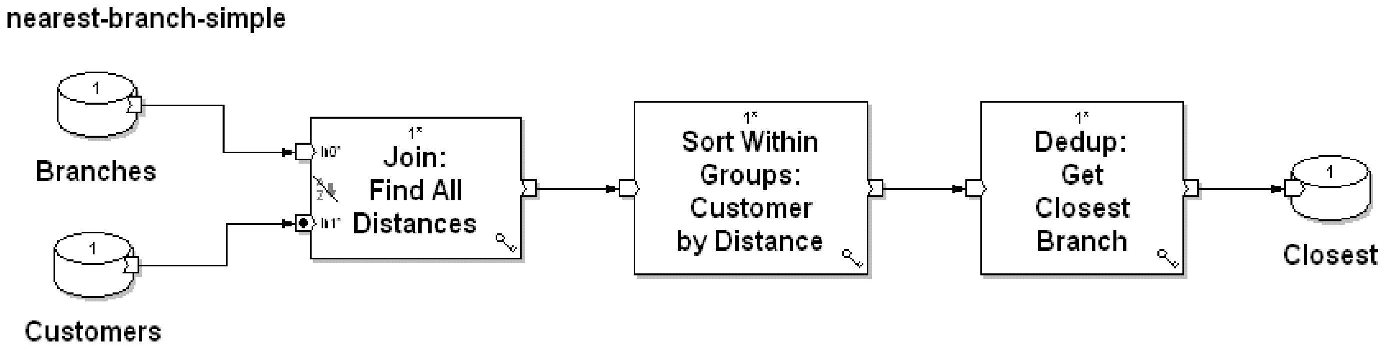


```
  route                          distance
1 Boston·to·Los·Angeles          4164.775592138735
2 New·York·to·Seattle            3897.099516017038
3 Tampa·to·San·Diego             3362.4167667603074
4 Charlotte·to·San·Francisco     3702.385465162561
5 Philadelphia·to·Portland       3876.551604256648
  [EOF]
```

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

Simple calculation of the nearest branch

The graph **nearest-branch-simple.mp** finds the nearest bank branch to each of a set of customers by using a simple and straightforward approach that may be acceptably efficient if the number of customers and branches is modest: it computes the distance to every branch for every customer and selects the shortest one.



Here is a description of what **nearest-branch-simple.mp** does:

1. Uses INPUT FILE components that do the following:

a. Specify the record formats for the locations of the branches and customers:

branches.dml

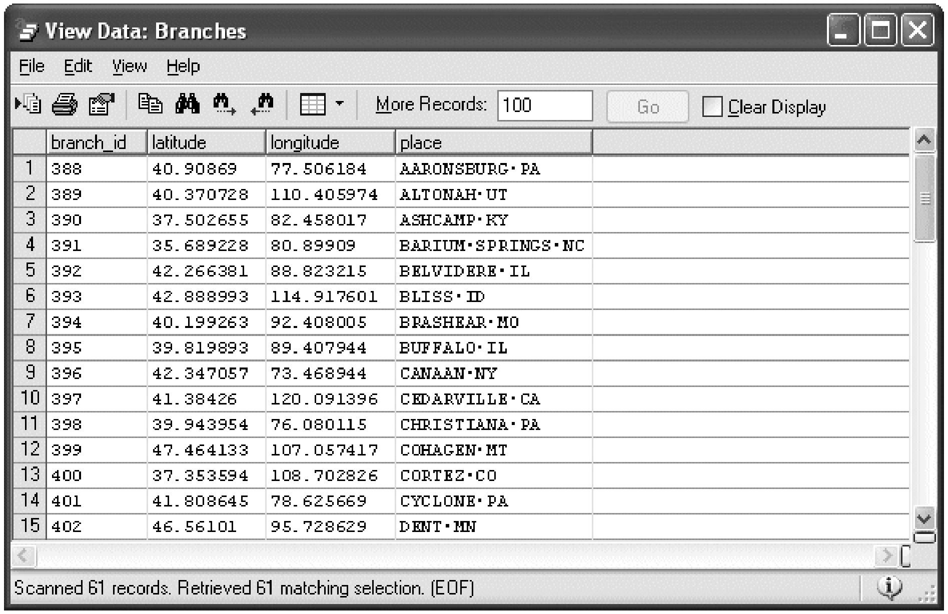
```
record
  decimal(",") branch_id;
  decimal(",") latitude;
  decimal(",") longitude;
  string("\n") place;
end
```

customers.dml

```
record
   decimal(",") customer_id;
   decimal(",") latitude;
   decimal(",") longitude;
   string("\n") place;
end;
```

b. Provide URLs to the data files **branches.dat** and **customers.dat**.

