**built in graph exercises**

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/casting\_join\_keys.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/casting\_out\_of\_town\_purchases.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/design\_replace\_join.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/design\_splitting\_data\_flows.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/design\_unnecessary\_join.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/departitioning\_gather.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/departitioning\_total\_sales.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/design\_conditional\_logic.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/design\_unnecessary\_reformat.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/design\_unnecessary\_sorts.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/design\_use\_sort\_within\_groups.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/dml1\_edit\_record\_format.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/dml1\_null\_value.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/dml2\_conditional\_dml.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/dml2\_varstrings.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/errors\_clean\_data.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/inmemory\_enriched\_transactions.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/inmemory\_sales\_relationship.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/join\_inner\_join.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/join\_outer\_join.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/layouts\_check\_layout.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/layouts\_override\_db.mp

/home/ggx/<your\_userid>/examples/graph-dev-basics/mp/layouts\_set\_layout.mp

**Advanced Class:**

**DAY 1:**

4:30 PM -- 8:30 PM IST   
        we'll take a break about halfway through, at about 6:30 PM IST   
        15-20 minutes   
  
"Advanced Graph Design"   
compound types -- records, vectors   
        syntax for declaring types   
        syntax for using functions to work with these types   
performance   
        pipeline, component & data parallelism   
        record format performance   
        how to measure performance   
reuse & productivity   
  
design --> I introduce a concept, discuss it, you can ask questions   
        we do a hands-on exercise   
        repeat until Friday   
  
After class today, I'll send instructions on how to get access to the PPT slides   
AND   
any notes that I write in Notepad like this, I will email at the end of class each day.   
  
++++++++++++++++++++++++++++++++++++++   
  
If you have a ROLLUP component, running with a serial layout, and you set the key to be "empty"   
        { }   
how many output records do you get?   
1   
Why? An empty key { } --> all records are considered to have the same key value.   
In that case, ROLLUP groups all the input records into 1 group (= same key value) and then summarizes them in a single output record.   
  
If you have a JOIN and you set the key to be "empty"   
        { }   
and let's say that you have 20 records on in0, and 30 records on in1, how many output records do you get?   
600 (20 \* 30) = "cartesian join"   
Why? An empty key { } --> all records are considered to have the same key value.   
In that case, every record joins to every other records (many-to-many).   
  
We have a slightly different situation, where we have 20 records on in0 and 1 record on in1.   
That 1 record is joined with every one of the 20 input records.   
This is a way to compute a value over the whole dataset & then join it back to every input record.   
  
=================================   
  
Our to-do list:   
1- From the input transactions, we need to find the store\_no for each customer that is from the most recent (max transaction\_date) transaction. (Customers can shop at different stores -- we need specifically the store\_no from the latest transaction)   
2- Add Stores dataset to the graph. (When to use LOOKUP FILE versus JOIN.)   
3- Get the manager's name + address from the Stores dataset based on the store\_no that we computed in step #1.   
  
====================================   
  
When you write rules that compute individual output fields in ROLLUP   
        out.max\_trx\_date :: max(in.trans\_date);   
        out.store\_no :: in.store\_no;   
Each output field -- in this case -- is computed individually. The value for the output field can come from \*different\* records in the input group. (Think about the fact that you \*can\* compute min(in.amt) and max(in.amt) in the same rollup -- those values have to come from diff records.)   
  
Output of the above:   
        max trx date --> from whichever record in the group has the latest trx date   
        store\_no --> from either the first or the last record in the input group   
  
customer\_id        store\_no                trx\_date   
1                A                Feb 15   
1                B                Mar 27   
1                C                Jan 20   
....   
  
max(in.trans\_date) --> Mar 27  (from 2nd record)   
in.store\_no --> either A or C   
        (both are incorrect in this case -- we want B because that's the store from 2nd record   
                        which contains max trx date)   
  
(Undocumented! So don't rely on this....   
        Rollup in-memory when used with a rule that doesn't use an aggregation function   
                out.store\_no :: in.store\_no;   
        returns the value from the \*first\* record in the input group.   
          
        Rollup with sorted input when used with a rule that doesn't use an aggregation function   
                out.store\_no :: in.store\_no;   
        returns the value from the \*last\* record in the input group   
)   
  
If you truly wanted the value from the first record in the group or the last record in the group,   
we have aggregation functions   
        first()   
        last()   
we recommend that you use those because they document the graph better -- they tell people exactly which value you wanted.   
  
  
===============   
  
  
customer\_id        store\_no                trx\_date   
1                A                Feb 15   
1                B                Mar 27   
1                C                Jan 20   
....   
  
  
What should our SORT key be so that we can select the most recent store for each customer?   
        {customer\_id ascending; transaction\_date descending}   
Then we do ROLLUP   
        key         {customer\_id} because we still just want 1 record per customer   
        xfr        out.store\_no :: first(in.store\_no);   
                we've sorted the input data such that the latest/most recent trx is the first record   
                for each customer   
                we'll output the store\_no from the first() record in each customer group   
        out dml        add store\_no to the output record format   
  
  
After the exercise, I'll show you why   
        sorting on descending + first()   
is better than   
        sorting on ascending + last()   
  
  
Sorting data descending versus ascending has the same performance. It doesn't matter what order you sort the records in -- you get the same performance.   
  
The first() function in Rollup does less work (per record) than the last() function.   
And thus, if you have a choice about which one to use -- first() is better than last().   
  
Now -- if your data was already sorted...   
        customer\_id & transaction\_date ascending (and you needed the data to be sorted by   
        trx\_date asc for other components/requirements in the graph), then you should use last().   
  
But if you have a choice about how the data is sorted, first() is the faster way to do it.   
  
  
You can look at how any of our aggregation functions are designed by looking at the Transform Expansion.   
Open the transform in Grid View in the Transform Editor.   
        View > Transform Expansion --> read-only detail of what the functions are doing.   
  
  
first()   
        for the first record in the group, it calls initialize() to set the value of the global variable to   
                be the value from the first record   
        for each record in the group, it calls rollup() which checks if the global is NULL   
                almost always it will \*not\* be NULL (condition is false) --> nothing happens   
                if the value in the first record had been NULL, replace that value with the value   
                        from the current input record (happens very rarely)   
        once we get the last record in the group, it calls finalize() to create an output record   
                outputs the value from the global variables.   
  
  
last()   
        for the first record in the group, it calls initialize() to set the value of the global variable to   
                be the value from the first record   
        for each record in the group, it calls rollup() which checks if the input value is NULL   
                almost always it will \*not\* be NULL (condition is true!) --->   
                        global variable is updated with the value from the current input record   
                rarely the input value will be NULL --> nothing happens   
        once we get the last record in the group, it calls finalize() to create an output record   
                outputs the value from the global variables.   
  
  
Big difference between first() and last()   
        for each input record -- most of the time, first() does nothing   
                        -- most of the time, last() updates the value of the global variable   
last() does more work than first()   
  
========================================================   
  
In the output of "air sandbox diff" and certain error/validation messages, you might see references to out\_metadata, in\_metadata, etc.   
Everythign that you can configure about a component (all its properties) are parameters.   
        out\_metadata --> name of the parameter that holds the value of the output record format   
        in\_metadata --> name of the parameter that holds the value of the input record format if   
                        there's only 1 input   
        in1\_metadata, in0\_metadata --> working with join for example   
  
  
What did we edit for exercise #1?   
        SORT                key        added transaction\_date descending           
        ROLLUP {cust\_id}        xfr        added rule for store\_no   
                        out dml        added store\_no   
        OUTPUT FILE        write dml        added store\_no   
        JOIN { }                xfr        added rule for store\_no (I used wildcards)   
                        out dml        added store\_no   
  
ROLLUP on { } wasn't changed because it has nothing to do with store\_nos.   
  
But the JOIN on {cust\_id} outputs the store\_no   
        why didn't we have to change its output DML? propagated from the output file (which we   
                        did modify)   
        why didn't we have to edit its transform to add a rule to pass through store\_no? already   
                used wildcard rules -- by adding store\_no to the input & output record formats,   
                the wildcard rules automatically updated to do a pass through of store\_no.   
  
We saved time on editing the graph by using propagation & wildcard rules!   
--> productivity tip : use propagation + wildcard rules whenever they make sense.   
        they make the graph simpler (because there are fewer record formats + they are fewer   
                rules in the transform -- easier to read/understand)   
        they make it quicker to edit the graph if you need to.   
  
================================   
  
LOOKUP FILE versus JOIN -- which one to use?   
  
Two considerations:   
        -- performance   
                speed -- looking up records in a lookup file is slightly faster than in-memory Join   
                memory usage -- amount of memory required for loading a particular file into   
                        memory for lookup versus in-memory join is basically the same   
                        multiple components can share the same lookup file if they are running   
                                on the same server   
                        whereas if you did multiple in-memory joins on the same data file, it   
                                would be loaded into memory separately for each join   
What if the data you need to join (either as the lookup file or as the nondriving input to the in-memory Join) is very large?   
        in-memory Join has a max-core parameter -- limits how much memory the join is allowed to use   
                --> failure from lack of memory is extremely extremely rare   
                if the amount of memory required is larger than the max-core, Join writes temp   
                        files on disk to store the data until it can process it ("spillage")   
        lookup file does not have that limit -- it will try to use as much memory as it needs to load the   
                entire lookup file into memory   
                if the amount of memory you need > how much available RAM your system has -->   
                        operating system starts "swapping" (moves data from memory to   
                        temporary files on disk); if the swapping becomes very frequent, it can   
                        make \*all\* the applications on the server significantly slower, failures can   
                        occur   
        spillage is better than swapping -- spillage affects only your graph, all the other applications   
                are unaffected + we know our own components, we can arrange the data that is   
                spilled to disk in the most efficient way possible (the operating system doesn't have   
                that knowledge)   
        in a case with an extremely large unsorted file, your best choices is actually to sort the data   
                the sort will spill data to disk too -- when you have a lot of spillage, it spills faster   
                        than in-memory Join.   
  
What is "very large"?   
        relative to how much available (memory) RAM you have   
        more RAM you have -- the bigger your lookup files can be   
        after you get beyond a few GBs, conventional lookup files often drop off in performance   
                --> JOIN   
                --> other types of lookup files (ICFF -- index compressed flat file -- require   
                        time investment in designing maintenance & architecture around these)   
        often the choice to use lookup file or not -- when the data is more than a few megabytes is   
                typically an architectural decision   
        if the lookup file is smaller than ~67 MB -- that's the default max-core for in-memory JOIN,   
                either way, you're using about that much memory (under this size is often okay   
                for the developer to choose which works best)   
  
==================> tl;dr   
        if you have relatively small, unsorted data --> LOOKUP FILE offers best performance   
        if you have larger, unsorted data --> depends on how much available memory you have   
                LOOKUP FILE versus in-memory JOIN versus SORT + JOIN with sorted input   
        if you have very large unsorted data --> SORT or use ICFFs   
  
==========================================================   
graph design   
        what type of join behavior do you want?   
                because lookup file is not connected by a flow to the rest of the graph, there is   
                no way to extract records from the lookup that weren't matched   
                --> lookup() cannot do full outer or right outer joins to access unmatched records   
                                in the lookup   
                if you need to do those types of joins --> use JOIN   
        where is the data that you want to use as a lookup?   
        is the data that you want to use as a lookup in the format you need / or does it have to be   
                transformed?   
                if you need to read the data from a db/spreadsheet/XML doc and/or transform it   
                        (add/drop fields, do computations) before you use the data as lookup   
                        then you have to add phasing to your graph or you need two graphs   
                --> explore both options -- is lookup file faster+more convenient that those benefits   
                        outweight the cost of phasing/maintenance of two graphs   
                or is join a better choice   
        are there other components in this graph that use the same key or similar keys?   
                if there are no other components in the graph that use this key, often lookup file   
                is better --> you can usually call lookup() from an existing component in the graph,   
                you don't have to repartition the data, you can use a component like Reformat that   
                preserves record order   
  
                if there are other components in the graph that use this key -- especially if the data   
                is sorted on that key, then often join is better choice because the data is already   
                sorted / you have to repartition the data for that key anyway   
  
==================> tl;dr   
        lookup file works best for data that is already in file, formatted/transformed into the format you   
                need, inner join or left join (where unmatched records from lookup aren't required)   
        use join if you do need to do a full outer / right outer join or if you have already partitioned +   
                sorted the data on that key   
  
  
=========================================================   
  
The Stores data is   
        10 kB and is unlikely to grow larger in the future   
        already stored in a file & formatted the way we need (e.g., we don't have to do aggregations,   
                etc.)   
        there are no other components in this graph that use store\_no as the key   
        we don't care about stores that customers haven't shopped at (inner join)   
  
What should we use?   
        LOOKUP FILE   
          
  
=============================================================   
  
Does it matter -- in terms of performance -- if the data in lookup file is sorted or not?   
Very slightly, if the lookup file is pretty large.   
  
If this is a lookup file that you'll be using often by many other graphs & the data is fairly large, then it might be an overall net benefit to sort it.   
  
If this is a small lookup file or it's only used in one graph --> it's probably not worth the cost of sorting it for the small benefit that you get from slightly faster lookup()s.   
  
  
When you run a graph that uses a lookup() -- at the start of the phase that the component(s) that call the lookup() run   
        --> load the data in the lookup file into memory   
        --> while we do that, we create an "index" (hash table -- also in memory -- that enables us   
                to quickly determine if a particular key exists in the lookup file or not & if yes,   
                enables us to quickly locate that record)   
-- at the end of the phase, both the lookup file + its index are removed from memory (and the memory is released)   
  
--> if you add LOOKUP FILE to your graph but you never call lookup() on it -- it never gets loaded into memory! LOOKUP FILE only uses memory if you actually access it with lookup()   
--> if you have two components that use LOOKUP FILE & the components are in two different phases, index is created + LOOKUP FILE is loaded & unloaded from memory twice   
        ==> can you move the two components into the same phase so that they can share the lookup   
                file data + index?   
        ==> if not -- you can pre-create the index & save it in a file   
                saves you some time at phase startup-- you don't have to create the index again!   
  
If you have a lookup file that is used by components in different phases in the same graph and/or by components in different graphs AND you know that the lookup file will \*not\* change between phases/graphs, you can pre-create the index using the WRITE LOOKUP component.   
And then you use the LOOKUP TEMPLATE component and the lookup\_load() function to load the lookup into memory before you call it -- enables you to specify that you have an existing index to use (instead of recomputing it)   
  
Other neat usage of WRITE LOOKUP is that it will tell you how big the index will be (how memory it will need!)   
        Stores.dat --> WRITE LOOKUP --lookup--> TRASH (size of the data)   
                                --index--> TRASH (size of the index)   
Stores.dat is 10 KB + 6 KB index.   
  
  
  
If you're familiar with dynamic lookups -- lookups that can be appended to while the graph is running & the graph can then access the new data   
        --> use LOOKUP TEMPLATE component to specify the lookup file data   
        --> use lookup\_load() to load the lookup data into memory   
        --> data is unloaded from memory & the memory is released when any of the following occur   
                you call lookup\_unload()   
                or the component finishes   
                or the phase finishes (all components in the phase shut down when the phase is   
                        done)   
  
  
=======================================================   
  
Nested record or subrecord   
        record inside another record   
Let's assume that this is an input record (named "in")   
        record   
          string(1) id;   
          record   
             string(int) full\_name;   
             date("YYYYMMDD") birthyear;   
          end student\_info;   
        end   
  
student\_info is a nested record / subrecord   
  
in.id --> string(1)   
in.student\_info --> record   
             string(int) full\_name;   
             date("YYYYMMDD") birthyear;   
          end   
in.student\_info.full\_name --> string(int)   
  
dot syntax enables us to access the value of field inside a record   
        in.field1   
        lookup("Stores Lookup", in0.store\_no).store\_manager   
        out.max\_trx\_date   
  
shortcuts   
in.student\_info.full\_name --> string(int)   
in..full\_name  --> string(int)   
        (works only if there are no other subrecords with a field called full\_name)   
  
You can have as many levels of nesting as you want.   
The more levels of nesting you have --> the slower it is to find things that are very deeply buried.   
Flatter formats that have fewer levels of nesting are faster.   
        --> mainframe data   
        --> XML data (SWIFT, health care formats)   
        --> metaprogramming (using transform code in a graph to dynamically write the record formats,   
                transforms, & keys for that graph at runtime -- enables the same graph to be   
                used with many different inputs)   
        --> some of our functions return "record" as a type -- specifically the max() & min() aggregation   
                function in ROLLUP when used with a comparison key (more on that later!)   
  
  
=======================================   
  
wildcard rules   
        out.\* :: in.\*;   
  
record assignments   
when you do a record assignment, the syntax is that the source (in or the function, like lookup) must provide values for \*all\* the fields in the target record (out)   
        out :: in;   
  
  
If a field in the output of a transform is not assigned a value from any rule in the transform   
        --> we check output record format to see if that field allows NULL values   
                if yes, assign NULL to that field   
                if no,   
                        --> we check output record format to see if the field has a default value   
                                if yes, assign default value to that field   
                                if no,   
                                        --> that input record is rejected   
  
If an input record is rejected, what happens next depends on the "reject-threshold" setting of the component   
        never abort --> input record goes to the "reject" port & the component continues processing   
                        with the next input record   
        abort on first reject (default option) --> component stops execution, the graph stops & exits   
                with a failure status -- why that record was rejected   
  
  
==========================================   
  
Order of the rules that you write in a transform does not matter at all -- in terms of performance or logic.   
The transform always executes the rules in the order of the output fields. Topmost output field --> whichever rule(s) assigns to it is executed first.

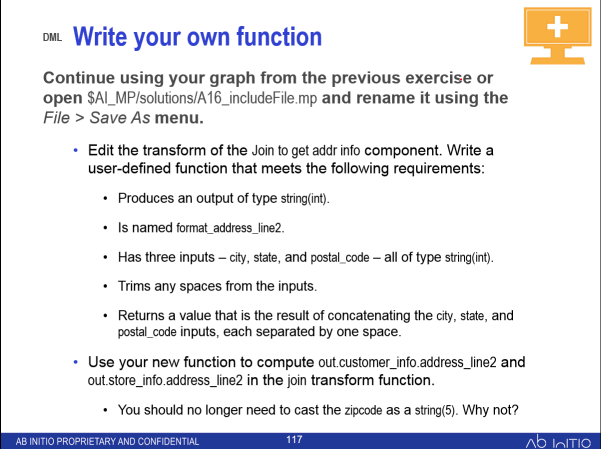
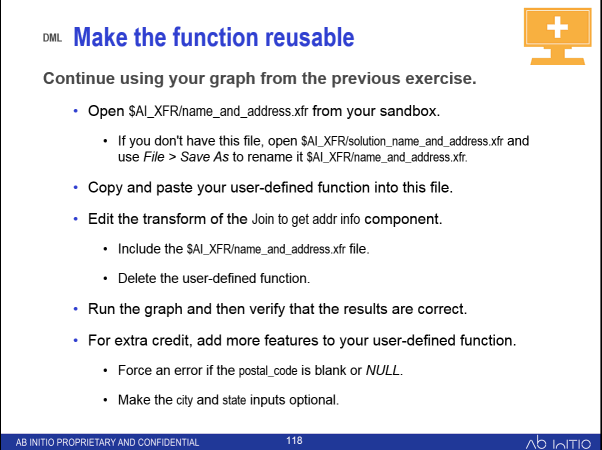
**DAY:2**

Whenever you have 2 or more wildcard rules in the same transform, you need to use priorities to establish the order in which the rules will be evaluated.   
  
out.value :1: in0.value;   
out.value :2: in1.value;   
out.value :: "x";   
  
1 is the highest priority, blank/empty is the lowest (last) priority.   
It evaluates the expression  in0.value --> if it returns a non-NULL value, we're done. We move on to the next output field.   
        if it returns NULL, we check if there is another rule with a lower priority & if yes, we evaluate   
        that expression  in1.value  --> if it returns a non-NULL value, we're done. We move on to the next output field.   
        if it returns NULL, we check if there is another rule with a lower priority & if yes, we evaluate   
        that expression  "x" --> this is never going to be NULL, so we're done -- that's the value.   
  
  
I see a lot of people write lookups like this -- they're worried that the lookup() function won't find a match.   
out.store\_info :: if (lookup\_match("Stores Lookup", in.store\_no))   
                lookup("Stores Lookup", in.store\_no)   
                else allocate();   
  
This is inefficient and it's a poorly designed pattern. The pattern is: let's check if something works and then if it does, we'll do it again for real this time. Issue: the operation gets done twice most of the time, because hopefully most of the time it should work.   
Better pattern: try the thing & deal with whatever value it returns if it doesn't work.   
  
What does lookup() return if it doesn't find a match? NULL   
  
You can rewrite the code above much more efficiently:   
out.store\_info :1: lookup("Stores Lookup", in.store\_no);   
out.store\_info :: allocate();   
  
If there is no match -- lookup() & lookup\_match() do about the same amount of work. In both cases, we check if there is that key value in the Lookup index.   
If there is a match, lookup() does more work because it goes & gets the matching record.   
  