Here is an excerpt from **branches.dat**:



	branch_id	latitude	longitude	place
1	388	40.90869	77.506184	AARONSBURG·PA
2	389	40.370728	110.405974	ALTONAH·UT
3	390	37.502655	82.458017	ASHCAMP·KY
4	391	35.689228	80.89909	BARIUM·SPRINGS·NC
5	392	42.266381	88.823215	BELVIDERE·IL
6	393	42.888993	114.917601	BLISS·ID
7	394	40.199263	92.408005	BRASHEAR·MO
8	395	39.819893	89.407944	BUFFALO·IL
9	396	42.347057	73.468944	CANAAN·NY
10	397	41.38426	120.091396	CEDARVILLE·CA
11	398	39.943954	76.080115	CHRISTIANA·PA
12	399	47.464133	107.057417	COHAGEN·MT
13	400	37.353594	108.702826	CORTEZ·CO
14	401	41.808645	78.625669	CYCLONE·PA
15	402	46.56101	95.728629	DENT·MN

Scanned 61 records. Retrieved 61 matching selection. (EOF)

2. Uses a JOIN to combine the two input datasets as follows:

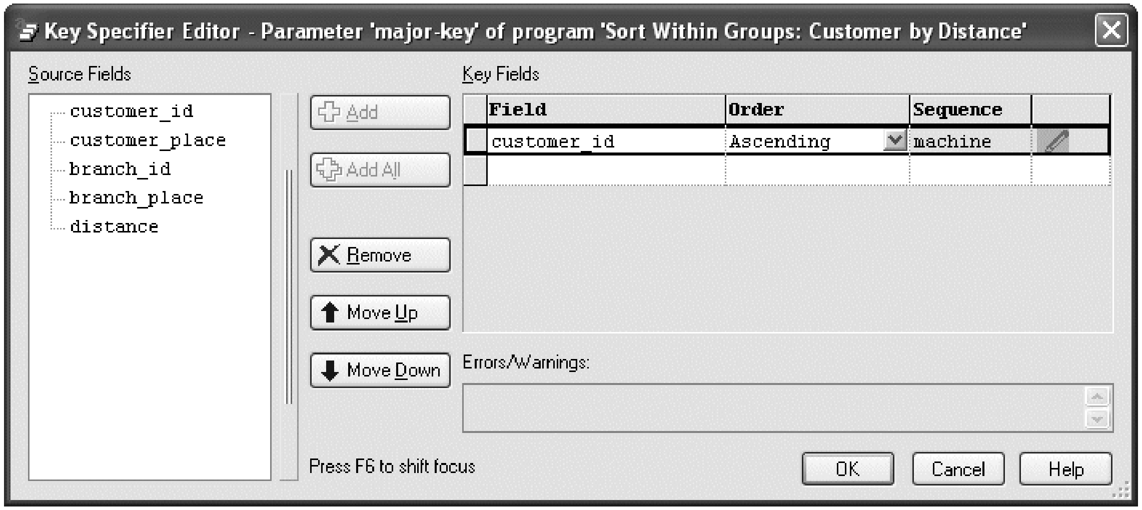
a. Sets the **key** parameter on the JOIN to empty: **{ }** to combine every branch with every customer.

b. Calculates the great circle distance between every branch and every customer. The complete JOIN transform looks like this:

```
include "~$AI_XFR/great-circle-distance.xfr";

out::join(in0, in1) =
begin
  out.customer_id :: in1.customer_id;
  out.customer_place :: in1.place;
  out.branch_id :: in0.branch_id;
  out.branch_place :: in0.place;
  out.distance :: great_circle_distance(in0.latitude,
                  in0.longitude, in1.latitude, in1.longitude);
end;
```

3. Uses a SORT WITHIN GROUPS to produce, for each **customer_id**, a list of records sorted in ascending order by the distance to each branch:

a. Sets the **major-key** parameter to **customer_id** and the order to **Ascending**:

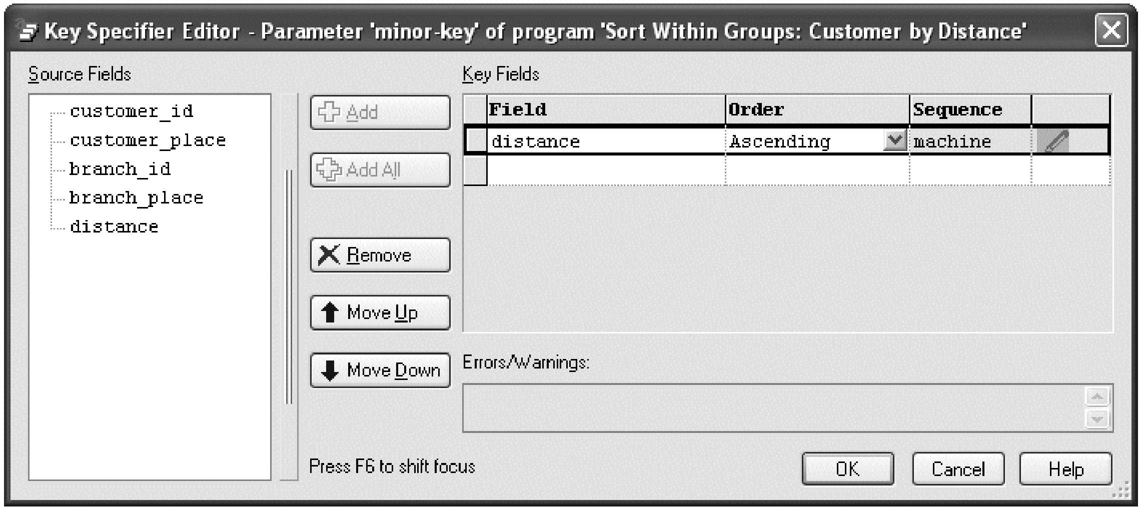


Source Fields:

```
customer_id
customer_place
branch_id
branch_place
distance
```