  
We have two very useful functions that work with errors & null values.   
        first\_defined -- equivalent of prioritized rules, it keeps trying expressions until it finds one that   
                        returns a non-NULL value, if it runs out of expressions -- it returns NULL   
        null\_if\_error -- if the expression that you're trying returns a error --> it converts that error   
                        into a NULL value   
                      you can have multiple arguments & it will try each one until it finds one that   
                        succeeds or if none succeed, returns NULL.   
  
There is a difference between getting a NULL value versus getting an error.   
Most functions & operators return a NULL value if any of their input arguments are NULL.   
  
NULL == NULL returns NULL (not true or false)   
but!! if you are doing a component that uses keys (like JOIN or ROLLUP), NULL values are joined or grouped together. In a JOIN a record with a NULL key value will join another record with NULL key value.   
But in expressions, unless the function is specifically designed to deal with NULL values (like is\_null, is\_defined), then NULL as any argument returns NULL.   
  
$ m\_eval 'NULL + 1'   
While executing transform assignment:   
NULL value in assignment for NULL + 1.   
  
$ m\_eval '"a" + 1'   
Error in expression "a" + 1:   
While compiling expression "a" + 1:   
Type mismatch.     
("compilation error" --> While compiling )   
  
$ m\_eval 'read\_file("somefilethatdoesnt.txt")'   
While executing transform assignment:   
Could not read from file "somefilethatdoesnt.txt": No such file or directory.   
 in assignment for read\_file("somefilethatdoesnt.txt").   
("runtime error" --> While executing )   
  
What null\_if\_error() does -- is turn runtime errors into NULL values.   
  
$ m\_eval 'null\_if\_error(read\_file("somefilethatdoesnt.txt"), "x")'   
"x"   
  
$ m\_eval 'null\_if\_error("a" + 1, "x")'   
Error in expression null\_if\_error("a" + 1, "x"):   
While compiling expression  (still a validation error)   
  
  
You can rewrite the code above much more efficiently:   
out.store\_info :1: lookup("Stores Lookup", in.store\_no);   
out.store\_info :: allocate();   
OR   
out.store\_info :: first\_defined (lookup("Stores Lookup", in.store\_no), allocate());   
  
(Aside: allocate() -- sets up an empty value for compound data types like records and vectors. Allocates the memory space necessary for it.)   
  
===============================   
  
At runtime, wildcard rules expand into individual rules for each matching output field.   
So if our lookup file had fields   
        store\_manager   
        address\_line\_1   
        address\_line\_2   
  
and our output record had the same field names.... then this:   
        out.\* :: lookup("Stores Lookup", in0.store\_no).\*   
would expand into three separate rules at runtime:   
        out.store\_manager :: lookup("Stores Lookup", in0.store\_no).store\_manager   
        out.address\_line\_1 :: lookup("Stores Lookup", in0.store\_no).address\_line\_1   
        out.address\_line\_2  :: lookup("Stores Lookup", in0.store\_no).address\_line\_2   
This is searching the lookup file 3x!!!   
  
Wildcard rules are great when you have an input field or a variable that already has a value assigned to it.   
They are not great to use with functions that return records because the function gets called once for each matching output field (recomputing the record value multiple times).   
  
===================================================================   
  
integer is one of the fastest, most efficient data types   
        integer values are represented in binary, as powers of 2   
        integer arithmetic is built-in to the CPUs   
        integer computations can be done in the hardware, very quickly   
  
decimal arithmetic is done in software (C, Java, Python...)   
        slower   
          
  
you can convert dates + datetimes to integer types without any loss of data values   
AND you can continue to do date computations on the values!   
          
$ m\_eval '(integer(4))(date("YYYYMMDD"))"20150421"'   
42113   
$ m\_eval 'date\_month(42113)'   
4   
$ m\_eval 'date\_day(42113)'   
21   
  
integer value of a date is the number of days since Jan 1 1900.   
integer value of a datetime is the number of microseconds since Jan 1 1900 (and this has to be integer(8) to be big enough to hold the value).   
  
===================================================   
  
The + operator is "overloaded"   
        if the operands are numeric types, it adds the values together (sum)   
        if the operands are string types, it concatenates the values together   
  
But what if you try this:   
        (string(""))"1" + (decimal(""))"2"   
--> it fails!   
it doesn't know what to do -- do you want to concatenate or add?   
  
        (decimal(""))"1" + (decimal(""))"2"        --> "3"   
        (string(""))"1"  + (string(""))"2"                ---> "12"   
  
  
We want to concatenate zipcode -- which is a decimal(5).   
        options:   
                ... + (string(5))most\_recent\_store.zipcode   
        OR   
                string\_concat(......, most\_recent\_store.zipcode)   
  
Same thing is happening in both cases!   
        if you write         (string(5))most\_recent\_store.zipcode   
                you are using automatic type conversion to convert the zipcode value to string(5)   
                "casting expression"     
  
        if you write         string\_concat(......, most\_recent\_store.zipcode)   
                the string\_concat function is using automatic type conversion to convert the   
                zipcode to a string type   
  
=======================================================   
  
Every component --> reads a stream of raw bytes on its in port(s) and converts that to records by parsing the data using the input record format that you've provided   
        smaller data --> less time / work to read   
        simpler data --> less time / work to read   
By reducing the size of the data & simplifying the record format, you can improve the performance of all the components in the graph.   
There is a cost to filtering/aggregating data + to changing the record format -- in order for this to be a net benefit, you typically are starting with a very complicated format + lots of data (big benefit from fixing it) such that the cost of fixing it is outweighed by the benefit.   
  
SORT reads the data twice (assuming that it spills to disk) -- once to sort it, and then once again to read it from the spillage files on disk.   
AND! Sort does a lot of comparison operations on the key values to sort the data.   
Of all the components that we have, SORT benefits the most from having a simplified format for the keys + the data as a whole.   
  
  
===================================================   
  
customer\_id in the input data is decimal(10)   
and I convert it to integer(8) before sorting. Why 8-bytes for the integer? The input data could contain a value that is too big to fit into integer(4).   
  
integer is a binary data type, represents whole numbers (no fractional values) as powers of 2.   
Each byte has 8 bits. Each bit can have a value either 0 or 1.   
So if I wanted to represent the value 12 as powers of two, it is 2^3 (8) + 2^2 (4)   
as an integer, we can use 1 byte to represent that value   
        0 \* 2^7 +   
        0 \* 2^6 +   
        0 \* 2^5 +           
        0 \* 2^4 +   
        1 \* 2^3 +   
        1 \* 2^2 +   
        0 \* 2^1 +   
        0 \* 2^0   
00001100 (1 byte) --> 12   
  
Formula that you can use to figure out what is the largest value that you can store in a particular integer   
        math\_pow (2, 8 \* N - 1) - 1   
where N is the number of bytes in your integer   
the -1 inside math\_pow is because you need 1 bit to represent the sign (+ or -)   
the -1 outside math\_pow is because you need a spot to represent value 0   
  
                shortcut        max value   
integer(1)                char        m\_eval 'math\_pow(2, 8 \* 1 - 1) -1' --> 127   
integer(2)                short        m\_eval 'math\_pow(2, 8 \* 2 - 1) -1' --> 32767   
integer(4)                int        m\_eval 'math\_pow(2, 8 \* 4 - 1) -1' --> 2147483647   
integer(8)                long        m\_eval 'math\_pow(2, 8 \* 8 - 1) -1' --> 9223372036854776000   
          
maximum possible value for decimal(10) is                9,999,999,999   
maximum possible value for integer(4) is                2,147,483,647 (smaller!)   
need an integer(8) for the customer\_id   
  
=======================================================   
  
Two things to consider about data types:   
        how "fast" the data type is -- how quickly we can parse raw data stream into this type   
        how "space efficient" the data type is -- how much space does it require on disk or in memory   
                                to store values for this type?   
  
fixed-size type   
        e.g., string(10), decimal(15), date("YYYYMMDD")   
        fastest -- we know exactly how many bytes away the end of the value is   
        most space efficient \*if\* the data values are fixed-size   
        least space efficient \*if\* the data values are variable-size because the smaller values   
                must be padded with spaces, all values use the same amt of space (max size)   
  
record   
   string(50) firstname;   
   string(50) lastname;   
end   
Becky                                            Clayton                                         <-- raw data   
to find the value of lastname -- we just read ahead 50 bytes (don't even look at the values) and then read the next 50 bytes as the lastname value.   
  
  
delimited types   
        e.g., string(","), decimal("\t")   
        slowest because we have to search each character and compare to the delimiter to see if   
                we've reached the end of the value   
        most space efficient type because we store just the value + delimiter (no extra spaces!)   
  
record   
   string(",") firstname;   
   string(",") lastname;   
end   
Becky,Clayton,<-- raw data   
to find the value of lastname -- we have to read each of the characters and compare to "," to find the end of the firstname field "B", "e", "c", "k", "y" and then "," tells us that we've found the end of the firstname, which is the start of the lastname... we then read the characters "C", "l", "a", "y", "t", "o", "n", "," again, comparing each character to the delimiter "," so that we know when we've found the end of the lastname field. A lot more work!   
  
  
length-prefixed type   
        e.g., string(int), string(integer(4)), string(decimal(3))   
        faster than delimited & slower than fixed-size -- because we do have to read and interpret the   
                prefix value but then we can read (without evaluating) the next X characters to   
                get to the end of that field (so we don't have to individually evaluate each character   
                like we do for delimited data)   
        almost as space efficient as delimited because we store just the value + the prefix (no extra   
                spaces!)   
  
record   
   string(decimal(2)) firstname;   
   string(decimal(2)) lastname;   
end   
 5Becky 7Clayton<--raw data   
We read decimal(2) -- first two bytes, value is "5" so we can read (without evaluating) the next 5 bytes. Then we read decimal(2) again, value if "7" so we know that the next 7 bytes are the value of the lastname.   
    
  
I used decimal(2) as the prefix for these two string values. Was that a good idea?   
What is the size of the longest string that I can represent with decimal(2) as my prefix? 99 characters   
What is the size of the longest string that I can represent with integer(2) as my prefix? 32767 characters   
We use integers for string prefixes!   
  
=====================================   
  
For exercise to add REFORMAT to optimize the data types (ex 6)   
        add REFORMAT        dml        edit its out record format   
                        xfr        one rule -- either record assignment or wildcard   
        edit ROLLUP        dml        edits its out record format to keep cust\_id as integer(8)   
                        key        because the input fields are no longer inside DETAIL subrecord   
                        xfr        same reason   
        edit SORT         key        same reason   
  
========================================

**DAY:3**

Let's say that your component sees input data that looks like this:   
800-555-1212   
  
record   
  string("\n") us\_phone\_number;   
end   
View Data --> parses the raw data using the record format that you've provided.   
us\_phone\_number   
800-555-1212   
  
record   
  string("-") area\_code;   
  string("-") exchange;   
  string("\n") number;   
end   
View Data   
area\_code        exchange                number   
800                555                1212   
  
You could do this with REDEFINE FORMAT -- read the data with one format & apply a different format (to the same data!) on the output.   
  
  
==============================   
  
You have data that looks like this:   
20150422   
  
And your input record format is:   
        record   
          date("YYYYMMDD") birthday;   
        end   
  
And you want to convert it to:   
        record   
          date("YYYY") year;   
          date("MM") month;   
          date("DD") day;   
        end   
  
To do this with REDEFINE FORMAT -- all you need to do is supply the output format on its out port.   
To do this with REFORMAT -- you need to do two things:   
        supply the record format on its out port   
        AND   
        in its transform, use either string\_substring() or date\_year/date\_month/date\_day functions   
        to extract each value.   
Either component will work, but REDEFINE FORMAT is faster because it does less work!   
  
In most situations where you could use REDEFINE FORMAT, you might also be able to use REFORMAT -- but REDEFINE FORMAT will always be faster than REFORMAT because it always does less work.   
So if you can use REDEFINE FORMAT instead of REFORMAT, you should!   
  
  
=================================   
  
Decimal, date, and string types are all character-based types.   
Data values are represented with alphanumeric characters.   
If I want to represent the value 12 as a decimal, the raw data contains the characters "1" and "2"   
and you can read those as a string because "12" is a perfectly valid string.   
With redefine format, you can also play around (in a limited way) with the data types for these character-based types.   
  
====================================   
  
  
What does the TRASH component do?   
        drops the data, discards the data   
        sends the data to /dev/null   
  
What does (almost) every component in Ab Initio need to do?   
        parse its input data   
  
Because TRASH doesn't do anything else -- throws the data away -- it is a great way to test how "expensive" it is to parse a particular format.   
  
  
On our rtf-ggx server with about 6 GB of data   
        read as 35 delimited fields, the TRASH used 31 CPU seconds to parse the data   
        read as 1 length-prefixed field, the TRASH used 4 CPU seconds to parse the data.   
  
The "cost" of parsing this data would be the same for \*every\* component in the graph.   
If you had 50 components in the graph that read the delimited record format, your graph would use 1500 CPU seconds just to read the data.   
If you had 50 components in the graph reading the length-prefixed format, your graph would use only 200 CPU seconds to read the data.   
  
The cost of taking the initial data and redefining + reformatting is an overall benefit if you have many delimited source fields + many components that have to read that data.   
  
==================================   
  
Most components in Ab Initio using "pipelining"   
        read a few records from the previous component   
        process those records   
        and send them to the output (to the next component)   
        transfer of records between the component processes is done in "shared memory" (part of   
                your RAM)   
        uses a very small amount of memory because it's only working on a few records at any given   
                time   
        nothing is written to disk --> very fast, very efficient   
  
Like streaming videos --> video is downloaded as you watch it (records are processed as they get to the component) & you can delete what you've already seen!   
  
Regular video -- you have to download the whole video & store it on disk before you can watch it.   
  
When you \*do\* have something that writes data to disk in a graph -- it does slow it down.   
--> try to get rid of the thing that writes data to disk (if you can, not always possible)   
--> try to reduce -- as much as possible -- the amount & complexity of the data being written to disk   
        (simple record formats, smallest amt of data possible)   
  
  
  
=========================   
  
GDE is really really good at identifying graph designs that \*might\* require flow buffering to prevent deadlock.   
Deadlocks don't always occur -- in some complex situations, they are dependent on the timing or the data that you're processing.   
When you run the graph, the Co>Operating System monitors the flows that the GDE has identified & if they start backing up (deadlock starts occuring), the Co>Operating System writes data on those flows to temp files on disk that we call "buffers" or "flow buffers". Flow buffer temp files prevent the deadlock, allow the graph to keep running, and are then read when needed & deleted when no longer needed.   
All automatically!   
  
Flow buffers are great --> they prevent something much worse (deadlock) from happening.   
But in many cases, you can redesign the graph to avoid having deadlock design in the first place!   
  
  
Flow buffers don't always occur before Join.   
Typical design that results in a possible deadlock situation is one where the data is "split" into different flows and then recombined by a component that requires the records to be processed in a particular order.   
(Join needs the records to be key matches -- so it can't process a record until it finds its key match.)   
You also see this with MERGE & CONCATENATE.   
Typically you Partition the data (data is split onto different flows) and then you MERGE/CONCATENATE (data is combined back into a serial flow).   
        MERGE reads the records from the inputs & collates them (reads from the partitions such   
                that the sorted order of the data is maintained)   
        CONCATENATE reads all records from partition0 first, then partition1 and so on.   
                (What if the data from partition0 isn't there yet? Other partitions have to wait!)   
  
===================================================================   
  
If you have a LOOKUP FILE that contains only 1 record, you can use an empty key   
        { }   
  
Then what does the lookup() function look like to access this one record?   
        lookup("Avg Purchases Lookup")   
  
What is the return data type of the lookup() above?   
        record   
  
We want the value of the field avg\_purchases in that record....   
        lookup("Avg Purchases Lookup").avg\_purchases   
  
=====================================================================   
  
The flow buffers that we had -- the one (in0) -- wrote \*almost\* (20 out of 21 MB) all the data on the flow to temp files on disk.   
We redesigned the graph, but now we have a phase break (that we didn't have before) on that same flow. It writes all the data on the flow to disk.   
  
How could the phase break be better than the flow buffer?   
Two aspects to the performance of writing data to disk   
        --> how much data you write (obviously, less data = faster)   
        --> how the data is written: is it written all in one long stream (=faster) or is it written in   
                little bits at a time (=slower)   
  
With flow buffers, we are trying to limit how much data is written to disk. And so we write a small bit of data and we check -- do we need to write more?   
If you only have a small bit of flow buffering --> that's great.   
If you have a large amount of flow buffering --> that's not so great.   
  
With phase breaks, we have to write all the data on the flow to disk. And so we have one big long write --> very efficient.   
  
If you have a flow buffer that writes almost all the data on the flow to disk, it will be slower than the equivalent phase break!   
  
=====================================

Here are the instructions for the exercise -- two slides. Below that, I have some hints on how you can test your function on the command-line (without having to build a graph with specific data to test it).

  
  
  
  
  
**How to test your function.**   
  
Use the "EME Shell Access" shortcut on your desktop to log into the server (rtf-ggx) with your remote desktop user id & password.   
  
Then we need to tell the m\_eval command where to find your user-defined function. Type this command to do so:   
export AB\_INCLUDE\_FILES="<path to your sandbox>/xfr/name\_and\_address.xfr"   
  
I would type this:   
export AB\_INCLUDE\_FILES="/home/ggx/ggx\_rbuchheit/private\_sand/BDS/BDS\_inDepth/xfr/name\_and\_address.xfr"   
  
Then you can test your function with m\_eval. The general form of the command is:   
m\_eval 'format\_address\_line2(<arguments to test>)'   
  
1- Here is my test of the basic functionality:   
m\_eval 'format\_address\_line2("New York", "NY", "12345")'   
        should return        "New York NY 12345"   
  
2- Use descriptive names for your input arguments. This makes the built-in error messages (like this one) easier to understand.   
m\_eval 'format\_address\_line2("New York", "NY")'   
        should return an error message with something like "Missing value for formal argument postal\_code, in calling format\_address\_line2".   
  
3- Here's a test of the extra credit for the city and state inputs being optional.   
m\_eval 'format\_address\_line2(zipcode="12345")'   
        if you've done the extra credit, you'll get a result "12345"   
        if you haven't done the extra credit, it will fail with an error about missing values for arguments.   
  
4- Here's a test of the extra credit for NULL values for postal\_code.   
m\_eval 'format\_address\_line2("New York", "NY", NULL)'   
        if you've done the extra credit, this should output an error message that you supply   
          
  
What happens if the city and state arguments are NULL? The exercise doesn't specify what to do here -- but it does say that the city & state should be optional. So, you could reasonably conclude that it should just return the zipcode:   
m\_eval 'format\_address\_line2(NULL, NULL, "12345")'   
        should possibly (but not specified in requirements) return "12345"

**DAY 5:**

Let's say that I declare the following data type:   
        string(int)[int] addresses;   
can also be written as:   
        string(integer(4))[integer(4)] addresses;   
  
And my vector contains these values [vector "abcdefghijklmnopqrstuvwxyz", "1 Elm St"]   
How much memory/disk space do I need to store these values?   
        "abcdefghijklmnopqrstuvwxyz" = 26 bytes + 4 bytes for the string prefix   
        + "1 Elm St" = 8 bytes + 4 bytes for its string prefix   
        + prefix for the vector = 4 bytes more   
        = 46 bytes total to store these values   
  
What about if my vector contains no values?  [vector ]   
How much memory/disk space do I need to store this vector?????   
        4 bytes to store the prefix value (which is "0") for this vector.   
  
=========================================================   
  
We want to learn how to write loop expressions and loop statements.   
And these expressions can use if expressions and if statements....   
We need to know the difference between an expression versus a statement   
  
if-statement that runs an assignment statement if the condition is true   
        if (is\_defined(in.salary)) rank = rank + 1;   
if the result of the condition is false, nothing happens --> rank just keeps its current value   
  
assignment statement that uses an if-expression   
        rank = if (is\_defined(in.salary)) rank + 1;   
if the result of the condition is false, the expression still needs to return a value -- it returns NULL and the assignment statement becomes (rank = NULL).   
  
  
I can get either behavior from both syntaxes.   
If I want rank to keep its current value if the condition is false...   
        if (is\_defined(in.salary)) rank = rank + 1;   
        rank = if (is\_defined(in.salary)) rank + 1 else rank;   
  
If I want rank to be NULL if the condition is false   
        if (is\_defined(in.salary)) rank = rank + 1 else rank = NULL;   
        rank = if (is\_defined(in.salary)) rank + 1;   
  
=============================================   
  
Here's an expression to compute an output field.   
        out.x :: in.x + if (in.a == 1) in.y else 0;   
  
If I wrote this instead:   
        out.x :: in.x + if (in.a == 1) in.y;   
and the value of in.a was 10. What is the value of out.x?   
        NULL   
if the value of in.a == 10 -- then the if-condition is false and the if-expression returns NULL, so it looks like   
        out.x :: in.x + NULL;   
when you use operators like +, -, etc. with a NULL value, the result is always NULL.   
        out.x :: NULL;   
  
So that's why I used the "else 0" in the original expression above.   
  
  
This is important -- because parameters cannot have NULL values.   
  