Field	Order	Sequence
customer_id	Ascending	machine

b. Sets the **minor-key** parameter to **distance** and the order to **Ascending**:


Key Specifier Editor - Parameter 'minor-key' of program 'Sort Within Groups: Customer by Distance'

Source Fields
- customer_id
- customer_place
- branch_id
- branch_place
- distance

Key Fields

Field	Order	Sequence
distance	Ascending	machine

Errors/Warnings:

Press F6 to shift focus

4. Uses a DEDUP SORTED component to get the shortest **distance** for each **customer_id**:

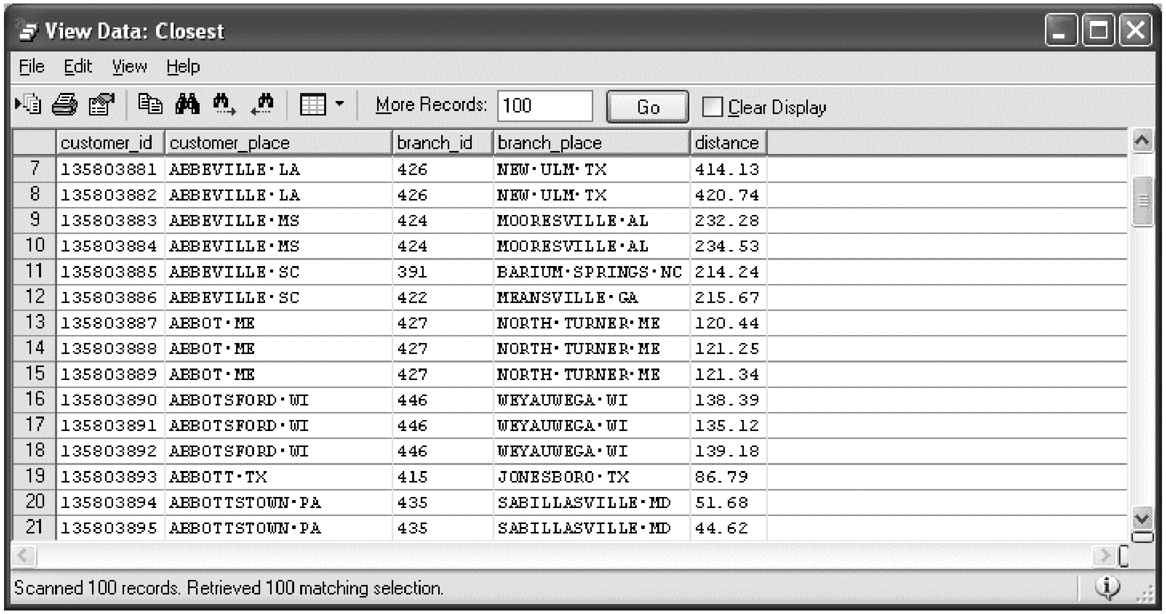
a. Sets the **key** parameter to **customer_id** and the order to **Ascending**.

b. Sets the **keep** parameter to **first** to keep the record with the smallest value (shortest **distance**).

5. Outputs the data, producing records in the format specified by **nearest-branch.dml**:

```
record
  decimal(",") customer_id;
  string(",") customer_place;
  decimal(",") branch_id;
  string(",") branch_place;
  decimal("\n".2) distance;
end
```

Here is an excerpt from the output:



	customer_id	customer_place	branch_id	branch_place	distance
7	135803881	ABBEVILLE·LA	426	NEW·ULM·TX	414.13
8	135803882	ABBEVILLE·LA	426	NEW·ULM·TX	420.74
9	135803883	ABBEVILLE·MS	424	MOORESVILLE·AL	232.28
10	135803884	ABBEVILLE·MS	424	MOORESVILLE·AL	234.53
11	135803885	ABBEVILLE·SC	391	BARIUM·SPRINGS·NC	214.24
12	135803886	ABBEVILLE·SC	422	MEANSVILLE·GA	215.67
13	135803887	ABBOT·ME	427	NORTH·TURNER·ME	120.44
14	135803888	ABBOT·ME	427	NORTH·TURNER·ME	121.25
15	135803889	ABBOT·ME	427	NORTH·TURNER·ME	121.34
16	135803890	ABBOTSFORD·WI	446	WEYAUWEGA·WI	138.39
17	135803891	ABBOTSFORD·WI	446	WEYAUWEGA·WI	135.12
18	135803892	ABBOTSFORD·WI	446	WEYAUWEGA·WI	139.18
19	135803893	ABBOTT·TX	415	JONESBORO·TX	86.79
20	135803894	ABBOTTSTOWN·PA	435	SABILLASVILLE·MD	51.68
21	135803895	ABBOTTSTOWN·PA	435	SABILLASVILLE·MD	44.62

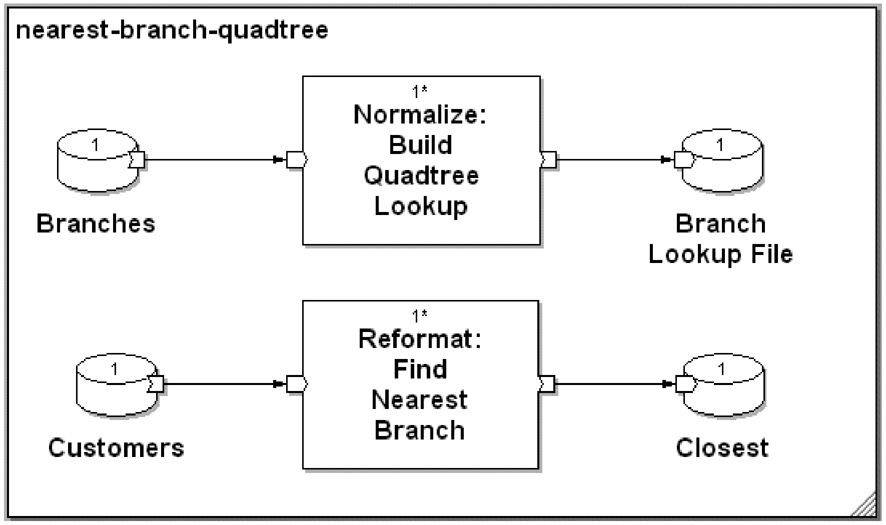
Scanned 100 records. Retrieved 100 matching selection.

Efficient calculation of the nearest branch

The following sections describe a more efficient (and more scalable) solution to finding the nearest branch for every customer. The algorithm is more complex, and the graph, **nearest-branch-quadtree.mp**, runs in two phases:

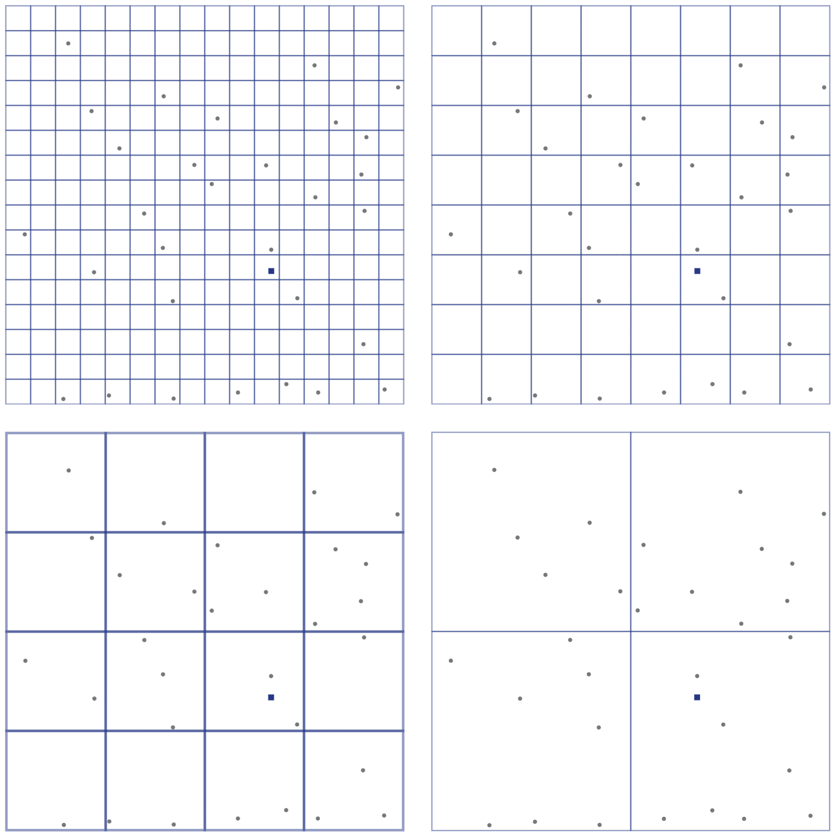
- Building a lookup file of map grids (below). The first phase builds a lookup file that divides a map of branches into a sequence of grids, starting with a fine-grain grid and building up to a very coarse grid. The lookup file is indexed by the size of a specific square in each grid and the **x** and **y** positions of that square.

- Using the lookup file to find the nearest branch (page 203)



Building a lookup file of map grids

The following figure illustrates the results of calculating the first four grids for a number of branch locations. The code used to create the grids is described in "What the transform does" (page 199).



What the transform does

The graph's NORMALIZE component builds the lookup file as follows:

1. Includes **utilities.xfr**, a transform that does the following:

  a. Includes the transform for calculating the great circle distance (page 188).

b. Defines constants for:

- The location of the center of the map (a point in the center of the continental United States)
- The locations of the latitudes of zero distortion
- Other needed values computed from the above
- An integer representing a square size (in kilometers) that is larger than the surface of the earth

c. Converts a **point_deg_type** to a **point_km_type**.

d. Uses the **point_km_type** value and a square size to get the southwest corner of the square containing the **point_km_type**.