If you are writing an expression to compute a parameter value, you must make sure that it won't return NULL. If it's an if-expression, always include an "else part". If it's some other type of expression, consider using the first\_defined function   
        first\_defined() --> evaluates its input arguments one at a time until it finds one that does not   
                return NULL.   
        first\_defined( in.end\_date - in.start\_date, today() - in.start\_date, -1)   
  
          
==============================================   
  
We have this expression to compute a parameter value:   
        $[ if (SORT\_DATE == "Yes") 1 else 0]   
  
You can simplify this expression so that there's less typing. Does anyone know how?   
        $[ SORT\_DATE == "Yes" ]   
  
This is a conditional expression -- it uses a comparison operator (==, or > or < or >= ....) to compare two values. If the conditional expression is true, it returns 1 and if it's false, it returns 0.   
  
================================================   
  
When to use block expressions?   
        begin --> end   
Whenever you need to use statements as part of the code to compute a parameter.   
  
If you have a simple expression to compute the parameter value -- you don't need begin-end because you can just assign the expression directly to the parameter.   
  
But if you need to do more complicated things like declare variables or use statements -- then you need begin-end because you cannot use a statement to compute a parameter value. Parameter value must always be computed with a single expression (it can be very complicated -- but it has to be a single expresison that returns 1 value)   
A parameter cannot have 2 values --> you need an expression to compute it.   
  
begin-end with result:: is just a very complicated expression that returns a single value (the value of "result").   
  
=========================================================   
  
Three common types of expressions --   
        if expressions   
        conditional expressions   
        begin-end (block expressions)   
  
You can also have if-statements and begin-end statements.   
  
The way that you identify whether something is an expression or not is...   
        if it follows = or :: --> it's an expression   
        or if you are computing a parameter value --> it's an expression   
        the argument to m\_eval has to be an expression   
  
  
m\_eval --> displays the data using a command-line View Data (m\_dump)   
m\_eval -raw --> displays the raw data as-is without any data format   
m\_eval -cpu --> tells you an approximation of how much cpu time it took to do the computation   
  
  
m\_eval AND the DML expression pane (F8) in Expression Editor have no "context"   
        they cannot see your input fields or any variables in your transform   
        they cannot see the input data because you aren't running a graph   
  
If you want to test expressions against input data, you need to run the graph & use the Breakpoint Debugger.   
  
If you want to use variables in your expression, you need to use a begin-end block expression to keep them together.   
  
m\_eval 'begin   
        let alphabet = for (let i, i < 26) : char\_string(i+65);   
        result :: for (let letter in alphabet) : string\_downcase(letter);   
end'   
  
In the DML Expression Pane, you can enter multiline expressions by using Ctrl-Enter to go to the next line.   
Ctrl+<up arrow> retrieves the previous expression for editing   
  
===============================   
  
So "number" is initially a string value -- we cast to a decimal so we can add it:   
        (decimal(3))number + 1   
but you don't specify an output type for the result of the expression   
so, we take a 3-digit decimal (number) plus some unknown decimal type (1) and the result is an unknown decimal type.   
  
        (decimal(3)) ( (decimal(3))number + 1 )   
this takes the result of the expression number + 1 and casts the entire result as a decimal(3)   
the parenthesis around the expression ( (decimal(3))number + 1 ) applies the casting (decimal(3)) on the outside to the whole expression.   
  
  
We want to play around with numbers in a vector using m\_eval or Evaluate DML Expression in the GDE   
  
for (let number in string\_split("100,200,300"", ",")) : (decimal(3)) ( (decimal(3)) number + 1)   
---> string\_split returns a vector of strings, the data type is string(int)[int]   
---> to do math on those strings requires a lot of casting....   
  
for (let number in [vector 100, 200, 300]) : number + 1   
---> if you don't put quotes around the number, the default data type is "integer"   
---> [vector 100, 200, 300] -- the syntax is called a "vector constructor" and it returns a vector of integers   
  
  
There are constructors for both vectors & records   
        [vector "a", "b", "c"]   
and this constructs a record with two fields (firstname & lastname) with the values "Becky" & "Clayton"   
        [record firstname "Becky"   
                   lastname "Clayton" ]   
  
        [record firstname string\_upcase(in.fname)   
                            lastname string\_downcase(in.lname) ]   
  
The values that you supply for the vector elements or field values in the record can be constant values or they can be any expression -- you put an if-expression in there if you wanted!   
        [vector 0+1, 1+2, 3+4]   
  
  
  
===========================================   
  
Compute the vector :   
        [vector 100, 200, 300, 400, 500 ]   
  
with an index for loop expression:   
        for (let i, i < 5) : (i + 1) \* 100   
  
with a while loop expression   
        begin   
           let i = 0;   
                   result :: while (i < 500) i = i + 100; : i;   
        end   
  
  
================================   
  
Loop expressions always compute a vector as a result. Always.   
  
Loop statements can compute any value that you want. It can be a simple value like a decimal or a string. You can build up a string value by concatenating values together. (There is a better way to do this one!)  You can get a total by summing values for each iteration.   
  
If I wanted to build up a string by concatenating a series of values in a loop... I could write a for loop statement....   
        let string(26) alphabet\_string = "";   
        for (let i, i < 26)   
                alphabet\_string = alphabet\_string + char\_string(i+65)   
          
Or I could write a for loop expression that builds a vector and then converts the vector into a string   
        string\_join( for (let i, i < 26) : char\_string(i+65), "")   
(This is better & faster than concatenating strings)   
  
Big difference in syntax between loop expression & loop statement:   
        loop expression always comes after = or :: or is used to compute a parameter value   
        loop expression has ":" to indicate the expression that computes the current output vector   
                element   
  
        loop statements are standalone & they don't have ":"   
  
Many of the things that you might want to do with a for loop statement can more easily (and more quickly!) be done with a vector function. Check to see if there is already a function that does what you -- before you write a loop!   
Search Help for "vector functions" to see an alphabetical list   
  
==================================================   
  
Your accounts on the training server are active until August 31 2015. You can log in at any time, from anywhere (doesn't have to be your office!) to continue exploring the software installed on the server, check out the Help Library, continue working with these slides (there are 2 more lessons!) and this sandbox.   
  
If you have general questions -- please feel free to send me an email. I'm always happy to hear from students!   
        rbuchheit@abinitio.com   
  
If you have questions about something specific for a customer project -- please contact support@abinitio.com (they'll be better equipped to answer you more quickly & thoroughly)   
        let them know what customer you're working for   
  
=====   
begin   
  let i = 2;   
  let j = 0;   
  result :: while (i < 6) j = i + 2; : j;   
end   
  
This is an infinite loop. You are never increasing the value of i -- so your while loop condition is never false. You get a "bad allocation" error because we run out of memory trying to allocate the result vector.   
  
The way to fix it is that you must also increment i by some value so that eventually it gets greater than 6.   
begin   
  let i = 2;   
  let j = 0;   
  result :: while (i < 6) begin j = i + 2; i = i + 1; end : j;   
end   
  
The begin-end here enables you to have 2 (or more) statements that you can run for each loop iteration.   
  
=====   
  
Loop expressions \*always\* assign to the output vector in the order of the elements.   
        element 0 is always assigned first, then element 1 and so on.   
        you cannot skip elements -- but you can have blank elements.   
  
If you wanted every other element to be blank...   
        for (let i, i < 10) : if (i % 2 == 1) (decimal("")) i else ""   
i % 2 either returns 0 or 1 (false or true) --> a false returns a blank element ""   
    
  
There is no do-while loop in Ab Initio.   
Do-while loop is basically a shortcut for doing the thing once & then looping.   
        In DML, you just do the thing once & then loop it with a regular while loop.

**Express IT**

Log into Remote Desktop Connection   
  
Start Internet Explorer from your remote desktop shortcut   
[http://rtf-ggx:6563/bre\_training?appIdentifier=](https://owanj.galaxysi.com/owa/redir.aspx?SURL=VW2ySvxbuowrzApx9BpFBBb-yk8tp42ac1Oj2E8kNCpc0uCGGE7TCGgAdAB0AHAAOgAvAC8AcgB0AGYALQBnAGcAeAA6ADYANQA2ADMALwBiAHIAZQBfAHQAcgBhAGkAbgBpAG4AZwA_AGEAcABwAEkAZABlAG4AdABpAGYAaQBlAHIAPQA.&URL=http%3a%2f%2frtf-ggx%3a6563%2fbre_training%3fappIdentifier%3d)<yourid>   
  
        example:   
                [http://rtf-ggx:6563/bre\_training?appIdentifier=ggx\_jdoe](https://owanj.galaxysi.com/owa/redir.aspx?SURL=DziM_uv_qXC9WmL1AYfjf9-AOMBuoEqz37680q1QZuxc0uCGGE7TCGgAdAB0AHAAOgAvAC8AcgB0AGYALQBnAGcAeAA6ADYANQA2ADMALwBiAHIAZQBfAHQAcgBhAGkAbgBpAG4AZwA_AGEAcABwAEkAZABlAG4AdABpAGYAaQBlAHIAPQBnAGcAeABfAGoAZABvAGUA&URL=http%3a%2f%2frtf-ggx%3a6563%2fbre_training%3fappIdentifier%3dggx_jdoe)   
  
        [VERY IMPORTANT: IE won't allow you to bookmark this & it won't save it in your history.   
                You need to type every character yourself every time you need to enter the URL]   
  
Username: <same as your remote desktop id>   
Password: <same as your remote desktop password>   
  
You should see a list of 3 applications + several existing configurations.   
If you have 0 applications visible —> then you typed in the URL incorrectly. You will need to quit IE, restart IE and type the URL correctly.

**Express IT**

**DAY 1:**

Class is Monday through Friday   
        4:30 PM IST — 8:30 PM IST   
        we’ll take a 20 minute break about halfway through each day, around 6:30 PM   
  
This class is a bit longer than 20 hours —> in order to be able to finish all the material, there will be “homework” each night where you’ll spend some time working on hands-on exercises.   
This should between 30 minutes to an hour each day — you can complete the homework after class or before the class the next day.   
  
The PPT slides for this week’s class are on your desktops   
        BRETraining.pdf   
  
If you have questions   
        you can send them at any time by sending me a chat message in GoToMeeting   
        or   
        I’ll periodically pause and ask if anyone has questions & you can ask then   
                on the phone   
  
============================   
  
In Express>It ACE (Application Configuration Environment)   
  
You start with a generic graph in the GDE   
        the components in the graph are configured to use parameter values   
        for their record formats, data location URLs, transforms, etc.   
        parameters are declared as graph parameters   
  
You run a graph like this by creating a pset to supply values for the graph parameters, and then you run the pset.   
  
In the GDE, you can create a pset using the Input Values Editor.   
  
You can have as many psets as you want for a particular graph.   
But each pset is associated with exactly 1 graph —> a pset cannot be reused with other graphs. psets are specific to the 1 graph that they were designed for.   
  
  
ACE provides an web interface for analysts to create psets. They don’t have to learn how to use the GDE.   
  
Someone (often the developer who designed the graph) creates an “application template” —> this is a user interface that is designed to help the analyst fill in appropriate, valid values for the pset.   
designing a graph —> skill: data flow programming   
designing an application template —> skill: user interface design, you are adding & setting up UI controls to create a template that aids the end user in creating a valid pset.   
  
Someone else (often a data or business analyst) creates a “configuration” for that application template —> a configuration is simply choosing values for the UI controls in the template. Values that are chosen are used to write/compute the values that are saved in the pset.   
  
Each graph can have as many application templates as want (but usually there is only one)   
Each application template can be associated with 1 or more graphs/plans (but usually there is only one)   
Each application template can be used to create 1 or more configurations (and usually there are \*many\* configurations for a particular application template)   
Each configuration —> pset that can be used to run the graph associated with its template.   
Each configuration is specific to exactly 1 template.   
  
  
In ACE configurations   
        Save —> save the configuration & the pset that is generated from the   
                        configuration to your sandbox   
        Save & Publish —> save the configuration & pset to your sandbox   
                                and also check them both into the EME TR project.   
  
  
==============   
  
There are two types of rulesets:   
        embedded —> created via a BRE Mapping UI control in an ACE configuration   
                                        ruleset is used to generate a transform which is embedded   
                                        in the pset that is generated from the ACE configuration   
                                        that transform can be used only in the graph associated with   
                                        that pset (which is associated with a particular application   
                                        template)   
  
        standalone —> created via the Express>It “mapping” or “BRE” interface   
                                you configure the inputs, outputs via the BRE interface   
                                then create mapping expressions & rules   
                                mapping expressions & rules are used to generate a transform which   
                                is then saved in an .xfr file   
                                .xfr file can be used in any graph that uses the same inputs & outputs   
  
===============   
  
For ACE   
— you start with a generic graph that uses input parameters to configure the components   
— a developer (often the person who designed the generic graph) creates an application template in ACE that guides end user through setting parameter values   
— the end users (often a data/business analysts) use ACE to create configurations for the template, each configuration sets different values for the UI controls   
— values in each configuration are used to generate a pset   
— each pset can be used to run the graph   
  
One of the UI controls that you can add to a template is BRE mapping control (BusinessRulesPopup). End user writes a ruleset in the BusinessRulesPopup —> ruleset is used to generate a transform, value of that transform becomes the value of an input parameter (in the pset) for the graph.   
—> rulesets that you create with the BusinessRulesPopup are called “embedded” rulesets.   
  
Can we think of the template as a generic graph and configs as psets?   
That’s a bad idea.   
If you design the template as if it were a graph —> then you might as well just teach the analysts how to use the GDE. Template should be a well-designed user interface that guides the end user through selecting valid values for the parameters. Design of the graph is irrelevant to the design of the template. Because the template is only dealing with parameters.   
If you think of the template as being related to the graph design —> you will end up with a very poor template design.   
You want to think of the template as a way to configure parameter values.   
Configuration (created by the end user) is a series of choices & values that the end user makes in the template UI.   
Values in the configuration might not be written directly to the pset. The template might be designed to do computations on the configuration values before assigning them to the parameter values in the pset.   
  
Template that I showed in the demo is a perfect example of this.   
Where did you see me select a record format? I didn’t.   
Where did you see me provide a physical path for the input file? I didn’t.   
  
The template was designed to give the user a limited list of EME dataset objects and then compute the record format & the physical path for the input file based on the properties of the EME dataset object.   
  
Idea —> end user cannot make a mistake where he/she chooses a record format that is incompatible with the physical data. Better because it is easier for the end user & it results in compatible/valid values.   
  
This is much better than having the end user than having the user select a record format from all the possible DML files & a physical path from all possible filesystem locations —> but that’s what you’d end up with if you designed the template to mimic the graph design.   
  
===========================   
  
for standalone (BRE) ruleset   
— developer creates a graph (possibly not a generic graph, could be hardcoded) in the GDE and realizes that the graph has complicated mapping logic   
— someone (usually the graph developer) creates a new ruleset in BRE & configures the inputs, outputs to match the input DML & output DML of the transform component in the graph   
— someone else (often a data/business analyst) writes the ruleset : defines the logic for mapping the input to the output target   
— ruleset is used to generate a transform, saved as an .xfr file   
— .xfr file can be used as the transform for a component in this graph or for any other graph can use the same inputs & outputs (specifically, those same record formats).   
  
Think about a REFORMAT component   
        you need input record format (in port)   
        you need output record format (out port)   
        transform that assigns values to each of the output fields — can be generated   
                via BRE ruleset   
  
==========================   
  
With ACE & embedded rulesets   
        there is both   
        Save —> saves to the sandbox only, your own local copy   
        Save & Publish —> also checks in to the EME TR   
  
With BRE & standalone rulesets   
        there is not an option to have a local-only copy of the ruleset/ xfr   
        rulesets are never exported to the sandbox anyway — saved in the EME TR   
                only   
        generated .xfrs also cannot be saved local-only; there is only the option   
                to save the EME TR   
        (there is no option to save to a sandbox — you need to check the project   
        from the TR to get the .xfr into your sandbox)   
  
=============================   
  
This class:   
        — we’ll learn how to write an embedded ruleset   
        — we’ll learn how to set up and write rulesets via standalone BRE   
  
but we will not have time to teach you how to write templates (that’s a separate 5 day class itself!)   
If you’re curious about how to write templates   
        the Help Library is an excellent place to start (it has a tutorial & some simple   
        demonstrations)   
  
================================   
  
Why use Express>It?   
Why write rulesets instead of transforms directly?   
  
If you have a generic graph and you want business analysts to be able to configure the parameters —> ACE templates make that much easier.   
  
If you don’t have a generic graph but you do have a situation where you have very complicated business logic (specifically something that is difficult for non-domain experts — like developers) to understand —> BRE rulesets enable the person who designed the logic (analyst) to write the rules directly   
        —> provides a helpful environment for working with complex logic   
        — ruleset lineage views   
        — helpful testing mechanisms per record & for whole datasets   
  
====================================   
  
In a ruleset, when you start typing, we try to do an autocomplete — we give you a list of possible completions for the variable/function/parameter name that you’ve started typing.   
  
It’s a good idea to use autocomplete because it formats the variable correctly.   
Start typing, when autocomplete gives you a list of completions, use your mouse to click the one that you want to select & then hit Enter key —> makes the selection and fills in the variable/function name for you   
  
For example, using Autocomplete, my expression to compute per\_unit\_price is   
total\_price / in.quantity   
  
Why is it just “total\_price” but it is “in.quantity”?   
They are both fields from the input record format  — we call them “input variables”.   
Hint: take a look at the output variables (list of fields from the output record format)   
  
total\_price is a name that is found only in the input variables   
the name “quantity” is found in both the input variables and the output variables   
  
you can use output variables to compute the values of other output variables   
you need to specify which value to use!   
  
in.quantity —> fully qualified name   
                                        name + where it came from   
total\_price —> short name   
                        because there is no other variable with the name total\_price   
                        we can just use it as-is   
  
  
========================   
  
I mentioned that — for simplicity — BRE has just 4 data types   
        number   
        string   
        date / datetime   
        record   
  
BRE makes these data types visible for internal variables   
but the underlying data types from the DML record formats are known to BRE   
because what do you need for a minimum ruleset?   
        input variables (from input DML record format)   
        output variables (from output DML record format)   
  
BRE automatically maps the DML data types to the available BRE types.   
Don’t worry about this.   
  
========================   
  
The Computed Values column in the mapping view is visible only once you have a valid value/expression/rule assigned to each output field.   
  
Recommendation for workflow:   
— enter placeholder values like 0 or “” for each of the output fields that need values   
— then go back and configure the more complicated expressions to do the correct computation for each output field   
  
enables you to focus on one output variable at a time — you can write the expression and immediately see the result (without having to do all the expressions for all the fields at the same time).   
  
=========================   
  
First of all — not all DML functions are \*visible\* in BRE   
        some of those because you can’t use them   
        others because we feel that they’re too difficult for analysts to use   
          
My guess is that string\_split() works — but remember that it’s a function that produces a vector as output — so your output data has to have a vector/List field in the output record format.   
  
Many of the functions that you cannot use directly in BRE (most of them related to lookups) can be wrapped inside a user-defined function and then included in a ruleset so that analysts can use that function   
        — idea is that you as the developer make the built-in function easier to use   
                by setting up & specifying the more difficult parts   
        — internally the user-defined function uses these more difficult functions   
        — analysts uses your helpful function directly   
  
Example:   
        lookup\_count() and lookup\_next()   
        if you want to return a vector that contains a list of all the matching records in a lookup — you use lookup\_count to figure out how many matches & then lookup\_next to cycle through them inside a for loop   
        that’s way too difficult to expect an analyst to do on their own   
        so these functions are not visible in BRE   
  
        if you wanted to make that functionality available   
        —> you (the developer) could create a user-defined function that hardcodes the information and does the looping and the BRE-user uses your user-defined function.   
  
        string(int)[int] out :: lookup\_all\_state\_abbrev (string(int) state\_name) =   
        begin   
                let num\_matches = lookup\_count(“States Lookup”, state\_name);   
                out :: for (let i, i < num\_matches) : lookup\_next(“States Lookup”).abbreviation;   
        end           
  
===========   
  
When editing an embedded ruleset   
        there is an OK button at the bottom right corner   
        that saves the ruleset that you’ve defined \*in the configuration in your current web   
                session only\*   
  
That also takes you back to the configuration page where you can then   
        Save   
or   
        Save & Publish   
the configuration (which includes the ruleset value + all of the other values you’ve entered into the configuration)   
  
  
Ending class today —   
        click OK to save your ruleset in the configuration   
        click Save & Publish to save & check in your configuration

**DAY 2:**

in BRE, there are three types of rulesets   
        REFORMAT — by far, the most common — what we’re talking about all week   
        ROLLUP — rarer — we’ll talk about how this works on Friday   
        VALIDATION — rarer — same thing here   
  
Notice that there are not rulesets for join, scan, normalize, etc.   
That’s because BRE is about simplicity —> these other types of components are more difficult to understand.   
  
Often a graph that uses BRE Ruleset will have “extra” reformat component — where you have in the graph design “simple” transforms in Join, Scan, Normalize, Rollup, Filter by Expression to get the data the way you want it to be — filter out records you don’t want, to expand vectors or create vectors, to join from other data sources, etc.   
\*Then\* after the data is the way you want it to be — you add Reformat — which is going to use complicated “business-logic” transform generated from the ruleset.   
  