The complete **utilities.xfr** transform looks like this:

```
include "/~$AI_XFR/great-circle-distance.xfr";

constant real(8) phi_0 = 40 * to_radians;
constant real(8) lambda_0 = 90 * to_radians;

constant real(8) phi_1 = 20 * to_radians;
constant real(8) phi_2 = 60 * to_radians;

constant real(8) n = (math_cos(phi_1) - math_cos(phi_2))/
                        (phi_2 - phi_1);
constant real(8) G = math_cos(phi_1) / n + phi_1;
constant real(8) rho_0 = radius * (G - phi_0);
constant int largest_square_size = 65536;
```

```
out::map_coordinates(in) =
begin
  let real(8) theta = n * (in.longitude * to_radians -
lambda_0);
  let real(8) rho = radius * (G - in.latitude * to_radians);
  out.x :: rho * math_sin(theta);
  out.y :: rho_0 - rho * math_cos(theta);
end;

out::containing_square(point, size) = begin
  out.x :: size * decimal_round_down(point.x / size, 0);
  out.y :: size * decimal_round_down(point.y / size, 0);
end;
```

2. Includes **point-types.dml**, which defines two data types as follows:

- **point_km_type** — Represents a point on the map, with coordinates in kilometers

  ```
  type point_km_type =
  record
    real(8) x;
    real(8) y;
  end;
  ```

- **point_deg_type** — Represents a point on earth, with coordinates in degrees

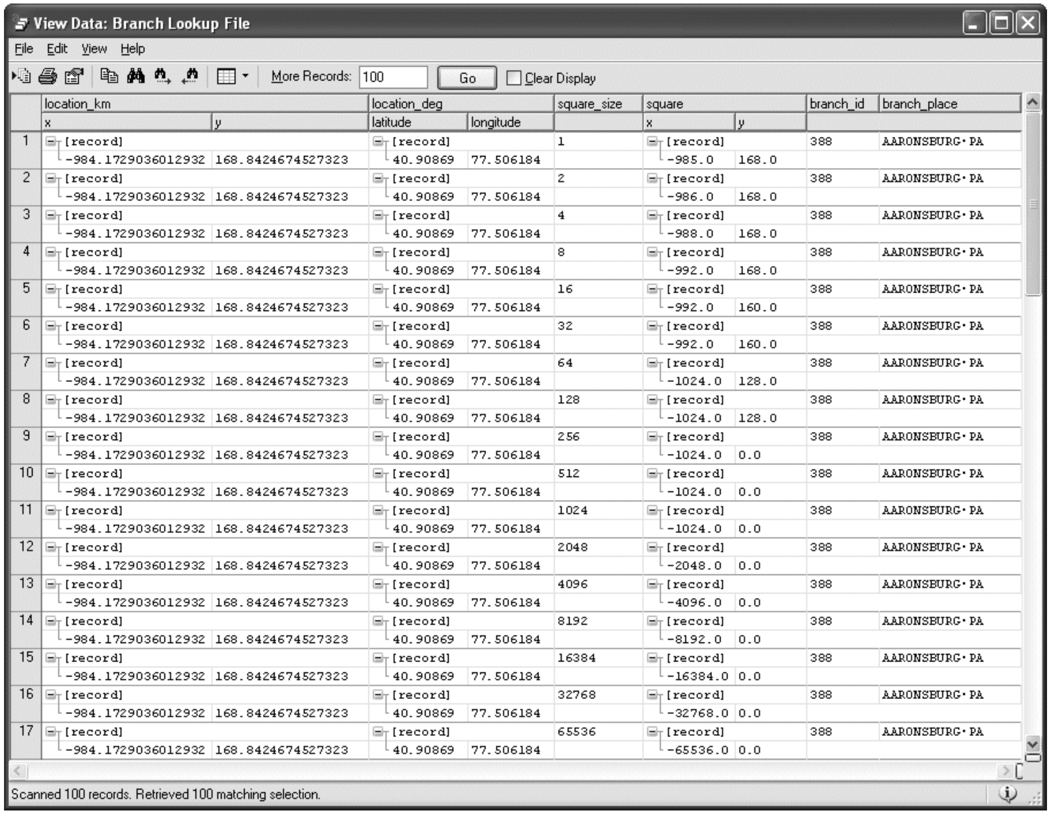
  ```
  type point_deg_type =
  record
    real(8) latitude;
    real(8) longitude;
  end;
  ```

3. Calculates 17 grids with increasing square sizes for each input branch location, converted to a **point_km_type** by **map_coordinates**. The length of a square's side (in kilometers) is a power of 2 from 1 to 65,536 (2^0 to 2^{16}). A square of length 65,536 km is larger than the earth's surface and so is guaranteed to contain every location.

The complete NORMALIZE transform looks like this:

```
include "~$AI_XFR/utilities.xfr";
include "~$AI_DML/point-types.dml";

out::length(in) =
begin
  out :: 17;
end;

out::normalize(in, index) =
begin
  let point_km_type location_km = map_coordinates(in);
  let integer(4) square_size = 1 << index;
        /*The size of a square, in kilometers, is 2 to the power of
          its index.*/

  out.location_deg.* :: in.*;
  out.location_km :: location_km;
  out.square_size :: square_size;
  out.branch_id :: in.branch_id;
  out.square :: containing_square(location_km, square_size);
  out.branch_place :: in.place;
end;
```

Following is an excerpt from the output (**Branch Lookup File**):



View Data: Branch Lookup File

File Edit View Help

More Records: 100 Go ☐ Clear Display

	location_km		location_deg		square_size	square		branch_id	branch_place
	x	y	latitude	longitude		x	y		
1	[record]		[record]		1	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-985.0	168.0		
2	[record]		[record]		2	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-986.0	168.0		
3	[record]		[record]		4	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-988.0	168.0		
4	[record]		[record]		8	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-992.0	168.0		
5	[record]		[record]		16	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-992.0	160.0		
6	[record]		[record]		32	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-992.0	160.0		
7	[record]		[record]		64	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-1024.0	128.0		
8	[record]		[record]		128	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-1024.0	128.0		
9	[record]		[record]		256	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-1024.0	0.0		
10	[record]		[record]		512	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-1024.0	0.0		
11	[record]		[record]		1024	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-1024.0	0.0		
12	[record]		[record]		2048	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-2048.0	0.0		
13	[record]		[record]		4096	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-4096.0	0.0		
14	[record]		[record]		8192	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-8192.0	0.0		
15	[record]		[record]		16384	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-16384.0	0.0		
16	[record]		[record]		32768	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-32768.0	0.0		
17	[record]		[record]		65536	[record]		388	AARONSBURG·PA
	-984.1729036012932	168.8424674527323	40.90869	77.506184		-65536.0	0.0		

Scanned 100 records. Retrieved 100 matching selection.

Using the lookup file to find the nearest branch

To find the nearest branch, the graph **nearest-branch-quadtree.mp** uses a REFORMAT in combination with the **Branch Lookup File** (described in the previous section) to search successively larger grid squares until the closest branch is found.

The REFORMAT transform does the following:

1. Includes **utilities.xfr** (described in “What the transform does” on page 199), which in turn includes **great-circle-distance.xfr** and has a function for converting map coordinates from **point_deg_type** to **point_km_type**:

   ```
   include "~$AI_XFR/utilities.xfr";
   ```

2. Includes a DML file that defines the record type used in the lookup file:

   ```
   include "~$AI_DML/branch-lookup-type.dml";
   ```

   The record type is defined as follows (see **point-types** on page 201):

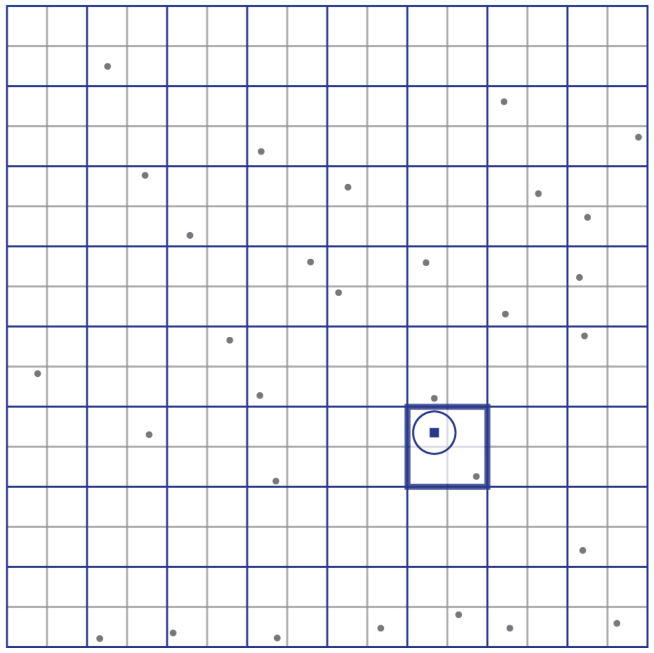
   ```
   include "/~$AI_DML/point-types.dml";

   type branch_lookup_type = record
     point_km_type location_km;
     point_deg_type location_deg;
     integer(4) square_size;
     point_km_type square;
     decimal(",") branch_id;
     string("\n") branch_place;
   end;
   ```

3. Defines a variable for an imaginary branch that is much farther away than any real branch, for use in initializing the loop that finds the closest branch:

   ```
   let branch_lookup_type far_away_branch =
     [record
       location_km  [record x 1000000 y 1000000]
       location_deg [record latitude 1000 longitude 1000]
       square_size  1
       square       [record x 10000000 y 1000000]
       branch_id    -1
       branch_place "Far away"];
   ```

4. Defines a function, **four_containing_squares(***point*, *size***)**, that takes a **point_km_type**, and a **size**, which is the length of a square in the grid built by the NORMALIZE (see "What the transform does" on page 199). The function returns the southwest corner of the square containing that point, and the southwest corners of the three other squares closest to the point. These are the four squares that overlap a circle of radius **size** centered on **point**. (See the figure after the code below.)

```
out::four_containing_squares(point, size) =
begin
  let real(8) x = size * decimal_round_down(point.x / size, 0);
  let real(8) y = size * decimal_round_down(point.y / size, 0);
  let real(8) s = (real(8))size;
  let real(8) x_off = if ((point.x - x) < (s / 2)) -s else s;
  let real(8) y_off = if ((point.y - y) < (s / 2)) -s else s;

  out :: [vector [record x x y y],
                 [record x x + x_off y y],
                 [record x x y y + y_off],
                 [record x x + x_off y y + y_off]];
end;
```