To certain extent — depending on your file sizes — you can get around the restriction on joining by using lookup files. Assuming that the other data sources are small enough files, you can get join behavior this way.   
  
=====================   
  
Think about using a LOOKUP FILE in a graph. What parameters do you need to configure in the component?   
—> record format   
—> key   
—> data location URL   
—> “name” — label of the lookup file component   
  
To access data from a lookup, you call the lookup() function. What are the arguments to that function?   
—> expression that matches the key value from the LOOKUP FILE   
                        called the “Default Key” in a ruleset   
—> “name” — label of the lookup file component that you want to access   
  
lookup(“Stores Lookup”, decimal\_strip(in.store\_id))   
lookup(“State Lookup”, “PA”)   
lookup(“Regions Lookup”, in.state)   
        expression that matches the key value can be a constant, a variable or fieldname,   
        a parameter value, it can use a function or operator to compute values, etc.   
  
  
One syntax that you might see in BRE for accessing lookup file data values is:   
        State Regions.region(in.state)   
                State Regions —> name of the lookup in the ruleset   
                .region —> field value that you want to return as your result   
                in.state —> value/expression that is going to match the key values in the lookup   
  
        —> use this syntax when you either have not defined a default key in the ruleset   
        OR you want to override the default key in the ruleset   
  
Another syntax is:   
        State Regions.region   
                State Regions —> name of the lookup in the ruleset   
                .region —> field value that you want to return as your result   
  
        —> use this syntax when you have declared a default key in the ruleset   
                current lookup record is automatically matched to the current value from the   
                default key   
  
======================   
  
To a large extent — it is a bad idea to spend time trying to optimize the expressions and rules in a ruleset.   
For two reasons:   
        —> when we generate the transform from the ruleset, we do quite a bit of optimization there (we avoid calling lookups multiple times, we avoid recomputing values multiple times, etc.)   
        —> performance optimization usually makes the ruleset more difficult to read and understand   
  
Whole point of BRE is to create a ruleset that is \*easy\* and transparent to write/test/debug.   
  
Avoid premature optimization —> get the ruleset written correctly, generate a transform, check the performance AND then only if the performance of the generated transform is not up to you need, then consider taking a look at optimizing the ruleset (definitely contact support@abinitio.com for assistance with this — we want to help you with this so that we can automate it in the future!)   
  
=======================   
  
  
The record testing values (Computed Values column) are computed on-the-fly as you switch to each different test record.   
  
The file testing values are computed by secretly generating a transform and creating a graph to run it — but the data values are saved in a temp file (which is deleted when you quit the BRE or when you change to a diff configuration or ruleset)   
Any time you change how that transform or graph is designed (change the ruleset, add filtering conditions to the data in the file test, you change the inputs/outputs/lookups) —> you would need to click the “Run” button to re-run the test using the changed values.   
  
  
Whenever you run a file test, you have the option of saving the temp data (result from the file test) into a more permanent file called the “baseline” file.   
In general, you can have multiple baselines for each ruleset by having multiple deployments (set of values for working with test data in a ruleset — we’ll talk more about those later).   
If you have just one deployment —> then you’ll have only 1 baseline file for that ruleset.   
  
Each time you run a file test, you can choose to either keep the results temporary or save those particular results into the baseline file.   
This enables you to then do a simple type of regression testing   
        —> run a file test, save the results into the baseline file   
        —> make some changes to the ruleset   
        —> run the file test again & the BRE will show you (by marking them with gold stars)   
                any values that are different in this test output versus what you have saved in   
                the baseline   
  
  
  
For record testing, you have options about whether you want to   
        Compute & Validate   
        Validate only   
and whether you want that to be   
        Automatically   
        Manually   
  
I tend to use Validate only “manually” —> when I am first creating a new ruleset so that I don’t get annoyed with all the error messages as I am still setting up   
I would use Validate only “automatically” —> in a situation where I didn’t have test data for some reason (so I don’t get annoyed with errors about missing test data)   
I tend to use “Compute & Validate” manually —> when I have a complicated ruleset or for some reason the testing is slow where re-computing the output takes a while   
99% of the time, I use the default — Compute & validate automatically.   
  
====================   
  
BRE has a ruleset lineage viewer.   
—> not the same thing as dataset lineage like you see in the GDE and EME   
—> ruleset lineage is limited to just the current ruleset   
—> if you want to see dataset lineage that includes the rules/mappings in the ruleset : generate a transform for the ruleset and then use it in a graph and do dataset lineage from there   
  
You can view lineage for   
— the whole ruleset or specific rules in the ruleset to gives you an idea of the relationships and dependencies of the variables/rules in the ruleset   
— source variables to see how they are used in the ruleset   
— target variables to see how the ruleset computes them   
  
  
Lineage is available in both embedded (rulesets created in ACE configurations) and also standalone rulesets.   
I just didn’t show in ACE because we don’t have any interesting/complicated rulesets in our configurations.   
  
===================   
  
In many rules (not all) — the number of different output values that you have = number of cases that you need.   
  
Here’s some simple conditional logic —> let’s turn it into a rule   
  
if Gender is female and the Age < 12 then Category is “girl”   
else if Gender is male and the Age < 12 then Category is “boy”   
else if the Age < 12 then Category is “child”   
else if the Age < 18 then Category is “teenager”   
else if Gender is female and Age < 65 then Category is “woman”   
else if Gender is male and Age < 65 then Category is “man”   
else if Age < 65 then Category is “adult”   
else if Age >= 65 then Category is “senior”   
else Category is “unknown”   
  
What are my triggers? gender, age   
What is my output? category   
How many cases do I need? 9 (8 cases + 1 default case)   
  
My rules grid would look like this:   
                                triggers                                        output   
                                Gender        Age                        Category   
case 1                        “F”                        < 12                        “girl”   
case 2                        “M”                        (same)                “boy”   
case 3                        (any)                (same)                “child”   
case 4                        (any)                < 18                        “teenager”   
case 5                        “F”                        < 65                        “woman”   
case 6                        “M”                        (same)                “man”   
case 7                        (any)                (same)                “adult”   
case 8                        (any)                >= 65                        “senior”   
case \*                        (any)                (any)                        “unknown”   
  
  
Notice how I write the “comparison cells”   
        “F”        —> this tests the condition (Gender == “F”)   
                                the equality test == is implied; it’s the default comparison that we do   
                                this compares the value “F” to the value of the Gender field in the   
                                        current input record   
                                if they’re equal —> it returns true   
                                if they’re not equal —> it returns false   
                                if the Gender value in the current record is NULL —> it returns NULL   
        < 12 —> this tests the condition (Age < 12)   
                                this comparison cell supplies the comparison operator (< ) to use   
                                this compares the value 12 to the value of the Age field in the current   
                                        input record   
                                if Age < 12 —> it returns true   
                                if Age >= 12 —> it returns false   
                                if the Age value in the current record is NULL —> it returns NULL   
        (same) —> is a shortcut that means “the same comparison as the previous   
                                        case for this trigger”   
                                the (same) for the Age trigger in case #2 —> (Age < 12)   
        (any) —> is a shortcut that means any value (including NULL) returns true for   
                                        this comparison   
                                it is saying don’t apply this trigger to this case   
  
  
The rule that we are looking at above is called a “single fire” rule. There is another type of rule called “multi fire” that we’ll talk about later this week.   
Single fire rules implement if-else logic —> “Only the first true case fires.”   
What does that mean?   
  
A case “fires” when all the trigger conditions for that case are true.   
The triggers are AND logic.   
        Gender == “F” AND Age < 65 —> case #5 fires   
  
When a case “fires” —> we use the output expression for that case as the value for the output variable.   
If case #5 fires, the output expression for that case “woman” is assigned as the value of the output variable (Category).   
  
The BRE evaluates the trigger comparisons for the first case.   
        If all the conditions are true, we’re done —> we take the output expression for   
                that case and assign to the output variable and then we move on to the next rule.   
        If any of the conditions are NULL or false —> then we move on to the next case   
                We evaluate all the trigger conditions for the second case.   
                        If all the conditions are true, we’re done —> we take the output expression for   
                                that case and assign to the output variable and then we move on to the next rule.   
                        If any of the conditions are NULL or false —> then we move on to the next case   
                                We evaluate all the trigger conditions for the third case   
                and so on until either we find a case that fires OR we get to the default case.   
  
If case #5 fires —> cases #1 through #4 were all false or NULL   
                                            all the trigger conditions in case #5 were true   
                                            cases #6 through \* were never evaluated.   
  
  
  
  
Anything can be a trigger   
        — input variables   
        — lookup variables   
        — output variables (BRE automatically designs the transforms to evaluate the rules in an   
                                                        order that is correct based on dependencies between rules)   
        — internal variables   
        — parameters   
        — expressions (you can write an expression that is computed and then compared to the   
                                        comparison cell value)   
  
For outputs, you can compute values ONLY for:   
        — output variables   
        — internal variables   
  
  
In Ex2\_Use\_Loyalty\_Lookup configuration, create a rule to compute the “name” output field with this logic:   
if Gender is female and the marital status is married   
        then name is “Mrs. “ + first name + “ “ + last name   
else If Gender is female and the marital status is unmarried   
        then name is “Miss “ + first name + “ “ + last name   
else if gender is male   
        then name is “Mr. “ + first name + “ “ + last name   
else first name + “ “ + last name   
  
My triggers are: gender, marital status   
My output is: name   
I need 4 cases (3 cases + 1 default)   
  
How can I find out what sort of values are in the data for the Gender and Marital Status columns?   
Quick Profile   
  
Gender is female—>        “F”   
Gender is male —> “M”   
marital status is married —> “1”   
marital status is unmarried —> “0”   
  
The 1’s and 0’s are enclosed in quotation marks because the marital status variable is a string data type. When we compare string values, we put constant values like 1, 0, M, F inside quotation marks.   
  
========   
  
What if you wrote a new rule in your ruleset and you ran a file test and you noticed that one of the cases fired 0 times… what are the possible reasons for that?   
If the case fired 0 times, it means that case was never true.   
How could it happen that a particular case is never true? What could cause that?   
1- If you don’t have any records in the input test data that “exercise” this condition. There are no input records where the trigger conditions are all true.   
Solution: add one or more records to your test data set so that when you run the file test, this case is fired.   
  
2- Your logic in the rule is incorrect and you have written a case that is “unreachable”.   
  
  
Here’s an example of an unreachable case:   
                        trigger                                   
                        Age                        Gender   
case 1                < 18                        any   
case 2                < 18                        “F”   
case 3                > 18 and < 7        any   
  
case #2 is unreachable because for any record that it is true for (Age == 16 and Gender == “F”, for example), case #1 is also true. Case #1 fires first and we’re done.   
  
case #3 is also unreachable because you cannot have a numeric value that is both > 18 and also < 7   
  
For simple cases like these — the BRE actually can often identify them and warn you about them before you even run a file test.   
But for more complicated logic — you would see this in a file test where times fired = 0   
  
=================================   
  
OR logic   
  
The way that rules work is that \*all\* of the trigger conditions for a case must be true for that case to fire. “AND” logic.   
like if Gender is female AND the marital status is married   
  
What if we need to implement OR logic?   
There are two types of OR logic that you might run into   
— where you are comparing two different values for the same trigger   
                < 12 or > 65   
                “F” or “M”   
This one is simple — you can use the “or” operator as part of your comparison cell.   
You can do the same thing with “and”   
                > 18 and < 65   
                  
  
— where you are testing one trigger condition or a different trigger condition   
                if credit limit > 10000 or salary > 1000000   
                if status is gold or date\_started is before Jan 1 2000   
For these situations, often the simplest design is to just create multiple cases for that output value, using (any) for the other triggers that you don’t want to test for that case. See example below   
  
if status is gold or date\_started is before Jan 1 2000 then decision = ‘accept’   
else if status is silver or date\_started is before Jan 1 2010 then decision = ‘maybe’   
else decision = ‘reject’   
  
                        triggers                                                        output   
                        status                        date\_started                decision   
case 1                gold                        (any)                                “accept”   
case 2                (any)                        < “2000-01-01”        “accept”   
case 3                silver                        (any)                                “maybe”   
case 4                (any)                        < “2010-01-01”         “maybe”   
\*                        (any)                        (any)                                “reject”   
  
  
=======================================   
  
Tomorrow:   
        — standalone BRE   
                        how to set up inputs, outputs, lookups   
        — internal variables   
        — named constants   
        — more work with writing more complicated rules   
                        expression triggers

**DAY 3:**

What are the minimum three things we need in order to have a valid ruleset:   
input variables   
output variables   
expressions and/or rules that assign valid values to each output variable   
  
================   
  
When you configure input variables, output variables, lookup variables in a ruleset, you choose an EME Dataset Object to do so. You do \*not\* use a DML record format.   
The EME Dataset Object enables us to link additional metadata from the BRE to that dataset (business names, comments, etc.)   
  
How do you get an EME Dataset Object?   
Because BRE is designed to be used by analysts — you cannot create a dataset object directly in the BRE. It must already exist before you can use it in a ruleset.   
  
1- If you have a graph that uses that dataset (file, table, etc.) and you check-in the graph & do dependency analysis (DA), then DA creates an EME Dataset Object for you.   
  
2- You can bulk import dataset objects from db table schemas, logical models (like Erwin, Oracle Designer), or spreadsheet descriptions either using the “Connector” graphs (available in the GDE Organizer) or using Metadata Hub.   
  
3- If you use Data Profiler product (not Quick Profile), any datasets that you profile will have EME Dataset Objects created for them.   
  
4- There are components in the Organizer (Load File Dataset, Load Table Dataset) that you can use to build a graph that loads dataset objects into the EME. (This is what the “Connector” graphs do.)   
  
5- There is an EME command “air reposit dataset file” that you can use to create file datasets in the EME TR.   
  
  
  
EME dataset objects do NOT contain any actual data.   
They are quite small because they only collect \*metadata\* about the dataset (record format, how the dataset is used in graphs, business names for the fields in the BRE).   
  
In order to use file & record testing for a dataset in the BRE   
        you must provide the EME dataset object (= record format)   
AND   
        you must also provide a file path to a file in the filesystem that contains the actual   
                data that you want to test   
  
You provide the file path in the Deployments of the ruleset.   
Deployments properties configure how the ruleset is tested & how transforms are generated from the ruleset.   
  
==================   
  
How do you create EME dataset objects for intermediate data in the middle of a graph?   
  
1- Have a REFORMAT component that will run the xfr created by BRE ruleset.   
2- Put the REFORMAT component in a subgraph   
3- In the Parameters of the subgraph Properties, I create a new parameter named eme\_dataset\_mapping with the Type “Dataset Mapping”.   
4- When I close & reopen the subgraph, the subgraph now has a data tab that enables me to create EME Dataset Objects for each port in the subgraph (e.g., in port, out port)   
  
Once you’ve done that, you’ll also need to create test data for the inputs & lookups in the ruleset   
— that is sometimes done manually so that you can be sure that you have at least one record to test each case in the ruleset logic   
or   
— can be done by running the graph up to the point of the ruleset & then filtering/manually editing the file so that it’s not too big but does contain all the necessary records to test the data   
  
==================   
  
What do you need to do to set up a LOOKUP FILE in a graph?   
— label for the component   
        —> given when we configure the lookup   
— filesystem path to the data file (Data Location URL)   
        —> given in the Deployments   
— record format   
        —> comes from the EME Dataset Object that we choose to configure the lookup file   
— key   
        —> given when we configure the lookup   
  
If we want to use record testing with the lookup, we should provide a Default Key —> expression or field from the ruleset that matches the key from the lookup   
that enables us to see the matching lookup record for the current test record   
  
  
===================   
  
We want to implement the following logic in our ruleset   
If the customer is between the ages of 30 and 50 AND   
  there is a store in the state that the customer lives in AND   
  the email address is not blank   
then set regional\_message = “Here is your weekly email discount”   
else set regional\_message = “”   
  
What are the triggers?   
        email\_address — what resource contains the email\_address? Input (loyalty)   
                    remember that there is a BRE keyword is\_blank that refers to the trigger column   
                        not is\_blank(email\_address) is wrong!   
                        instead, your comparison cell should just be   
                                        not is\_blank   
        age — what resource contains the age? none of the resources contain age   
                        we have the dob (from the Input) but we do not have the age   
                        we want the age in years; today() - dob is the age in days   
                        one solution: trigger expression column to compute the age as a trigger   
        state — what resource contains the state? Both lookups and the input   
                        so which resource should we use? Stores.state   
                                because we have already linked in.state to Stores.state via default key   
                        so what value should we compare Stores.state against? if the state that customer   
                                lives in is not found in the Stores lookup, what value does the lookup   
                                return? null   
                        we want to compare the trigger column Stores.state to   
                                        not is\_null   
                        because we want to have a message ONLY if there is a Store with a state that   
                                matches the customer’s state   
What is the output? regional\_message   
How many cases do we need? 2 (1 + default)   
  
Implement the logic above for the regional\_message output. Also create a rule for name\_with\_salutation, and create an expression (using the lookup) to assign the region output variable.   
Then run a file test and make sure that both your cases fire for your new rule.   
  
  
Your rule for regional message should currently look like this:   
                triggers                                                                                        output   
                email\_address        Stores.state        Age in Years                regional\_message   
case 1        not is\_blank        not is\_null                > 30 and < 50                “Here is your weekly discount.”   
case \*        (any)                        (any)                        (any)                                “”   
  
Now we want to customize the message for each region.   
Add an additional trigger & cases to the existing rule so that you output the following messages….   
  
If the customer meets all the above requirements AND   
        is from region = “SOWE” then regional\_message = “It’s summer in the Southwest!”   
  
Else if the customer meets all the above requirements AND   
        is from region = “NEWE” then regional\_message = “Let’s go to the beach!”   
  
Else if the customer meets all the above requirements AND   
        is from region = “MIDW” then regional\_message = “Growing some corn!”   
  
Else if the customer meets all the above requirements AND   
        is from region = “WEST” then regional\_message = “Sunny days in the west!”   
  
Else if the customer meets all the above requirements   
        then the regional\_message = “Here is your weekly discount.”   
  
Else regional\_message = “”   
  
  
Make sure to run a file test & check that all your cases fire.   
(Hint: take a close look at the format of the values for the region value and make sure that the values you are comparing match that exactly. You don’t need to do a quick profile — just look at the values in the record testing.)   
  
What is the issue that is causing the cases to not fire?   
values are not “SOWE” or “NEWE”   
values that you need to compare to are “SOWE      “ and “NEWE      “ and so on.   
  
Methods to fix this:   
1- Quick & dirty — change the values that you’re comparing in the comparison cells in your cases. So rather than using “SOWE” using “SOWE      “   
2- Use an expression trigger column instead — enables you to apply string\_trim to the region values before the comparison.   
3- Use named constants.   
  
  
I chose the quick & dirty method (for now). I ran my file test & the results were good. So now I want to do two things:   
1- Save the file test results into my baseline file.   
2- Save the ruleset.   
  
But when I try to save the file test results into the baseline file, it says that saving to the baseline is disabled because there is a record that has been rejected!   
  
In the file test results, records that have been rejected are highlighted in red.   
Remember that the file test results are links — so you can click any value in that record to go back to the ruleset, setting that record as the current test record.   
You will then be able to see the error message indicating why that record was rejected.   
  
  
Our issue is that record #376 has state = “PR”   
but “PR” is not found in the State Region lookup   
so the value State Region.region is null (lookup returns NULL)   
and the problem is that the out.region output variable doesn’t allow null values.   
  
So any ideas on what we could do to fix this?   
1- Edit the record format for the marketing record format to allow null values or give it a default value for the output field.   
Problem: this will be time consuming   
        we have to open the record format in the GDE   
        edit the record format (making sure that this will not have bad effects on any of   
                our graphs that use this record format)   
        check in the record format   
        update the dataset in the BRE resources   
  
2- What could we do directly in BRE?   
        we could create a rule but that seems like a lot of work   
                                trigger                                output   
                                State.region                out.region   
case 1                        not is\_null                        State.region   
case \*                        (any)                                “”   
  
OR   
        we could use the first\_defined() function (much simpler!)           
        first\_defined (State Region.region, “UNKNOWN”)   
          
  
=======================   
  
Tonight —   
        —> finish your ruleset up to the point that I showed in class today   
                that’s the only homework   
        —> do make sure to \*save\* your ruleset before you leave tonight even if you’re not   
                finished   
  
I showed the quick & dirty method of just comparing to “SOWE    “   
but you could also use an expression trigger column with string\_trim to get the same results.   
  
I’ll stick around until 8:30 PM (11 AM) in case you want to continue working. I’ll be happy to take a look at your ruleset.

**Exercise "Marketing\_Selection"**   
We did all these steps in class today, but if you were unable to finish -- please use these instructions to complete the exercise.   
  
1. Switch to the standalone BRE view.   
  
2. Create a new ruleset in the BDS\_grocery project named "Marketing\_Selection"   
  
[Notes:         EME dataset paths are          /Projects/BDS/BDS\_grocery/data/...]   
3. Add loyalty.dat as the input dataset.   
4. Use marketing.dat for the output dataset   
5. Add stateregion.dat  and stores.dat as lookups (be sure to provide both the Lookup Name and Component Label.   
For both lookups, "state" is the key. Make sure to set the Default key (in.state) for both lookups as well.   
  