5. Defines several variables. Following are descriptions of the most important ones:

VARIABLE	PURPOSE	INITIAL VALUE	DETAILS / CHANGE PER ITERATION
position_deg	Location for which the nearest branch is sought, in degrees	Location provided in the input record	The transform does not change the value of this variable.
position_km	Location for which the nearest branch is sought, in kilometers	Location provided in the input record, converted to kilometers	The transform does not change the value of this variable.
size	Length (in km) of a side of a square in the current grid	**1**	Initially, this is the size of the smallest square to search in. The optimal value depends on the typical distance between branches. At the end of each iteration, **size** is doubled, and the next iteration searches the next-larger grid size.
closest_branch	Location of the branch that is currently closest, as a **branch_lookup_type**	**far_away_branch**	Initially, this is the location of the imaginary branch defined earlier (see Step **3** above), which is farther away than any actual branch. Whenever the transform finds a branch that is closer, it assigns the newly found branch location to this variable. When the loop has iterated to search the next-larger grid size (see **size** above) and has not found a branch closer than the current one (see **distance_to_closest_branch** below), the loop terminates, and the value of **closest_branch** is output.

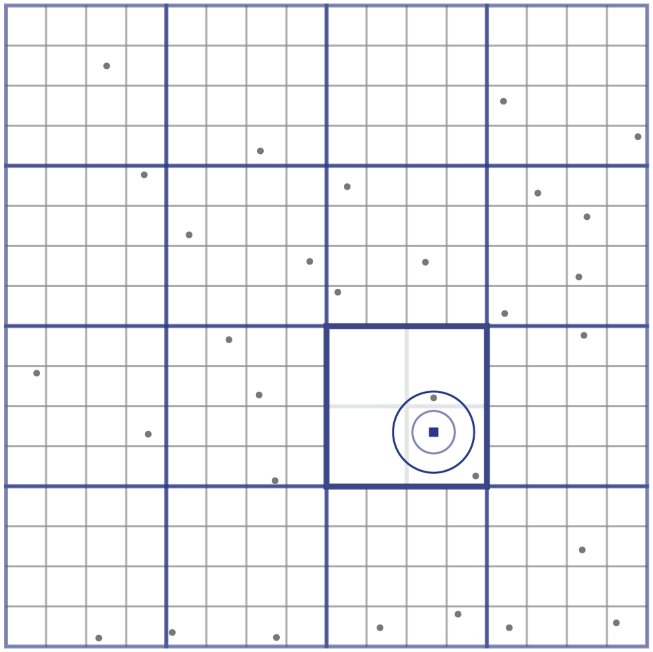
VARIABLE	PURPOSE	INITIAL VALUE	DETAILS / CHANGE PER ITERATION
distance_to_ closest_branch	Distance to **closest_branch**	**1000000**	Initially, this is set to a distance that is farther away than any actual branch. Whenever the transform finds a branch that is closer, it assigns the distance to the newly found branch to this variable. When the loop has iterated to search the next-larger grid size (see **size** above) and has not found a distance shorter than the current value of this variable, the loop terminates, and the value of **closest_branch** is output.

Here are the definitions in the transform:

```
out::reformat(in) =
begin
  let point_deg_type position_deg = in;
  let point_km_type position_km = map_coordinates(position_deg);
  let integer(4) size = 1;
  let branch_lookup_type closest_branch = far_away_branch;
  let real(8) distance_to_closest_branch = 1000000;
  let point_km_type[4] four_squares;
  let integer(4) j;
  let point_km_type square;
  let integer(4) count;
  let integer(4) k;
  let branch_lookup_type branch;
  let real(8) distance;
```

6. Loops to search an incrementally larger square in the **Branch Lookup File** as long as the current square size is smaller than the largest possible square size and the distance to the closest branch is greater than the current square size divided by 4. If the distance to the closest branch is not greater than the current square size divided by 4, the larger grid cannot possibly contain a closer branch; the closest one has been found, and the loop can be terminated. (See the figure below.)

```
while (size <= largest_square_size and
       distance_to_closest_branch > size / 4)
```



a. Inside the loop, the transform calculates a vector containing the coordinates of the southwest corners of the square containing the point, and of the three squares closest to it, each of length **size**:

```
begin
  four_squares = four_containing_squares(position_km, size);
```

b. Still inside the loop, the transform looks up the **square_size** and the **x**, **y** coordinates of the four containing squares in the **Branch Lookup File** to see whether it contains a corresponding entry (or entries) for one or more branch locations. If it does, the transform calculates the great circle distance between the input position and the branch location(s). If more than one branch location is found for a given square size, it keeps the closest one:

```
    for (j, j < 4)
      begin
        square = four_squares[j];
        count = lookup_count("Branch Lookup File", size,
                             square.x, square.y);
        for (k, k < count)
          begin
            branch = lookup_next("Branch Lookup File");
            distance = great_circle_distance(
                                  position_deg.latitude,
                                  position_deg.longitude,
                                  branch.location_deg.latitude,
                                  branch.location_deg.longitude);
            if (distance < distance_to_closest_branch)
              begin
                // Found a new closest branch.
                distance_to_closest_branch = distance;
                closest_branch = branch;
              end
          end
      end
```

c. As the last operation in the loop, the transform increases the square to the next-larger size:

```
      size = size * 2;
    end
```

7. Outputs the customer and branch data:

```
    out.customer_id :: in.customer_id;
    out.branch_id :: closest_branch.branch_id;
    out.branch_place :: closest_branch.branch_place;
    out.customer_place :: in.place;
    out.distance :: distance_to_closest_branch;
  end;
```

Cook>Book graphs listed by component and feature

Graphs listed by component

Following is an alphabetical list of the most important components used in the *Cook>Book* sandboxes, and the graphs that use them.