6. Edit the default deployment with the appropriate values for the data locations:   
Test data locations for default deployment (advice: type the path once & then copy+ paste)   
/home/ggx/jun8/<yourid>/users/<yourid>/BDS/BDS\_grocery/data/loyalty.dat   
/home/ggx/jun8/<yourid>/users/<yourid>/BDS/BDS\_grocery/data/marketing\_baseline.dat (this file doesn’t exist yet it will be created later)   
/home/ggx/jun8/<yourid>/users/<yourid>/BDS/BDS\_grocery/data/stateregion.dat   
/home/ggx/jun8/<yourid>/users/<yourid>/BDS/BDS\_grocery/data/stores.dat   
  
        example: /home/ggx/jun8/ggx\_jdoe/users/ggx\_jdoe/BDS/BDS\_grocery/data/loyalty.dat   
  
  
7. For output mappings:   
Start by providing simple values like "" or 0 for all output fields. Run a file test. Does it succeed?   
Then add more complicated expressions & rules, one at a time, testing as you go. (You don't have to run a file test every time, but you should check the Computed Values column to make sure that the outputs are correct.)   
  
a. Use default values like "" or 0 for the birthday\_message and offered discount.   
b. The email\_address field should be auto-mapped -- you should not need to edit it. (Turn on AutoMap in   
        ruleset properties panel at bottom of screen)   
c. Use the "region" value from the Store Regions lookup to set the value for the region field.   
        first\_defined(State Region.region, "UNKN")   
d. Create a rule to populate the values for name\_with\_salutation that adds "Mr." "Mrs." or "Miss" or nothing to the first\_name + last\_name, depending on marital\_status and gender.   
e. Create a rule to populate the values for the regional\_message using this rule logic:   
        First, create a new trigger column. Right-click and edit the Trigger Properties. Set the Value Source to "Expression", name the   
        column Age, and set the expression to        ( today() - dob ) / 365   
        Then create this rule logic:   
        triggers                                                        output   
        Age                        Stores.state        email\_address                regional\_message   
case1         > 30 and < 50                not is\_null        not is\_blank                "here is your weekly discount"   
case\*        any                        any                any                        ""   
  
8.   
Edit the rule that populates the regional\_message output. Keeping the existing logic, but add an additional trigger and cases so that the regional\_message values are customized based on where the customer lives....   
        Stores Region.region                Message   
        "SOWE        "                        "Summer in the southwest!"   
        "MIDW"                        "Growing some corn!"   
        "NEWE"                        "Let's go to the beach!"   
        "WEST"                        "Sunny days in the west!"   
        any                                "Here is your 10% discount."   
(\* default -- doesn't match any conditions)         ""           
  
The logic from step #7 (that this applies only to customers between the ages of 30 & 50, who live in a state that also has a store, and have a non-blank email address) still applies.   
You should simply add an additional trigger column & cases as necessary to output these messages.   
  
Note: Make sure to run a file test after you've edited the rule. Do you see any of the new messages? If not -- go back and double-check your input test data values. (Hint: the input test data values have spaces in them.) Remember to check the "Times Fired" column in your rule (after you've run a file test) to make sure that all of your cases have fired.

**DAY 5:**

In a ruleset you can copy & paste the comparison cells using Ctrl-C and Ctrl-V on the selected cells.   
You can copy from one ruleset and paste into another.   
OR you can copy from a ruleset, paste into a spreadsheet or text file, and then paste into the new ruleset.   
  
Ways to use this:   
— copy from one rule and paste into another   
— if someone has already written the logic in a spreadsheet (which is actually quite common), you can copy from the spreadsheet and paste into a new rule   
  
For text files, it also works — the entries in the text file should be tab-delimited.   
  
You can edit any deployment to change which baseline it uses.   
If you created a baseline file with Deployment named Winter called marketing\_winter.dat   
you can edit the Deployment named Fall to use that baseline file if you want —> but you do have to edit the path to use that file.   
The baseline files for deployments in the same ruleset are interchangeable — they can be used with any deployment (because they all use the same record format — output record format) just by editing that deployment to use that file.   
You cannot tell a file test to compare to a specific baseline file directly — it automatically uses whatever baseline file you’ve specified in the deployment.   
So —  without editing — you cannot compare the test data output from Winter deployment to the baseline that you saved for the Fall deployment. The only way to do that would be to edit the Winter deployment to use marketing\_fall.dat and then run a file test —> that would then compare the test data output from Winter deployment to the saved output from Fall deployment.   
  
============================   
  
If you have declared a parameter in a graph or sandbox that you want to be able to use in a ruleset (in the xfr generated by the ruleset)   
—> in the ruleset, you create a parameter with that exact technical name   
then that parameter can be used throughout the ruleset: as a trigger, in comparison cells, to compute output expressions for cases, in expressions that compute output fields   
very similar to how you could use a named constant —> except that the parameter is declared in the graph so when you generate an xfr from the ruleset —> parameter will be embedded in the xfr and at runtime, the value of the parameter comes from the pset/graph/sandbox environment that the graph runs in.   
  
  
==============================   
  
What are some reasons why you might want to turn on logging for a deployment (so that the generated xfr contains write\_to\_log commands to write information to the component’s log when running the graph)?   
— when you run the graph with full data, you are getting rejections or incorrect values that you don’t see in the small test data in the ruleset   
        debugging!   
— you have a requirement to audit which cases in each rule were used to compute the output   
— maybe others?   
  
=================================   
  
Think about a ROLLUP component in a graph. What parameters do you need to configure to use ROLLUP?   
key   
is input sorted or should the rollup run in-memory?   
DML record format (in, out ports)   
transform   
  
If we want to use BRE to create a ruleset that generates a ROLLUP transform   
        —> DML record format (in, out ports)   
        —> we need to know the key   
        —> we need to know whether the input is sorted or the rollup runs in -memory   
                        why? (the internal variables that the rollup transform uses are different   
                                        sorted input —> global variables for the internal computations/aggregations   
                                        in-memory —> uses temporary\_type for internal computations   
  
  
Big differences between ROLLUP ruleset versus REFORMAT ruleset:   
— in ROLLUP ruleset, use Tools > Rollup Keys to set the key fields + sorted property   
— in ROLLUP ruleset, there is no record testing because it doesn’t make sense since you have multiple input records —> 1 output record   
— in ROLLUP ruleset, to use typical aggregation functions that you use in ROLLUP, we have a special category in Keywords & Functions — Rollup functions   
  
In ROLLUP functions, can you do a sum of only the values that meet a particular condition?   
In GDE transforms, you can.   
All the aggregation functions in GDE have a optional conditional argument   
        sum (in.trans\_amts, in.trans\_date > “2000-01-01”)   
  
You can use the same syntax in BRE Rollup functions   
        rollup\_sum (in.trans\_amts, in.trans\_date > “2000-01-01”)   
  
============================================   
  
The purpose of a VALIDATION ruleset is to create a set of validation rules/expressions to validate the input data.   
Result of a VALIDATION ruleset is a specialized record that contains a list of all the reasons why a particular input record is invalid and optionally, error codes & substitute values.   
  
In a VALIDATION ruleset, you set the Input record format just like you would any other ruleset.   
But the output record format is pre-defined for you. When you create a new VALIDATION ruleset, the output record format is already there for you.   
You can still (optionally) define lookups, parameters, user-defined functions, internal variables —> in order to support your validation testing.   
  
The next big difference is that there are two views of the ruleset   
        Sheet —> like a spreadsheet that contains all the keyword tests that you want to   
                                run against the data   
        Rules —> same view that you get for Rules in other types of rulesets   
                                list of rules   
  
The next big difference is that there are a bunch of specialized keywords (Validations) for testing common properties of the data values   
        like max, size, min, etc.   
  
  
The generated transform from a VALIDATION ruleset (by default) is designed to be used in REFORMAT.   
But! You can generate a user-defined function from the ruleset, which can be called from any component. (You do this on the Info page in the Deployments — “Generate DML wrapper function”)   
  
============================================   
  
Everything you do in BRE could be done by writing a transform in the GDE.   
The purpose of BRE/ACE is to add \*productivity\* to your project.   
  
Writing complicated logic in a transform in the GDE has some drawbacks:   
— it’s difficult to test individual rules for output fields   
— it is difficult to debug complicated conditional logic   
— it is difficult to read which makes it more difficult to maintain   
— you have two choices: you can either have someone write the transform in the GDE on their own (and hope that it works) or it becomes the job of the graph developer to write both the graph & the transform   
  
When you have complicated, domain-specific logic — BRE makes sense   
— enables people who don’t know or don’t have access to the GDE to contribute to the project   
— integrated testing built in to test individual rules/ expressions   
                record testing   
                file testing + baseline   
                ruleset lineage   
                view data / quick profile   
— ruleset can be developed/tested/edited/maintained separately from the graph (& you can have multiple people editing/testing individual rules)   
  
  
The goal with ACE is that you have a reusable graph — transform might be pretty simple but it needs to be specified for each pset.   
You can have someone in the GDE writing these transforms but each person who is writing the transforms needs access to the GDE.   
With ACE, you can simplify the process by creating a specific UI (template) for your specific graph so that the user is guided toward choosing (and testing) valid values for each parameter. Don’t need any access to the GDE in order to do configurations in ACE.   
  
  
==========================   
  
When you write rulesets, you are looking for simplicity.   
        Simple things are easier to write, test, and debug.   
  
You can create a rule with 100+ triggers and 100+ cases but it will be difficult to debug & test & maintain (edit).   
You can create expressions that are if-else conditions (instead of writing a rule) but it will be difficult to debug & test & maintain.   
Doing these things will take away from productivity rather than adding to it.   
  
Remember how when we created internal variables — we could see the record test values for those internal variables —   
        visible in the rule grid   
        visible in the ruleset lineage   
        visible in the resources   
We can check what the value is & easily debug what’s going wrong in a rule.   
  
That’s a much better solution than writing a complicated expression.   
  
  
What if you do have a very complicated bit of logic to compute a particular output?   
Try to break it down   
        are there cases that are similar to each other? could you create two or three or four sets   
        of cases that use a common set of triggers?   
  
        if you can do that, then you can split up the logic into \*multiple\* rules   
        each rule contains only the triggers that set of cases use   
        each rule contains only the cases that are relevant to those triggers   
  
how do you save the intermediate results from each of these individual rules & then put them together to produce the final output value?   
        if you’re using single-fire rules —> use internal variables   
                        you can compute an internal variable with a rule!   
  
        if you’re using multi fire rules —> it’s even easier — just write multiple rules   
                each rule will add to the output of the previous rules!   
  
Validate-and-Quote ruleset has examples of both of these techniques   
          
        The 3 rules that compute Validation Messages are an example of splitting logic into   
                smaller subsets of triggers/cases based on similar attributes for multi fire rules that   
                compute List output:   
                Validate Person        1 trigger x 5 cases   
                Validate Vehicle  4 triggers x 10 cases   
                Other Validation Messages  3 triggers x 3 cases   
        If we had done in one big rule, it would have been 8 triggers + 18 cases — much more   
                        difficult to read & understand   
  
  
        The multiple rules that are used to compute intermediate values for the Total Premium   
                write their outputs to either other output fields or to internal variables   
        so that each intermediate is simple + easy to test on its own   
        then in the end in the Compute Total rule, we just add up the values of all the   
                internals  to compute the final output value   
  
Highly encourage you to explore the Validate-and-Quote ruleset   
        it is designed based on our recommendations for what is a good ruleset   
        shows you in practice what a good ruleset should look like.   
  
===========================   
  
Does anyone have any general questions about rulesets?   
  
I want to encourage you not to worry too much about the performance of the ruleset.   
Remember that we do optimizations on the logic when you generate the transform.   
You should focus on writing transparent, easy to understand rules.   
If you then find that the performance of the generated transform is not what you need — contact support — they can either suggest modifications or add this to our knowledge so that we can write optimization code (for BRE) for this situation.   
  
=============================   
  
Your accounts on the training server are active until August 31 2015 — you can log in at any time until then. I’ll keep the bre\_training web application running for as long as I can (at least a few weeks)   
so you can log in & explore the other rulesets (please check out Validate-and-Quote)   
try out experiments   
  
If you have any questions on general topics about BRE or Ab Initio in general,   
        rbuchheit@abinitio.com   
Happy to answer questions!   
  
If you have specific questions about a customer project,   
        support@abinitio.com   
Just let them know which customer & which project.   
If it’s a configuration or a ruleset, there is package for support!   
        Help > Support > Create Package for Support   
The package that it creates is a tar file that you can send it with your question.   
  
=======================   
  
The Express>It web application (BRE + ACE) is a .war file (web archive file) or an .ear file.   
When you build the .war / .ear file, you specify a “name” for the instance   
(ours is “bre\_training”)   
That shows up in the URL, the URL is typically something like:   
        [http://machine:1234/AIAppConf](https://owanj.galaxysi.com/owa/redir.aspx?SURL=0tqo21v-wz19AaLnCXDRl_40-xSyv2IpflXPgH9_1Mt9hChmIk7TCGgAdAB0AHAAOgAvAC8AbQBhAGMAaABpAG4AZQA6ADEAMgAzADQALwBBAEkAQQBwAHAAQwBvAG4AZgA.&URL=http%3a%2f%2fmachine%3a1234%2fAIAppConf)   
in that case “AIAppConf” is the instance name.   
  
Then in your abinitiorc file, you add entries to link an EME TR to that web instance.   
        so we have this:   
        AB\_AIR\_ROOT @bre\_training : /home/ggx/jun8/eme/tr   
  
When you log into Express>It, it will show the projects & files in that EME TR.   
  
As for linking to a particular project —> remember how you created a new standalone ruleset:   
        — choose the Type: Reformat, Rollup, Validation   
        —> choose which Project it is saved in   
        — give it a name   
  
When you create a new template in ACE —> you choose which graph(s)/plan(s) you want to associate with that.   
Any new configurations for that template are automatically in the same project.   
  
If you’re interested in learning more about how to build templates   
        in the Help Library, there is a whole section on this.   
        Express>It help                “Template Developer’s Guide”   
  
  
In V3.1, we combined ACE + BRE into a product called Express>It because so many customers were using both of them. By having that as one product, it makes licensing & setup simpler.   
So you have Express>It templates, Express>It rulesets.

**BRE:**

**DAY :1**

Monday through Friday, 4:30 — 8:30 PM IST   
We’ll take a 20 minute break about halfway through class each day — around 6:30 PM IST.   
  
  
V3.1 and earlier, we talked about two separate products that were both accessed from the same web application:   
        BRE — Business Rules Environment   
        ACE — Application Configuration Environment   
  
in V3.2 and later, we combined these two interfaces into a single product (still accessed from one web application) named:   
        Express>It   
        (Express>It = ACE + BRE)   
                has two interfaces:   
                        Rulesets Page (BRE — interface for writing standalone rulesets)   
                        Manager Page (ACE — interface for designing templates and editing configurations)   
  
  
Today’s class is learning how to design templates (earlier == ACE).   
We’ll start by configuring an existing template. We’re going to critique the ease-of-use of that template and then edit the template to improve its design. (=edit/improve existing template). We’re learn what is a “good” design for templates.   
After we do that, we’ll learn how to design a new template from the beginning.   
Along the way, you’ll learn how to use the most common UI controls, how to create variables in a template, how to work with metadata in a template.   
  
=============================   
  
The result of configuring a template in Express>it —> pset   
You can create and modify and edit and test psets directly from the GDE.   
Express>It does not add any functionality directly to your application.   
Benefits are indirect —> improved productivity   
        —> team who edits the configurations does not need to learn/understand/have a license for   
                        the GDE; they can work entirely from a web browser   
        —> configurations can be edited and tested (you can run the psets!) from the web browser   
        —> templates can be designed to constrain the options for what values are set in the   
                        configuration        (example: maybe one of the inputs must be a file from a particular   
                        directory — constraining that in the pset editor in the GDE is next to impossible,   
                        constraining that in a template in Express>it is very easy)   
                        the template developer can assist the configuration editors by designing a   
                                template that constrains them into making good choices   
  
  
There is a cost (in development learning + time) to writing templates. They are not trivial.   
Cost of designing and developing the templates needs to be outweighed by the benefits of including SMEs in the development & testing process.   
  
Write once (one generic graph + one template)   
        Configure many many many times (many configurations = many psets)   
                —> use Express>It   
  
  
.apptempl —> file extension that is used for application templates   
                                        XML document   
                                        describes a UI that is associated with a graph, set of graphs or a plan   
                                        UI controls are associated with input parameters in the graph/graphs/plan   
  
.appconf —> file extension that is used for application configurations   
                                        XML document   
                                        describes the values that were set (saved) in the configuration for a particular   
                                                template (each configuration is associated with exactly 1 template)   
  
Both of these are created & edited through Express>It   
Files are saved in your sandbox, and can be checked in the EME TR project   
  
=================   
  
In Express>It   
        when you save a configuration   
                —> result:  .appconf file and a .pset file are saved in your sandbox   
                What sandbox?   
                When you log into Express>It, it creates a “ACE-managed sandbox” in a location   
                specified by configuration variables for Express>It   
                You have your own sandbox — sandboxes are arranged by user id — for each project   
                of each configuration/template that you edit.   
                (More on this later at the end of week — when we talk about architecture)   
  
        when you save & publish a configuration   
        —> result: .appconf file and a .pset file are saved in your sandbox AND   
                                checked into the EME TR project   
                How does it know which EME TR to use?   
                The EME to use with Express>It is configured using configuration variables   
                (more on that when we discuss architecture)   
  
        when you abandon your changes   
        —> result: any edits that you made through Express>It are lost   
                                files in your ACE-managed sandbox are reverted back to the latest version   
                                from the EME TR.   
  
Same thing is true for saving & publishing templates   
        .apptempl file is saved in the ACE managed sandbox   
        if you publish, it is checked in to the EME TR.   
  
======================   
  
For our class, your user id is used twice in the path to the ACE managed sandbox   
 — why?   
  
/home/ggx/aug31/<yourid>/users/<yourid>/ace-training   
  
APPCONF\_ROOT directory                /home/ggx/aug31/   
each one of us in working on a different branch in the EME TR so that we don’t overwrite each other’s work when we check files that have the same name   
                                                                                <yourid>   
each person who logs into a particular branch from Express>It gets their own ACE-managed directory for sandboxes   
                                                                                /users/<yourid>   
then the name of the project that the configuration/templates are saved in = name of the sandbox in the file system   
                                                                                ace-training   
  
  
For example, if I were to log into Rajesh’s branch with my user id, my ACE managed sandbox would be:   
        /home/ggx/aug31/ggx\_rmasanamjay/users/win\_rbuchheit/ace-training   
  
When Rajesh logs into his own branch with his own user id, his ACE managed sandbox is:   
        /home/ggx/aug31/ggx\_rmasanamjay/users/ggx\_rmasanamjay/ace-training   
  
When you log into a particular branch, Express>It automatically creates the subdirectories and project sandboxes for you.   
  
  
Instead of using ACE managed sandbox, can I direct files to my own sandbox?   
        Yes and no.   
        From Express>It — no. When you click “Save” button, it automatically saves in the   
                ACE managed sandbox. That cannot be changed.   
  
        However, if you save & publish from Express>It, then the files are checked in to the EME TR   
                and you can then check them out into any sandbox location that you want.   
  
=========================================   
  
What are some of the poorly designed and annoying things about the Exercise 1 template?   
What bothered you when you were editing the configuration?   
  
— no View Data for the source (no way to check that the file contains the data that you want)   
— no validation against the DML record format that you chose — was that record format a valid format for the source data   
— Fields for the filter by expression weren’t visible — the configuration editor needed to figure out the column names / field names themselves without any feedback about whether they were correct or not   
— Remember that we’re expecting SMEs to edit these configurations. Are they going to know that Ab Initio uses == to test equality in filter expressions? Most SMEs know SQL (vendor\_state = “CT” —> would cause a validation error when you try to run the graph).   
— Nothing was labeled very helpfully, there were minimal labels (like “Source file”) but no more detailed descriptions of what you were supposed to do.   
— Had to provide the same record format (transactions-map.dml) twice. Should have only needed to do that once.   
— You had to jump back and forth between the controls — first configure the source at the beginning and then configure the target at the end and then configure the filter expression. The order of the UI controls was not meaningful to someone who doesn’t make graphs (like SME).   
— No feedback on what you’ve already done. I noticed this when you all were working through the exercise. I would check your screens & I had no way of knowing what step you were on because there’s no feedback about when a particular value has been satisfied.   
In contrast — think about a graph in the GDE — the GDE displays yellow to-do cues to let you know what values you still have to fill in. Or what expressions/values are invalid. The configuration has no feedback like that. If you got up and went for a cup of tea in the middle of doing your work, you’d have no idea where you had left off.   
  
  
One of the most common UI mistakes that developers make when starting with Express>It is that they design the template to mimic the design of the graph.   
Problem: Whole point of Express>It is to make the configurations easy to edit for people who don’t know the GDE and don’t know how to create graphs. Why would you create a UI that mimics a product that they don’t know to use? Doesn’t make sense.   
Recommendation: think about the order in which the parameters need to be configured (record format \*before\* the filter expression that uses the record format) and design the template to guide users to do things in \*that\* order.   
Template does \*not\* have to be similar to the graph design — because the template is NOT a graph. It’s a UI for configuring \*parameter\* values.   
  
Just because the graph design is:   
        INPUT FILE —> FILTER —> OUTPUT FILE   
That does \*NOT\* mean that your template has to be:   
        Source File —> Filter — Target File   
  
Instead your template could be:   
        Source File and Target File —> Filter   
if that makes more sense for the SMEs and the order in which the parameter values are defined.   
  
  
  
===========================   
  
Two categories of UI elements in a template   
        containers — organizational things, they collect other containers and controls to group   
                        them together in useful ways   
                        examples: Canvas, Box, basic Popups   
        controls — used to ask the user for information or to show information to a user   
                        controls are how the user configures values for the parameters & metadata in the   
                                configuration   
                        controls are going to “ask questions” (path to source file) or   
                                “answer questions” (View Data)   
                        examples: Button, Check Box, Text Box, File Browser   
  
Components organizer in template editor shows \*only\* the components that you can use at a given time. So if you currently have a box selected, then it shows only elements that you can add to the box.   
  
Notice in the template for Exercise 1 that the labels of the popups have “…” at the end.   
This is a UI labeling convention   
        the … indicates that when you click that link it will open up a new window.   
You don’t have to do this — your company might have a different convention for that or none at all — but AI consultants are taught to do it this way.   
  
For Exercise 2A — we copied & pasted UI elements and deleted some containers that we didn’t need. But we didn’t add or delete any controls. We just moved the existing controls to new containers.   
The configuration (cfg1) only contains the values for the controls —> even though we edited the template — because we did not add/delete controls — cfg1 still works & still remembers all the values from previously.   
Example of how the workflow in the template is mostly unrelated to the graph design.   
  
====================   
  
What if you want to do a “Save as” from a template?   
        So that you basically have two copies of a template and then you can edit either copy.   
        Templates don’t have a duplicate button in the Express>It UI        You could still do this though (possibly — I haven’t done this) by using “air cp” to make   
                and rename a copy from the EME TR.   
        Unlikely that you would want to create a duplicate template in Express>It   
                why? typically most graphs only have 1 template associated with them   
                                (extremely rare to have two different templates for the same graph)   
                                each template is associated with a specific graph   
                                  
What if you want to do a “Save as” from a configuration?   
        You want to take an existing configuration and save it with a new name so that you   
                can edit the copy —> “Duplicate”   
        Very common because you might have many configurations for the same template that   
                have just minor differences in the configuration values. Duplicate makes a lot of sense.   
  
==========================

**Express>It web application:**   
Start Internet Explorer from your remote desktop shortcut   
        [http://rtf-ggx:6563/bre\_training?appIdentifier=](https://owanj.galaxysi.com/owa/redir.aspx?SURL=LumkOVBBerxW97PQJrN4sm2yJhhdjn-aVwnmQ4-mKYr1vB_MIk7TCGgAdAB0AHAAOgAvAC8AcgB0AGYALQBnAGcAeAA6ADYANQA2ADMALwBiAHIAZQBfAHQAcgBhAGkAbgBpAG4AZwA_AGEAcABwAEkAZABlAG4AdABpAGYAaQBlAHIAPQA.&URL=http%3a%2f%2frtf-ggx%3a6563%2fbre_training%3fappIdentifier%3d)<yourid>   
  
                The URL is case-sensitive & must be \*typed\* exactly as above.   
                example: [http://rtf-ggx:6563/bre\_training?appIdentifier=ggx\_jdoe](https://owanj.galaxysi.com/owa/redir.aspx?SURL=qwYj8hZcRZQarHTM9wER-gYTu9A_jJR7WOGYP0KcDEX1vB_MIk7TCGgAdAB0AHAAOgAvAC8AcgB0AGYALQBnAGcAeAA6ADYANQA2ADMALwBiAHIAZQBfAHQAcgBhAGkAbgBpAG4AZwA_AGEAcABwAEkAZABlAG4AdABpAGYAaQBlAHIAPQBnAGcAeABfAGoAZABvAGUA&URL=http%3a%2f%2frtf-ggx%3a6563%2fbre_training%3fappIdentifier%3dggx_jdoe)   
  
Log in:   
Username: <same as your remote desktop id>   
Password: <same as your remote desktop password>   
  
If you log in and you don’t see about ~15 or so application templates —   
if the list is blank — that means that you have typed the URL incorrectly.   
In that case, quit IE, restart it and retry the URL.   
  
  
**In the GDE**   
  
A.C.E. Managed Sandbox location:   
        /home/ggx/aug31/<yourid>/users/<yourid>/ace-training   
  
EME location           
        export AB\_AIR\_ROOT=/home/ggx/aug31/eme/repo   
        export AB\_AIR\_BRANCH=<your user id>   
  
=================================

**DAY 2:**

In our Exercise 1 template, there is a user-defined function named   
        file\_label   
  
What it does is — takes two arguments   
        if the value of the first argument is not blank, then the result of the function is the   
                value of the first argument   
        if the value of the first argument is blank, then the result of the function is the value   
                of the second argument   
  
        file\_label (“”, “abc”) —> “abc”   
        file\_label (“xyz”, “123”) —> “xyz”   
  
        if the first argument is a filesystem path  /path/to/some/file.dat   
                and the value is not blank, then it returns only the last part of the path (filename)   
        file\_label (“/path/to/some/file.dat”, “alternative text”) —> “file.dat”   
  
User-defined functions can be included in a template in the “declarations” on the “DML globals” tab.   
  
=======================================   
  
Parameters, variables, etc. in a template typically have properties that have a special characteristic:   
        default write property   
        default read property   
  
For example, for a parameter   
        the “expression” property is the default \*write\* property   
        the “resolved\_value” property is the default \*read\* property   
  
What is this read/write stuff?   
In a UI, anything that can hold a value (like parameters or variables) basically has two functions   
        you either need to write a value into that variable/parameter   
        or you need to read a value from that variable/parameter   
  
        when a person edits a configuration and I select a file path in a FileBrowser control,   
        the FileBrowsers writes the value for that path to a parameter or variable. Value   
        is written into the “default write property” (= set the value)   
  
        when a person saves a configuration and Express>It creates a pset, it reads the   
        current value of the parameter from the configuration and uses that in the pset   
        it reads the value of the parameter from the “default read property”   
  
  
Two categories of UI elements in a  template:   
        containers — organization   
        controls — interactive   
                        ask user for input (person who edits the configuration makes a choice —   
                        select a file, check a check box, select an option from combo box, create   
                        mapping rules) OR displaying feedback to the user (ViewData)   
  
        When a user edits a configuration, they are selecting values from the controls.   
        We need a way to \*store\* the value that the user has selected —> SourceTargetValue   
                        (or TargetValue) property of the control.   
        Then when we generate a pset from the configuration, we read the value that was   
                stored in the parameter resolved\_value property —> save that value to the pset   
        How do we link the SourceTargetValue property of a UI control to the value stored   
                in a parameter? “Bind” the metadata object (parameter) to the SourceTargetValue   
                property of the UI control.   
  
  
==============================   
  
Exercise 2B   
        we have not added, modified or deleted any controls yet   
        —> our changes from today are still reflected \*correctly\* in the configuration (cfg1)   
                that we created yesterday   
  
        if that’s the case, why did the exercise tell you to create a new configuration (cfg2)?   
        to test that the file\_label function worked properly for the Box labels when the file   
                paths weren’t set yet   
        in cfg1 —> the file paths for the source & target were already set and so you   
                wouldn’t ever see the alternate text   
        by creating a new configuration —> we were able to see the alternate text   
                “File not selected” and THEN see it change to the file path once we configured   
                the path to the source file.   
  
        Notice the workflow that I showed during my demonstration   
                I made a change in the template   
                then I switched (“Edit configuration” button) to the configuration to see the effect   
                        of that change and test that it worked   
                sometimes you can use an existing configuration to do that, other times you   
                        need to create a new configuration (like we did to test behavior of file\_label)   
  
====================================================   
  
In the ExpressionEditor list of properties   
        some properties are grayed out (like ExplicitCommit)   
        and other properties are bold   
  
        grayed out — has a default value and you don’t need to set it unless you   
                        want to change the value from the default   
        bold — doesn’t have a default value AND is required to set its value   
  
        some properties are \*required\* — you must set a value in order to be able to use   
                the control   
  
  
If we want to use ViewData control — what does View Data need to show you the data?   
        required:   
                data (usually from a URL, path) —> SourceValue property   
                record format (DML) —> RecordFormat property   
  
        optionally:   
                filter expression —> FilterExpression property   
  
  
Issue:   
        I edited the filter expression in the ExpressionEditor control.   
        Default behavior is to save the resulting expression to the SourceTargetValue property (proj.first.FILTER\_EXPRESSION) when I \*CLOSE\* the editor (in this case, the popup).   
  
        Problem: I have a View Data control on the same popup & I want it to display the   
        updated filter expression results when I edit the filter expression   
  
        Solution: Set “ExplicitCommit” property of ExpressionEditor to true.   
        Adds a button (Apply Changes) to the ExpressionEditor control so that the   
                person who is editing the configuration can choose to save the expression value   
                before they close the popup.   
                When the value gets saved, the ViewData will update.   
  
  
Issue:   
        When I change the filter expression and apply the changes, the view data updates   
        (good) but it doesn’t filter the data the way you would expect.   
                instead:           
                records that match the expression are shown in white   
                records that don’t match the expression are shown in red   
  
        Default “FilterMode” behavior.   
        What we want is to show only the records that match the expression   
        Solution: change the value of the FilterMode property of ViewData   
  
  
  
Summary:   
        ExpressionEditor control   
                RecordFormat        ——>         proj.first.SOURCE\_DML   
                SourceTargetValue —>        proj.first.FILTER\_EXPRESSION   
                ExplicitCommit —>                 true   
  
        ViewData control   
                FilterExpression        —>        proj.first.FILTER\_EXPRESSION   
                FilterMode        ——>                MatchingOnly   
                RecordFormat        —>        proj.first.SOURCE\_DML   
                SourceValue        —>                proj.first.SOURCE\_URL   
  
Why does the ViewData control have a “SourceValue” property but \*not\* a “SourceTargetValue” property?   
        Because it is read-only.   
        It displays data to the configuration user.   
        But there is no input for this control — the user does \*not\* set a value for this control   
                so there is no Target value to store.   
  
ExpressionEditor does have SourceTargetValue because there is a Target value to store — the expression that the configuration user enters into the control.   
  
============================

**DAY 4:**

Our idea is that we want to end up with a Filter expression that looks like this:   
        state\_abbrev member [vector “CT”, “MA”]   
  
We’re adding two controls so that the user in the configuration can choose:   
        1- what is the field name of the field that contains the state abbreviation values   
                (requirement: field name could be different in different inputs —> “generic graph”)   
        —> field\_name   
        2- what states does the user want to filter on   
        —> state\_list   
  
Two controls = two variables BUT we want to combine those two variables into an expression that we’ll use for the Filter by Expression component (references the $FILTER\_EXPR parameter) in the graph.   
To compute the value of FILTER\_EXPR based on the values on those two variables, we need an automatic assignment.   
  
        our variables are going to have values like this:   
                field\_name                <string>                state\_abbrev   
                state\_list                        <vector>                “CT”, “MA”   
  
        our FILTER\_EXPR must be a \*string\* value (Filter expressions are strings!) that looks like this:   
                state\_abbrev member [vector “CT”, “MA”]   
  
        additional requirement: if the user chooses \*not\* to use a subset (no filter expression), we want   
        the FILTER\_EXPR to be this:   
                “1”   
        Why? What does “1” mean as a filter expression? Always true = no filter on the data.   
  
        Our automatic assignment expression looked like this:   
                if (vars.use\_subset and not is\_blank(vars.field\_name))   
                        vars.field\_name + “ member “ + string\_representation(vars.state\_list)   
             else   
                        “1”   
  
        The string\_representation function takes a vector value (like the value returned as the Target   
                values from MultiChoiceList), converts it to a string representation of that value that is   
                written in a way that it can be used in expressions… like this:   
                [vector “CT”, “MA”]   
  
====================================   
  
FieldPicker has a RecordFormat property   
        —> expects that the value of this property is a string containing a record format   
                (NOT a path)   
  
        —> we have an variable, input\_file, that has the record format for the input file as both   
                a path property and as a string (record\_format) property   
                  
When we bind the metadata for the FieldPicker RecordFormat property   
        —> we need to make sure that we choose the correct property (record\_format) from input\_file   
variable   
  
================================   
  
For MultiChoiceList control, you must give it an initial list of Choices to display.   
There are a number of different ways to do so….   
        1- List Values   
                You click the “Add” button to add each item to the list   
                If we have 7 state abbrev. to choose from, then you need to click Add 7 times.   
  
        2- Expression   
                You write an expression either hard coded or using built-in functions (e.g., string\_split)   
                that returns a vector value.   
                [vector “MA”, “NY”, “CT”, “RI”]   
  
        3- Bind to metadata   
                if you have another variable or parameter in the template that contains a vector list   
                then you can use that as your list of choices.   
  
Someone is saying — could we get the list of state values from the data — from the state\_abbrev field?   
Think about a graph — how are you going to do that? How are you going to write a filter expression that uses your input data as part of the expression? It’s impossible — the filter expression needs to be computed \*before\* the graph starts processing the input data.   
Also we have 1.5 million input records — many of them have the same state. So you’re going to list 1.5 million states in your template? Using the input data doesn’t make sense.   
  
Using the input data to provide the list of possible states is not possible, not a good idea even if you could do it.   
Using a different dataset — like a lookup file or a simple text file that contains a list of states — is a reasonable idea for importing a list of values for the Choices for MultiChoiceList.   
        Two possible methods:   
        1- You can add a lookup to the template. This is \*not\* a lookup that shows up in the graph.   
                It is a lookup that simply provides source data for the template itself. Pretty advanced — but   
                it is the recommended & best way to do this.   
        OR   
        2- You can use metaprogramming functions like read\_file() to read the contents of a file on disk   
                to supply the data values. The drawback to this is that if the file changes, the configuration   
                doesn’t update to reflect those changes until you close and re-open the configuration.   
                The other drawback —> you have to figure out a way that this file is going to be present   
                in all your environments where the configuration might be edited. You’d want the file to   
                be located in a sandbox and checked into the EME TR.   
  
==================================================   
  
What if the user chooses an invalid field in the configuration? A field that doesn’t contain state values?   
Right now (as of Exercise 3B), they wouldn’t find out until they ran the graph.   
        —> Not good.   
  
We would like to show them what the result of choosing the filter expression is on some sample (small!!!) data so that they can verify that   
        — they’ve chosen the correct source file   
        — they’ve got a valid filter expression on the data   
  
===================================================   
  
Here’s what we want to do —   
        we want to provide (small) sample data to use as feedback in the configuration   
        we want to show the configuration user what the data will look like after it has been filtered   
  
Two methods:   
First —   
        1. Create a SampleData object —> reference to the small sample data file path   
        2. Create a FlowData object —> linked to a flow in the graph   
        3. Create a DataLink object —> enables you to substitute the SampleData as the input to the   
                FlowData (instead of the actual data — very large!! — data that the graph uses)   
                —> sample data is used only in the template, the graph pset is not affected by this usage   
OR   
Second —   
        1. Create a FlowData object —> linked to a flow in the graph   
                and then you set the sample\_data\_rpath property to point to a file to use as small sample   
                data for the input to the FlowData object   
  
It seems like the Second method is much simpler and easier.   
Why would we ever use the First method?   
        —> same SampleData object can be reused throughout the template   
        —> DataLink object can have things other than files as input   
                it can have other flows (like from a db table or from another flow) as input   
  
Second method is good to use when you are setting up just a single FlowData object and you have your sample data saved in a file.   
  
First method is a more general method — the input for step #1 doesn’t have to be a file. It could be another flow (like from a table or some other input).   
  
  
=================   
  
We need to be able to reference data directories in the filesystem.   
How do you typically set the path (data location URL) for a Input File in a graph? What does a typical path look like?   
        $AI\_SERIAL/filename.dat   
  
  
So in a template, we want to be able to reference the same parameters for file locations — because presumably our sample data will either be located in the data mount (somewhere like $AI\_SERIAL) or perhaps it will be saved in the sandbox (somewhere like $AI\_REFERENCE  — “reference” is a term that we use often for small, sample data).   
We need access to the sandbox parameters from the template. There are literally 100s of sandbox parameters — including them all in the template by default would be overwhelming and also \*slow\*.   
  
Solution:   
You create PDL expressions to reference whatever sandbox parameters you actually need to use in the template. (old way)   
You right click on the graph name in the Metadata list and choose Add Project Parameter (new way)

**DAY 5:**

On the ViewData control, the default FilterMode is to show all records   
        & highlight the ones that will be filtered out in red   
  
This is different than what ViewData does in a graph. In a graph, it shows only the matching records.   
  
Why is it different? Express>It is \*not\* designed for graph developers. It’s designed for business users. Many business users often want to see \*all\* the data with the records that will be filtered out marked rather than being not shown at all.   
Why might they want to do that? To validate that the filter expression is not filtering out any records that they want to keep.   
If you don’t show all the records, there’s no way to know if you’ve lost a record (dropped/filtered out) that you actually wanted to keep.   
  
=====================================   
  
What is the minimum information required to have a valid ruleset?   
        source record format (from EME Dataset Object) —> template designer is responsible   
                for setting this   
        target record format (from EME Dataset Object) —> template designer is responsible   
                for setting this   
        expressions or rules that map valid values to each of the variables in the target —>   
                written by the person who is editing the configuration   
  
        In terms of workflow, we need the person who is editing the configuration to choose   
        the source & target record format in the configuration BEFORE they start writing   
        the mappings (editing the BusinessRulesPopup)   
  
If you want to test the ruleset (check the results of running the mappings on actual data), you need:   
        1- valid ruleset (mappings are provided by the person who writes the configuration)   
        2- test data for the source + lookups (provided by the developer who writes the template)   
 (for file & record testing)   
  
============================   
  
For Exercise 3D   
Will the ruleset have test data if we don’t provide a SampleData path?   
No — it will give an error saying that there is no Layout for the Filter FlowData object.   
        (filter\_input\_flow).   
That’s because the Filter FlowData object doesn’t have a source. The DataLink is linked to the SampleData.   
  
Options?   
1. Since we’re just using files —rather than using DataLink method for attaching sample data to the graph, we can simply use the sample\_data\_path property of FlowData object for the Filter. In the automatic assignments, we can use an expression to set the sample\_data\_path to either be the sample data (if chosen) or the input file (if not sample data has been provided).   
OR   
2. Redesign the SampleData object using automatic assignments— create a variable to hold the sample data path (and configure the FileBrowser to use this new variable as the SourceTargetValue). If the variable value is blank, then the value of the SampleData object is the input\_file. If the variable value is not blank, then the value of the SampleData object is the variable value.   
OR   
3. If the user chooses not to provide SampleData — just set the BusinessRules to not use test data. Set it to “Validate only” rather than “Compute & Validate”   
  
=============================   
  
To include a Lookup in a ruleset, what information do we need to supply?   
        record format for the lookup —> supplied by the person who writes the template   
                design the template to ask the user to select a lookup file (from EME Dataset Object)   
  
        name / label of the lookup —> supplied by the person who writes the template   
                remember that the label of the lookup needs to match the label of the Lookup File   
                component in the graph.   
  
        key for the lookup —> chosen by the person who edits the configuration   
  
        one more thing…   
                lookup(“Lookup File Label”, …..)   
                default key —> an expression or value or field (usually from the source data) that   
                        matches the key of the lookup file   
                —> chosen by the person who edits the configuration   
  
  
Think about how we’re going to design our template to get this information.   
What UI elements do we need to add????   
        EmeBrowser control (inside a Popup) —> get the EME Dataset Object for the Lookup   
        Key Picker (inside a Popup) —> get the key for the lookup file   
                (default key is chosen inside the ruleset control)   
  
        What else would be nice to add to give feedback to the user?   
        ViewData on the data that they’ve chosen for the lookup file   
  
In order to configure a Lookup File inside a template, you need to use a special type of variable —> compound lookup variable   
  
  
=============   
  
Learning more about templates in Express>It   
        explore the Exercise 4 template   
        explore the appconf-examples templates (I’ll make those visible in our EME TR)   
  
        Read the Help!!!   
                especially the “Template Developer’s Guide”   
  
        If you think of any general Express>It questions after class —   
                rbuchheit@abinitio.com   
        Feel free to send me your questions & I’ll be happy to answer them for you!   
  
  
Your accounts on the training server are active until February 28 2016.   
I will leave up the Express>It installation with these exercises for as long as I can… probably at least a month or so.   
  
  
The Help Library has \*all\* our documentation.   
If you want to learn how to administer a particular product…   
set the Scope to “All topics”, choose the product — 99% of the time, there will be a section on “Administration”.

**Metaprogramming:**

**DAY1:**

Monday through Friday, 4:30 PM — 8:30 PM IST   
We’ll take a 20 minute break at about 6:30 PM IST   
  
The PPT slides for the class are already on your desktops. You should see a file PDL\_course.pdf on your remote desktop desktop.   
  