COMPONENT	GRAPH	PAGE
BROADCAST	global_sort.mp	57
	quantile3.mp	95
CONCATENATE	multiplying-data-with-departitioners.mp	18
	quantile3.mp	95
DEDUP SORTED	nearest-branch-simple.mp	193
DENORMALIZE SORTED	quantile3.mp	95

COMPONENT	GRAPH	PAGE
FILTER BY EXPRESSION	ht_validate1.mp	169
	ht_validate2.mp	174
	multiplying-data-by-3-way-self-join-with-FBE.mp	19
FIND SPLITTERS	global_sort.mp	57
FUSE	benchmarking-with-checking.mp	49
GATHER	multiplying-data-with-departitioners.mp	18
INTERLEAVE	multiplying-data-with-departitioners.mp	18
	quantile3.mp (Get Counts subgraph)	97
JOIN	difference.mp	65
	nearest-branch-simple.mp	193
	quantile1.mp	90
	quantile2.mp	92
	quantile3.mp	95
JOIN (Self-Join)	multiplying-data-by-3-way-self-join-with-FBE.mp	19
LOOKUP FILE	nearest-branch-quadtree.mp	197
MERGE	global_sort_serial_simple.mp	62
	ht_build.mp	176
NORMALIZE	fill-gaps.mp	34
	multiplying-data-by-fixed-number.mp	14

COMPONENT	GRAPH	PAGE
PARTITION BY KEY AND SORT	difference.mp	65
PARTITION BY RANGE	global_sort.mp	57
PARTITION BY ROUND-ROBIN	quantile3.mp (Get Counts subgraph)	97
READ MULTIPLE FILES	read_and_write_multiple_files.mp	144
	read_multiple_files_simple.mp	140
REFORMAT	calculating-distances.mp	186
	difference.mp	65
	ht_process1.mp	160
	ht_validate2.mp	174
	make_difference_metadata.mp	84
	name-parsing.mp	6
	nearest-branch-quadtree.mp	197
REPLICATE	benchmarking.mp	45
	benchmarking-with-checking.mp	49
	difference.mp	65
	ht_validate1.mp	169
	ht_validate2.mp	174
	quantile3.mp	95
	quantile3.mp (Get Counts subgraph)	97

COMPONENT	GRAPH	PAGE
ROLLUP	audit1.mp	111
	audit2.mp	115
	audit3.mp	116
	audit4.mp	119
	ht_build.mp	176
	ht_process1.mp	160
	ht_validate1.mp	169
	ht_validate2.mp	174
	quantile1.mp	90
	quantile2.mp	92
	quantile3.mp (Get Counts subgraph)	97
SAMPLE	global_sort.mp	57
SCAN	fill-gaps.mp	34
	ht_process2.mp	165
	making-unique-keys.mp	21
	quantile2.mp	92
	quantile3.mp (Get Counts subgraph)	97

COMPONENT	GRAPH	PAGE
SORT	global_sort.mp	57
	global_sort_serial_out.mp	62
	global_sort_serial_simple.mp	62
	quantile1.mp	90
	quantile2.mp	92
SORT WITHIN GROUPS	making-unique-keys.mp	21
	nearest-branch-simple.mp	193
TRASH	benchmarking.mp	45
	difference.mp	65
WRITE MULTIPLE FILES	read_and_write_multiple_files.mp	144

Graphs listed by feature

Following is an alphabetical list of important Ab Initio software programming features used by the graphs in the *Cook>Book* sandboxes, and the graphs that use them.

FEATURE	GRAPH	PAGE
Ad hoc multifiles	ad_hoc_command.mp	137
	ad_hoc_explicit.mp	134
	ad_hoc_parameter.mp	137
	ad_hoc_wildcard.mp	135
DML conditional fields	ht_data.mp	153
DML include	audit4.mp	119
	calculating-distances.mp	186
	ht_data.mp	155
	nearest-branch-quadtree.mp	199
	nearest-branch-simple.mp	193
	read_and_write_multiple_files.mp	144
	read_multiple_files_simple.mp	140

FEATURE	GRAPH	PAGE
DML looping	benchmarking-low-level.mp	54
	ht_validate2.mp	174
	make_difference_metadata.mp	85
	nearest-branch-quadtree.mp	203
DML **reinterpret_as** function	audit4.mp	126
DML subrecords	ht_data.mp	155
DML trig functions	calculating-distances.mp	188
	nearest-branch-quadtree.mp	199
DML vectors	ht_process3.mp	167
	make_difference_metadata.mp	85
final_log_output function in components	audit4.mp	121
Graph parameters	ad_hoc_parameter.mp	137
	benchmarking.mp	45
	generic_difference.mp	80
	make_difference_metadata.mp	84
	multiplying-data-with-parameter.mp	15
Local variables	name-parsing.mp	6
Metaprogramming	make_difference_metadata.mp	84

FEATURE	GRAPH	PAGE
Parameterized subgraphs	multiplying-data-with-parameterized-subgraph.mp	16
Simple string manipulation	name-parsing.mp	6

Index