I’ll email any notes that I write in Notepad to everyone after class each day.   
  
There is no “homework” for this class. We’ll do all the exercises during the class.   
However there is a “case study” and I’ll provide the instructions for it after class on Friday if (optional) you want to try it.   
  
+++++++++++++++++++   
  
We have 4 specific graphs that all do the same thing:   
        read a source file   
        aggregate the data   
        use the result to create a lookup file that is used by other graphs   
  
If we have a lot of these, it makes sense to create 1 generic graph and use parameters to describe the differences:   
        input file URL   
        input file record format   
        rollup key   
        rollup transform   
        output file URL   
        output file record format   
        additional optional components to do additional operations (normalizing the data)   
  
If we have a generic graph, we have three ways to get the parameter values for the graph:   
(1) Use a pset that some person configures in the GDE.   
(2) Use metaprogramming to \*compute\* some/all of the parameter values.   
(3) Create a template in Express>It and allow users to create configurations to fill in the parameter values.   
These three techniques can be combined — so you can use metaprogramming to compute some of the parameter values, use Express>It to enable some people to create psets for the remaining parameter values and/or use the GDE to enable other people to create psets for the remaining parameter values.   
  
  
When to use metaprogramming versus Express>It to get the value for a particular parameter?   
        — use metaprogramming when the parameter value can be specified as a variation on another parameter value.   
        examples: user selects the input record format, user selects 1 or more fields from the input record format, output record format = input record format minus those fields (we have a known method for computing the output record format)   
                                user selects the input record format, user selects 1 or more fields from the input record format, transform = max function applied to those fields (we have a known method for computing the transform rules)   
  
        — use Express>It when the parameter value can be literally anything — when there are no known algorithms/specifications/methods that always apply to create the record format or transform. In those cases, it’s better to create a really awesome template and let the user configure those in Express>It   
        you can use the record format editor and the ruleset editor controls to enable users to configure record formats, transforms, etc.   
  
By using metaprogramming, there is less work (fewer parameters) for the end user to configure. Metaprogramming saves time & effort for the end user when configuring psets for generic graphs.   
  
You can easily compute \*some\* of the parameters using metaprogramming. And whichever parameters don’t work well with metaprogramming could be configured using Express>It. You can combine these technologies as you see fit for your requirements.   
  
  
Typical design pattern for a graph that uses metaprogramming:   
        end user is asked to provide input specification (record format, file URL or table name) and some sort of information about how they want to proceed (list of fields to drop, which fields to aggregate, which key to use, etc.)   
        some or all of the remaining parameters are computed using metaprogramming functions (= this week’s class)   
  
  
=================================================   
  
How do parameters work in a graph?   
  
There are two types of parameters:   
        — “input parameters” : value for the parameter is provided by the end user (via pset, command-line, etc.) at run time   
        — “local parameters” : value for the parameter is computed by Co>Operating System at runtime.   
  
For local parameters, there are two values:   
        — “value” : that is assigned by the developer when the developer creates the graph   
        — “resolved value” : this is computed by the Co>Operating System at runtime by applying the Interpretation to its value. Resolved value is what is used by the components and transforms in the graph that reference the parameter. If a component has a reference to $ABC, then at runtime, the Co>Operating System substitutes the resolved value of ABC in place of that reference.   
  
  
  
Parameter Name                Value                                Interpretation        Resolved Value   
param1                                        xyz                                constant        xyz   
param2-bad                                $param1                        constant        $param1   
param2                                        $param1                        $-substitution        xyz   
param3                                        abc$param1                $-substitution        abcxyz   
param4-bad                                $param1\_abc        $-substitution        ERROR!   
        because it is ambiguous whether the parameter name is $param1 or whether it is $param1\_abc   
param4                                        ${param1}\_abc        ${}-substitution        xyz\_abc   
param5                                        $( id -n)                        shell        win\_rbuchheit   
param6                                        abc$(id - n)                shell        abcwin\_rbuchheit   
param7                                        $[ (date(“YYYYMMDD”))today())   PDL        20151026   
  
Interpretations:   
constant           
        resolved value = value   
        the value is read as-is and simply becomes the resolved value   
  
$-substitution   
        resolved value = value with any $-references replaced by the resolved value of that parameter   
  
${ }-substitution   
        same idea as $-substitution, except that parameter references are written as ${PARAM\_NAME}   
        why? in some cases, the $ reference can be ambiguous   
        ${ } clearly defines what is the parameter name and what isn’t   
  
shell   
        resolved value = value computed by the Korn shell by executing the command   
        anything inside $( ) is sent to the Korn shell & executed there.   
        the result of the execution is the value that is substitution in place of the $( )   
  
PDL (Parameter Definition Language)   
        PDL is a \*superset\* of all of the above Interpretations.   
        In the grid above, I could use PDL as the Interpretation for any of those parameters & I would get exactly the same result.   
        You can combine different types of syntax in the same parameter value and PDL will be able to resolve all of them.   
        So I could write this:   
                $AI\_SERIAL/$(id -n)/${FILENAME}\_today.dat   
        and it would be able to resolve correctly if the Interpretation was set to PDL.   
  
        PDL has an additional syntax that it understands, named “inline DML computation”   
        This syntax is written as $[ ]   
        Inside the $[ ], you can use Ab Initio DML syntax and functions to compute value.   
        You write a DML expression (= code that returns a single value), the resolved value of the   
                parameter is computed by the DML engine evaluating that expression.   
  
          
        In the latest version of the Co>Operating System & GDE, shell interpretation is no longer   
        an option. But you can still use $( ) syntax if you set the Interpretation to PDL — the code inside   
        $( ) will still be interpreted by the Korn shell. But there is no longer an option for shell   
        interpretation.   
        Why?   
        Shell interpretation is not recommended unless there is no other way to do it.   
        Using inline DML computation (typically the way that you would replace a shell interpreted   
        parameter) is \*faster\*, works better with dependency analysis, easier to use (you should   
        know how to write DML expressions, but you might not be so familiar with Korn shell   
        commands)   
        Why is inline DML computation faster than shell interpretation?   
                with inline DML computation, we’re using the Co>Operating System, computation is   
                        embedded in the graph (we don’t have to start anything extra)   
                with shell interpretation, we need to start a subshell from Korn shell, send the command,   
                        wait for the command to run, get the result, close the subshell for \*EACH\* shell   
                        interpreted parameter (=slower)   
        Why doesn’t dependency analysis work with shell interpretation?   
                security — there are commands that you might be allowed to run in dev that you can’t   
                        run (or don’t have permissions to run in prod)   
                filesystem versus EME — in the filesystem “rm $AI\_SERIAL/data” might be fine but if   
                        you do that in the EME, you could remove a lot of dataset objects that you wanted to   
                        keep. Running some commands in the EME might be very destructive in a way that   
                        you don’t expect (“rm -rf” would be disastrous in the EME!)   
  
  
        Why do we still have $-substitution around then? Why is that still an option?   
        There are a lot of cases where you simply want to write something like:   
                $AI\_SERIAL/myfile.dat — $-substitution is perfectly good for that   
                many of our components — by default — use $-substitution as the default parameter   
                interpretation for their parameters   
                backwards compatibility for our built-in components & their parameters   
  
                moving forward, it’s perfectly okay to use PDL for anything   
  
                  
============================   
  
A bit more on inline DML computation.   
  
Earlier, I wrote this:   
                $AI\_SERIAL/$(id -n)/${FILENAME}\_today.dat   
        and it would be able to resolve correctly if the Interpretation was set to PDL.   
  
  
You can combine inline DML computation with these other types of parameter references.   
        So I could write this as a parameter value:   
                $AI\_SERIAL/$(id -n)/${FILENAME}\_$[ (date(“YYYYMMDD”))today() ].dat   
        and it would be able to resolve correctly if the Interpretation was set to PDL.   
  
Inside the $[ ] (inline DML computation), there are some interesting syntax rules.   
For the examples below, assume all parameters are set to use PDL interpretation.   
  
Name                Value                                                                Resolved Value   
t\_date                $[ (date(“YYYYMMDD”))today() ]        20151026   
t\_year                $[ string\_prefix(t\_date, 4) ]                2015   
t\_month        $[ string\_substring(t\_date, 5, 2) ]        10   
t\_day                $[ string\_suffix(t\_date, 2) ]                26   
delim                ,                                                                        ,   
delim\_tx        string($delim)                                        string(,) —> invalid syntax   
delim\_t        string($”delim”)                                        string(‘,’)  —> valid syntax   
  
Notice that we reference the value of t\_date (earlier parameter value) without a $.   
That is the correct syntax.   
\*Inside\* the $[ ], the DML engine acts as if you are writing expressions inside a DML function. And that the parameter values defined earlier in the graph & sandbox are variables inside that function.   
  
  
Example of a reformat function:   
out :: reformat (in) =   
begin   
        let t\_date = “20151026”; // this is an implied data type — it automatically guesses “string”   
                                                                  // based on the initial value of “20151026”   
  
        out :: date\_month((date(“YYYYMMDD”))t\_date);   
end   
  
Notice that in the transform, I refer to the value of the variable by simply writing t\_date.   
That’s how parameter references work \*inside\* $[ ].   
  
  
Same example using parameters:   
Name                        Value                                                                Resolved Value   
t\_date                        “20151026”                                        “20151026”   
out                                $[ date\_month((date(“YYYYMMDD”))t\_date) ]        10   
  
Notice that the expression that I used to compute the output for the function can be used exactly (with no changes to the syntax) to compute the value of the parameter inside $[ ].   
  
  
  
Writing record formats & transforms is going to require that you work with quotation marks.   
— in a record format or transform, string constants are always enclosed in quotation marks.   
  
string(“,”) name = NULL(“”);   
out.field1 :: “x”;   
  
“,”, “”, and “x” are all string constants   
        default values for fields in a record format   
        delimiter values for fields in a record format   
        giving a string constant as the value for a field in a transform   
all of these are situations where you use string constants   
  
With metaprogramming, we are going to be working with string constants — as we create/modify record formats and transforms.   
  
One way to add quotation marks around a parameter value reference:   
        $”delim”   
if the resolved value of delim is |   
        then $”delim”   
resolves to “|”   
  
If you write   
        “$delim”   
then the resolved value is “$delim”   
The quotation marks on the outside of the $ prevent the $-reference from being resolved. This is a way of specifying constant interpretation when you have a $ in your string.   
  
Another way to add quotation marks around a value is to concatenate them:   
        $[ “‘“ + delim + “‘“ ]   
I am using two different sets of quotation marks. In both situations, I am writing double quote-single quote-double quote (“ ‘ “). The double quotes are because I am writing a string constant. The single quote is the string value.   
If the value of delim is |, the resolved value would be:   
        ‘|’   
  
You can do it the other way as well:   
        $[ ‘“‘ + delim + ‘“‘]   
Here I am using single quote-double quote-single quote (‘ “ ‘). The single quotes are because I am writing a string constant. The double quote is the value of that string.   
If the value of delim is |, then the resolved value would be:   
        “|”   
  
  
Another way to work with quotation marks is to escape them. (Above I used two different types of quotation marks to distinguish between the string value versus the quotation marks around the string value.) Here I’m going to use the same type of quotation mark, but I’ll escape the string value with \ (backslash). The backslash is called an “escape” because it tells the DML engine to ignore the special character (quotation mark) and treat it as a simple string.   
        $[ “\”” + delim + “\”” ]   
  
I wrote double quote-backslash-double quote-double quote (“ \ “ “). The outer quotes are indicating that we have a string constant. The value of that string constant is \”  — the backslash is there to escape the quote and treat it as a string constant rather than a special character.   
Resolved value, if delim = |, is   
        “|”   
  
This also works with single quotes (‘ \’ ‘)   
        $[ ‘\’’ + delim + ‘\’’ ]   
resolves to   
        ‘|’   
  
In DML syntax for record formats & transforms, you can use either single quotes or double quotes for string constants. So any of the above would be correct.   
  
Extremely common design pattern for record formats is to set up the begin-end of the record format as string constants and then compute the fields inside that hardcoded string   
record   
        $[ string\_join(for (let i, i < 5) : “string(‘,’) field” + (string(“”))(string(“”))i + “;”,   
                           “\n  “)]   
end   
  
===================================   
  
There are two types of script generation — you might see the earlier type if you are working with older graphs.   
        Version 2.13 compatible   
        Dynamic   
  
In order to use PDL interpretation, your script generation for the graph must be set to “Dynamic”   
If you are editing a graph and you don’t see “PDL” available as a possible Interpretation for a parameter, check the Script Generation setting:   
        Settings > Graph Settings > Script   
If you need PDL, change the generation to “Dynamic” — this may (not always) cause changes in how the parameter values are resolved and in the behavior of the parameters, so you should retest the graph if you make that change. (There are some rare side effects.)   
  
Starting in V2.14, dynamic script generation was available but not the default   
Starting in V3.0, the default script generation for new graphs is now “Dynamic”   
Starting in V3.1, the default interpretation for parameters in new graphs is PDL   
  
====================================   
  
In DML, there is syntax for a conditional expression — you see conditional expressions typically used   
        — select\_expr parameter in FILTER BY EXPRESSION   
and           
        — as the condition for if-logic                if (<conditional expression>) …   
  
  
You build a conditional expression using comparison operators (such as ==, >, <, >=) or unary operators (such as !, not) or using a function that returns a boolean (true or false)   
In Ab Initio, we don’t have a boolean data type. Instead, we use integer values   
        1        —> true   
        any other number (including 0) —> false   
  
If you have syntax that requires a conditional expression, it will return either 1 (true) or 0 (false)   
  
Notice that in our exercise (ex1), the Korn shell command returns 1 or 0   
        $( if [ $ARCHIVE = “yes” ]; then print 1; else print 0; fi ]   
  
So we can replace this with a simple conditional expression in DML and it will automatically return 1 or 0. The equality comparison operator in DML is ==   
        $[ ARCHIVE == “yes” ]   
and this will have a value of 1 (true) or 0 (false)   
  
  
In DML — there is no boolean data type.   
In parameters — there is a Boolean type Attribute. If you set the parameter Type to Boolean, then the parameter can have 1 of 2 values: True or False.   
  
Parameters do have Types — has two effects on the parameter:   
        — validates the resolved value that is computed as valid for that Type and there’s an error if   
                it’s not valid   
        — which Editor is used to edit the value if you click the Edit pencil in the Parameters Editor.   
  
For example, if you set the Type of a parameter to be “Record Format” then the resolved value of that parameter must be a valid DML record format (valid syntax) AND if you click the Edit pencil, it will open a Record Format Editor!   
  
  
Other attributes for parameters:   
        Input — either checked (default) or unchecked   
                                checked — we call this an “input parameter” and the end user is expected to provide a   
                                value for the parameter at runtime either via command-line, pset, or the GDE Input   
                                Values Editor   
                                unchecked — we call this a “local parameter” and the graph developer is expected to   
                                provide a value that will be resolved at runtime. Value for the parameter is saved as   
                                part of the graph (in the .mp file)   
  
        Required — either checked (default) or unchecked   
                                checked — end user or graph developer (depending on whether it is an input param   
                                or not) is required to provide a \*NON-BLANK\* value for the parameter   
                                unchecked — okay (not an error) for the parameter’s value to be blank.   
                                in either case, the resolved value can be blank and there is no error   
                                in the Korn shell, if a parameter’s value is blank — that parameter is “unset” — it is   
                                        regarded as the parameter not having a value at all.   
                                this is a way to make providing a value for a parameter optional (unchecked)   
  
        Export to Environment — either checked or unchecked (default)   
                                checked — parameter + value are exported into the Korn shell environment   
                                unchecked — parameter + value are local to the graph subshell only.   
                                Most of the time you want the default (unchecked = not exported)   
                                (see background below)   
  
        Kind — Keyword (default), Environment, Implicit, Positional   
                        specifies how you want to provide a value for Input parameters when you run the graph   
                        from the command line   
  
                        Keyword —                 air sandbox run my\_graph.mp -PARAM\_NAME value   
                        Positional         —         air sandbox run my\_graph.mp value   
                        Environment —         export PARAM\_NAME=value   
                                                                air sandbox run my\_graph.mp   
  
                        Keyword is a superset of Environment. If you do not specify a value for the parameter   
                        on the command-line, Co>operating System checks the environment to see if it is set   
                        there. Almost 100% of the time — you want to use default, which is Keyword.   
  
                        Positional is not recommended. There only for backward compatibility. It’s confusing &   
                        difficult to use if you have optional (= not required) or environment parameters.   
                                  
                        Implicit is used internally for our component parameters. Should not be used for graph   
                        parameters.   
  
        Location — we’ll talk about later this week & we’ll do some demo + exercises with it.   
  
—> Background before we can explain what Export to Environment does.   
When you run a graph, Co>Operating System starts a subshell. Graph runs inside that subshell. Any parameters are visible only to the graph running in that subshell. Most of the time that’s all you need. But sometimes the parameter values need to be visible to other programs (outside of the graph’s subshell). Typical cases for that are:   
        — when you are passing the parameter value as an argument to an external command   
                (like a command that you run using Run Program component)   
        — when you are overriding the value of a Co>Operating System configuration variable   
                        such as AB\_REPORT, AB\_JOB, AB\_SUMMARY\_FILE, etc. Anything that starts with AB\_   
        — when you have a parameter reference inside an external file that is saved on disk (for earlier   
                versions of the Co>Operating System before PDL interpretation and dynamic script   
                generation was available)   
In these cases, the parameter & its value must be exported to the environment so that it can be “seen” by the Korn shell or the external program.   
  
—> You have probably heard that you should \*NOT\* check “Export to Environment” unless you are sure that you need it. Why?   
For all parameters that are exported to the environment, we have to add the parameter name + value (its “definition”) to a command line command that we use to run the graph. Some OS have a limit on how many characters you can use in a command-line command. If you export a lot of parameters, you might hit that limit & you can’t run the graph! The OS fails the command.   
Also, exporting a lot of parameters to the environment slows down the graph startup somewhat.   
  
  
  
Requirement for a parameter is that is has a single non-NULL value.   
        — parameter values cannot be NULL   
        — value can be “scalar” or “compound”   
                        scalar value is a simple value like 1, -1, 0, “”, “xyz”, “20151026”   
                                        strings, decimals, integer, date/datetime   
                        compound value can be either a vector or a record   
  
If you want to hold multiple values in a single parameter, you would use a compound type like vector or record. (These are \*not\* parameter types — instead, the parameter Type is set to String. Co>Operating System recognizes record & vector syntax and allows that as String for parameters. Same thing with dates & decimals. There is an Integer parameter type.)   
  
        vector — list of values that all have the same DML type   
        record — collection of fields, each field can be a different type   
  
        you can have a vector of records (very very common type used in metaprogramming)   
  
========================================   
  
Syntax for declaring a vector — in metaprogramming we almost exclusively use length-prefixed vectors.   
        let string(10)[integer(4)] field\_names = allocate();   
  
“let” is declaring a variable inside a transform or begin-end expression   
string(10) is the data type for each element in the vector   
[integer(4)] is the length-prefix. What is the maximum value for integer(4)?   
        integer(1)                        127   
        integer(2)                        32,000   
        integer(4)                        about 2 billion   
if you use integer(4) as your prefix, your vector can have anywhere between 0 and 2 billion elements. Vector is dynamically sized based on its contents. So if it’s empty, the size is 0. If you put in 20 elements, the size is 20. Maximum size would be 2 billion.   
allocate() — when you declare a vector, you need to allocate some initial memory for its storage.   
  
        let record   
                string(int) variety;   
                integer(4) num\_plants;   
                date(“MMM-DD”) sowing\_date;   
        end [int] veggies = allocate();   
  
“let” is declaring a variable named “veggies”, and we’re allocating with allocate() some initial memory.   
What is the data type for the elements in this vector? record, specifically:   
        record   
                string(int) variety;   
                integer(4) num\_plants;   
                date(“MMM-DD”) sowing\_date;   
        end   
The size of the vector is specified with “[int]” —> length-prefixed vector. “int” is a shortcut for “integer(4)”   
Shortcuts are:   
        integer(1)                        —> char   
        integer(2)                        —> short   
        integer(4)                        —> int   
        integer(8)                        —> long   
  
You could write this:   
        decimal(“”)[char] num\_list;   
This is a length-prefixed vector. Each element in the vector is a delimited decimal — delimited with the NUL char (also written as “\0”). And the size of the vector can vary dynamically between 0 and 127 elements.   
  
If you’ve ever done any programming in C, you’ll have heard of NUL-terminated strings — that’s a common data type to use in C. In Ab Initio DML, you can write a NUL-terminated string (or decimal or date) as:   
        string(“”)   
or   
        string(“\0”)   
Common standard for described a delimited (variable-sized) data type.   
  
  
We have declared this type:   
        let record   
                string(int) variety;   
                integer(4) num\_plants;   
                date(“MMM-DD”) sowing\_date;   
        end [int] veggies = allocate();   
  
        veggies = [vector [record variety “Sungold” num\_plants 2 sowing\_date “JAN-03”],   
                                                [record variety “Brandywine” num\_plants 1 sowing\_date “JAN-05”],   
                                                [record variety “Jubilee” num\_plants 5 sowing\_date “JAN-01”],   
                                                [record variety “Mr. Stripey” num\_plants 3 sowing\_date “JAN-02”]];   
  
I put some values in this variable. And now I want to access the value of the 3rd element in the vector. What do I write?   
        veggies[2]   
the result is —> [record variety “Jubilee” num\_plants 5 sowing\_date “JAN-01”]   
Remember that vectors are indexed starting at 0.   
  
How do I access the value of the “variety” field in element at index 2 from veggies?   
Use dot syntax to access the value of a field inside a record (in.field1, out.cust\_id, lookup(“Some Lookup”, in.id).manager))   
        veggies[2].variety   
the result is —> “Jubilee”   
  
============================================   
  
This is a type:   
        utf8 string(int)   
charset = utf8   
form = string   
size = int   
  
A field consists of:   
        a type   
        a field name   
        optionally allowing NULL values   
        optionally having a default value   
        optionally having a condition   
        optionally having a comment   
  
This is a field:   
        utf8 string(int) address = NULL(“”) // here is a comment;   
        string(“,”) name;   
                this has an implied charset — on Linux/Unix/Windows it is ascii   
  
A record format consists of zero or more fields. Starts with the word “record” and it ends with the word “end”.   
        record   
                // fields listed here   
        end   
  
  
======================================   
  
In my graph….   
you can try this yourself in the Parameters Editor — create  new graph in BDS\_construction sandbox.   
  
store\_rec\_format                $[ record\_info (read\_type (AI\_DML + “/store.dml”)) ]   
type\_of\_first\_element        $[ store\_rec\_format[0].dml\_type ]   
last\_element                                $[ length\_of(store\_rec\_format) - 1]   
type\_of\_last\_element        $[ store\_rec\_format[last\_element].dml\_type ]   
  
In the computation for store\_rec\_format, I used the read\_type() function. There is also read\_file(). What’s the difference?   
        Both of them reads the contents of a file from the filesystem.   
        Result is a string containing the contents   
  
        Difference is   
                        read\_type() also validates the contents of the file to make sure that the contents are   
                                a valid record format   
                        read\_file() just reads the contents   
                        read\_transform() validates the contents of the file to make sure that the contents are   
                                a valid transform   
  
read\_type() is nice to use with record\_info() because read\_type verifies that you are really reading a record format \*before\* you try to break it up in the dml\_field\_info type.   
  
==============

**DAY 2:**

Record formats (and transforms & keys) are often saved as string values. Functions like read\_type() and read\_transform() return the record format/transform as a string value. Parameters that store record formats & transforms, act like those values are string values.   
  
let string(int) in\_dml = “record string(10) fname; string(10) lname; end”;   
  
  
Three functions to add & remove fields from a record format.   
        remove\_fields (string rec\_format, string fieldname)   
OR   
        remove\_fields (string rec\_format, [vector <list of fieldnames>])   
  
        add\_field (string rec\_format, string fieldname, string dml\_type, [optional args for other attributes for field])   
  
        add\_fields (string rec\_format, dml\_field\_info\_vec fields)   
                dml\_field\_info\_vec is a built-in type — a vector of dml\_field\_info records   
                you can create a dml\_field\_info record by using the make\_field() function   
  
  
There are two ways to represent a collection of fields (and/or record format)   
        — as a string   
        — using a structured dml\_type, such as dml\_field\_info   
Some metaprogramming functions use strings as input arguments & return strings as output. Other metaprogramming functions work with the structured types.   
  
Search Help Library for “Metaprogramming tasks using DML functions”   
        This categorizes the metaprogramming functions according to what you want to   
        do: work with data types, work with keys, work with transforms, etc.   
        Help definitions for each function will usually suggest other functions that you   
        might need to use   
  
  
  
=========================   
  
I have a delimited type:   
parameter         value           
in\_type                 “string(‘,’)”   
delimiter                $[ type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})].value ]   
  
$[ type\_info(in\_type).attributes ] —> returns a vector of records that contains various attributes for the type. There is no guarantee that there is a delimiter record and there is no guarantee on the order (you can’t assume that the delimiter attribute is always first — for example).   
  
We need to search this vector to see if there is a delimiter attribute & if yes, return the value of that delimiter attribute. Use a vector function, like vector\_search(), to find the element in the vector that has the delimiter key and return the value from that record.   
The 2nd argument for vector\_search() needs to be the same data type as the elements in the vector.   
In the “attributes” vector that type\_info() returns, the elements in the vector are records with two fields: key, value.   
The 3rd argument for vector\_search() is optionally a key that specifies which field(s) you want to match on.   
  
This expression:   
vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})   
returns the element # of the record that matches the 2nd argument based on the “key” field value.   
  
We have to access that element of the original vector   
type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})]   
And we want the value field from that element   
type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})].value   
  
Extremely helpful to build up these expressions step by step & test as you go.   
Extremely helpful to make each step be a separate parameter so that you can easily debug one step at a time. Then when you’ve the thing working —> combine all the steps into a single parameter value.   
  
Problem — in the expression above, if I provide an initial type (in\_type) that doesn’t have a delimiter, the expression fails with an error. Error is occurring because vector\_search returns -1 (no match found) and -1 cannot be used as the index into the vector.   
We can use functions such as   
        null\_if\_error()   
takes multiple arguments and it starts with the first expression and evaluates it. If it returns an error, it tries the next expression. If none of the expressions return a non-error value, then the function returns NULL.   
        null\_if\_error (complicated\_vec\_thing, “unknown”)   
  
Here’s what the final expression would look like:   
null\_if\_error(type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})], [record key “delimiter” value “no delimiter”]).value   
  
Very complicated & it repeats some parts — type\_info(in\_type) is repeated twice.   
This might be a good case for using block expression. Enables you to declare variables inside an expression and then return a single result.   
The $[ ] syntax requires that the code inside the $[ ] is a valid DML expression   
        expression —> code that returns a single result (scalar like string/date/decimal or compound type like record or vector)   
  
Basic syntax for a block expression   
$[ begin   
   <you can do whatever you want — declare local variables, write loop expression & statements, if logic…>   
   result :: <value that the block expression returns>;   
end ]   
  
  
We can take this very complicated expression   
$[ null\_if\_error(type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})], [record key “delimiter” value “no delimiter”]).value ]   
  
and turn it into a much simpler block expression:   
$[ begin   
        let dml\_type\_info in\_type\_info = type\_info(in\_type);   
        let record   
                string(int) key;   
                string(int) value;   
        end dummy\_record = [record key ‘delimited’ value ‘NULL’];   
        let int index = vector\_search(in\_type\_info.attributes, dummy\_record, {key});   
  
        result :: if (index >= 0) in\_type\_info.attributes[index].value   
                                else dummy\_record.value;   
end]   
  
  
Block expression is simpler than the single expression   
AND you can use the PDL debugger to step through it to debug it.   
In the Parameters Editor   
        Debug > Debug PDL   
  
  
The member operator only works for vectors.   
If you want to find if a particular string is embedded in some other string, use string\_index   
        string\_index (str1, “test”)   
The result of string\_index can be used as a conditional expression because it returns 0 (false) if the given string is not found in the original string. Returns 1 or some other number if the given string is found — 1 or some other number is equivalent to true.   
  
However, starting in version 3.2.5, we have slice expressions that work on both strings & vectors.   
        you can write something like this:   
                vec\_value [2:4]        —> returns a new vector containing elements 2, 3, 4 from the original vector   
                string\_value[2:4] —> returns a new string which is the equivalent of string\_substring(string\_value, 2, 3) — chars 2, 3, and 4 from the original string.   
    
  
=================================   
  
Let’s say that we have a setup like this:   
        your graph has three input parameters   
                in\_dml                —> input dml record format, as a string value   
                in\_delim        —> a delimiter that is used in the in\_dml   
                out\_delim        —> delimiter you want to use to replace in\_delim in in\_dml   
  
        you write this to do the replacement   
                out\_dml                $[ string\_replace (in\_dml, in\_delim, out\_delim) ]   
  
We want to replace every occurrence of the in\_delim with the specified out\_delim.   
  
This works perfectly fine if the in\_delim is something like “|”.   
Can you think of a situation where it wouldn’t work???? What’s a delimiter that would cause problems??   
        if in\_delim was a value such as “,” “;” “-“ “\_” any alphabetical char would be an issue, etc.   
        these characters are all used in DML syntax to define fields   
  
        if you had this:   
                string(“;”) fieldA;   
  
        and you specified in\_delim as “;” —> when you replaced with “,”, the result would be:   
                string(“,”) fieldA,   
        invalid DML syntax!!   
  
Using string\_replace is a very simple way to do the task (replace one delimiter with another) but it works in only limited circumstances.   
Should you use it?   
        — yes if you know what sort of delimiters the input record formats will be using   
        — no if you don’t know   
Use the simplest technique possible as long as you have some idea that it will work for the inputs that you expect to get.   
  
==============================================   
  
I want to add a new field to the \*beginning\* of an existing record format.   
The problem is — add\_field() and add\_fields() adds the new field to the \*end\* of the existing record format.   
  
How can we add a field to the beginning of an existing record format? join\_types()   
        join\_types takes 2 arguments, both of which need to be records.   
  
The existing graph has 4 input parameters:   
        input\_file   
        input\_dml   
        new\_field\_name   
        new\_field\_type   
  
I want to create a new record format that has 1 field — specified by the new\_field\_name and the new\_field\_type parameter values.   
Two methods (string manipulation OR metaprogramming functions) to do it:   
(1) string manipulation — I can just concatenate the parts together to form a simple record format   
  
Notice that I wrote out the record format syntax   
record   
        new\_field\_type new\_field\_name;   
end   
and then I put quotation marks around the things that are string constants (things that are \*not\* variables or parameters) and I use concatenation (+ operator or string\_concat() function) to put everything together   
“record “ + new\_field\_type + “ “ + new\_field\_name + ”; end”   
  
OR   
(2) metaprogramming functions   
        add\_field (“record end”, new\_field\_name, new\_field\_type)   
  
OR (if you had multiple fields this would be better)   
(3) structured metaprogramming functions   
        add\_fields(‘record end’, [vector make\_field(new\_field\_name, new\_field\_type), ….])   
  
================================   
  
Exercise 2   
Add field at the end — we can just use add\_field() — because it puts the new field at the end of the record format.   
$[ add\_field (input\_dml, new\_field\_name, new\_field\_type) ]   
  
Make the fieldnames lowercase   
$[ string\_downcase(input\_dml) ]   
        this works because all DML syntax is lowercase (data types, “record”, “end”, etc) and because none   
                of the fields use default values that have uppercase letters in them   
        the opposite (making the fieldnames uppercase — would not work, you couldn’t just use string\_upcase   
                because DML syntax like “RECORD” and “END” is invalid.   
        if any of your fields had default values that were uppercase strings — using string\_downcase() would   
                not be good.   
  
Remove the newline field   
$[ remove\_fields (input\_dml, “NEWLINE”) ]   
        “NEWLINE” is a string constant — it is not a variable or parameter reference   
        so it must have quotes around it, just like any other string constant.   
  
When we try the other pset (Run > Select Input Values —> chose pset/02b.add\_fields.pset)   
        this fails because there is no field named NEWLINE in the record format from the new pset.   
        in the new pset, the record format has a field named “newline”   
  
How do we fix this? We want to remove any fields named “NEWLINE” or “newline” — whatever is in the record format.  One way is to downcase the input\_dml before applying remove\_fields… This is the simplest (and thus an excellent idea) way that works for this particular requirement:   
$[ remove\_fields (string\_downcase(input\_dml), “newline”) ]   
This will still fail if there is no field that has a name like newline.   
  
Another way — if you were trying a bunch of different options — more general way to solve this issue is to use null\_if\_error()   
$[ null\_if\_error (remove\_fields(input\_dml, “NEWLINE”), remove\_fields(input\_dml, “newline”), remove\_fields(input\_dml, “Newline”), input\_dml) ]   
This works even if there is no newline field at all in the record format!   
  
Even better — combine both techniques…   
$[ null\_if\_error (remove\_fields (string\_downcase(input\_dml), “newline”), input\_dml) ]   
  
  
+++ For down casing the fieldnames +++++   
The more general method that always works is to loop over the vector, downcase the fieldnames only and create a new record format from that. This is fairly complicated.   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        for (let i, i < length\_of(input\_rec\_info))   
                input\_rec\_info[i].name = string\_downcase(input\_rec\_info[i].name);   
                  
        result :: add\_fields (“record end”, input\_rec\_info);   
end ]   
  
  
Important thing   
        — you need to know what you want as output   
        for the block expression above, I wanted a record format (string) as output   
        — you need to know what the functions you are using will return   
        record\_info returns a vector of records — that is \*NOT\* a record format   
        — at the end, I know that I can add that vector of records into a string record format using the   
                add\_fields() function   
  
  
I wrote this index for loop statement which works correctly to downcase the names of the fields in a record format. This method here (immediately below) is the best performance that gets the correct result.   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        for (let i, i < length\_of(input\_rec\_info))   
                input\_rec\_info[i].name = string\_downcase(input\_rec\_info[i].name);   
                  
        result :: add\_fields (“record end”, input\_rec\_info);   
end ]   
  
I started to write this — and then realized that it wouldn’t work. This is an element for loop statement.   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        for (let field in input\_rec\_info)   
                field.name = string\_downcase(field.name);   
                  
        result :: add\_fields (“record end”, input\_rec\_info);   
end ]   
This doesn’t work — the output record format is the same as the input record format — the field names are not downcased, they are still uppercase.   
This cannot work because of this part “let field” —> creates a variable instance that contains a record for the current field. In the loop, we are changing the name of the field \*inside\* that variable instance to be downcase, but we are \*NOT\* changing the name of the field inside the vector.   
  
  
There is one way to get this to work:   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        let dml\_field\_info\_vec output\_rec\_info = allocate();   
  
        for (let field in input\_rec\_info) begin   
                field.name = string\_downcase(field.name);   
                output\_rec\_info = vector\_append(output\_rec\_info, field); // running vector\_append inside a loop is slow   
        end   
                  
        result :: add\_fields (“record end”, output\_rec\_info);   
end ]   
  
Another way to get this to work:   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        let string(int) output\_dml = “record end”;   
  
        for (let field in input\_rec\_info) begin   
                field.name = string\_downcase(field.name);   
                output\_dml = add\_field(output\_dml, field); // running add\_field inside a loop is slow   
        end   
                  
        result :: output\_dml;   
end ]   
  
The problem with both of these methods is that they are significantly slower than the method I showed you at the beginning (use an index for loop and add the fields at once, at the end).   
  
+++++   
Performance tip: minimize the amount of work that you do inside a loop. The instructions inside the loop should be as simple as possible.   
+++++   
  
  
Bonus question — how would you add a field in the \*middle\* of an existing record format?   
Let’s say that we want the end user to choose which position to add the field into. Position 1 = first field in the record format, position 3 = third field in the record format, etc.   
  
Need to declare input parameters   
        input\_dml   
        new\_field\_name   
        new\_field\_type   
        new\_field\_position   
  
Do you think this will be easier to do working with the record format as a string or as a vector?   
The problem with working with the record format as a string is that we don’t know what to search for in the record format so that we can find the position where we want to insert the new field. If we reframed the question to be — how can we add a new field after an existing field in the record format — AND the end user gives us the name of the existing field — then we might be able to do this as a string.   
  
But without knowing where in the string to go or what field name to search for in the string, using the record format as string doesn’t work.   
  
We want to work with the fields in the record format as a vector. Now the fields are indexed in the dml\_field\_info vector and we can insert the new field into that vector.   
  
You can use looping syntax to insert a new element into position in an output vector — but, what you have to do is check the index with a if-statement to know when you’re at the right position & then insert two elements at that point — the new element and then the existing element that was already at that position. Complicated.   
  
Become familiar with the \*vector\* functions that are built-in to DML. There are lot of things you can do with vectors that don’t require looping.   
There are vector functions that are similar to what you can do with string functions.   
  
Let’s reframe the question again. What if I had a string value “abcdef” and I wanted to insert “X” at position 3. How would you do that with string functions?   
        string\_substring(str, 1, position-1) + “X” + string\_substring(str, position, length\_of(str))   
  
Our record format problem is extremely similar to this — except that we have elements in a vector instead of characters in a string. Vector functions can help us:   
        vector\_slice —> equivalent of string\_substring   
        vector\_concat —> equivalent of + (concat)   
  
        in this version of the Co>Operating System vector\_concat takes only 2 arguments. (Most recent version   
        3.2.5 allows more).   
        vector\_append — appends a value to a vector   
  
        vector\_slice is not exactly the same as string\_substring   
                string\_substring (str, starting\_position, length)   
                vector\_slice (vet, starting\_position, ending\_position)   
        Also vectors start numbering at position 0 and strings start number at position 1.   
  
  
$[ begin   
        let in\_rec\_info = record\_info(input\_dml); // input\_dml as a vector of records (each record is a field)   
  
        result :: add\_fields(“record end”, vector\_concat(vector\_append (vector\_slice(in\_rec\_info, 0, new\_field\_position-2), make\_field(new\_field\_name, new\_field\_type)), vector\_slice(in\_rec\_info, new\_field\_position-1, length\_of(in\_rec\_info)-1));   
end ]   
  
Remember that you can build this more simply using multiple parameters or variables inside your block expression so that you can test each part as you go.   
  
=======================================

**DAY 3:**

Design pattern for metaprogramming:   
        end user provides a record format for the source data   
        use metaprogramming functions to compute the other record formats used   
                in the graph   
        making modifications to the source record format   
                — adding or dropping fields   
                — modifying the field names   
                — changing the data types   
  
Design pattern for metaprogramming:   
        end user provides a record format for the source data   
        use metaprogramming functions (and often looping expressions & statements) to compute transform rules for the transform components in the graph   
        looping through the fields in the source record format & adding rules to:   
                — aggregate some of the fields (sum, count, min, max…)   
                — modify the values of some fields   
                — assign fields to the output record format that you computed also with   
                        metaprogramming functions   
  
  
Here’s a very simple transform function signature:   
out :: reformat (in) =   
begin   
end   
  
But there’s a lot more that you can do… you can have multiple arguments (like for a Join transform function), the arguments can have data types.   
  
You have the choice of using string functions to create the function signature…   
let string(int) xfr = “out :: reformat (in) =   
begin   
end”   
  
Or you could use metaprogramming functions in cases where the function signature is more complicated:   
let string(int) xfr = make\_transform(“join”, [vector make\_arg(“in0”, “record”), make\_arg(“in1”, “record”)] );   
  
  
Metaprogramming tip:   
        write out an example (hardcoded) of the record format or transform that you want to create   
        decide which parts are hardcoded string constants —> enclose those in quotes   
        replace everything else with the appropriate parameter or variable references   
  
  
Let’s say that I have an input parameter named:   
        new\_field\_name   
  
I want to create a transform function that looks like this:   
out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
        out.new\_field\_name :: “X”;   
end   
  
In metaprogramming, what I would write is this:   
‘out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
        out.’ + new\_field\_name + ‘:: “X”;   
end’   
  
I could also do this:   
let string(int) xfr = ‘out :: reformat (in) = begin out.\* :: in.\*; end’;   
xfr = add\_rule(xfr, ‘out.’ + new\_field\_name, ‘“X”’);   
          
        single quote — double-quote — X — double-quote — single-quote   
        single quotes are because “X” is a string constant inside the add\_rules function   
        double quotes are because X is a string constant inside the transform that we want to create   
                with the metaprogramming functions   
  
You could write this with the quotes in the other order   
        double quote — single quote — X — single-quote — double-quote   
like this:   
let string(int) xfr = ‘out :: reformat (in) = begin out.\* :: in.\*; end’;   
xfr = add\_rule(xfr, ‘out.’ + new\_field\_name, “‘X’”);   
The order of the quotes — whether you use single or double quotes for the outer quoting — doesn’t matter. All that matters is that you’re consistent — all the single quotes are on the outside & double quotes on the inside — or vice-versa.   
  
  
If I want to write a rule that looks like this:   
        out.new\_field\_name :: string\_trim(in.new\_field\_value);   
  
and new\_field\_name & new\_field\_value are input parameters, then my add\_rule function looks like this:   
        xfr = add\_rule(xfr, “out.” + new\_field\_name, “string\_trim(in.” + new\_field\_value + “)” );   
  
The add\_rule function automatically adds the “::” assignment operator and the “;” semi-colon that ends the rule.   
In your add\_rule function, you write only the code necessary to describe the left-hand side (lhs) and the right-hand side of the rule (rhs).   
  
==========================   
  
We are extracting data from a mainframe and loading it into a database table.   
The DML record format for the mainframe data looks like this:   
record   
   string(10) FIELD1;   
   string(10) FIELD2;   
   string(20) FIELD3;   
end   
  
Typically mainframe record formats use capital letters for the field names and they are usually fixed-size fields.   
  
The DML record format for the database table looks like this:   
record   
         string(“\x01”) field1;   
        string(“\x01”) field2;   
        string(“\x01”) field3;   
        string(1) newline = “\n”;   
end   
  
Typically record formats generated for database tables use lower case field names, and usually have delimited fields and a newline field at the end.   
  
We want to use metaprogramming to create a transform to transform the mainframe data format into the database table format.   
  
How do we approach this problem?   
1- Write out examples of what your inputs might look like — we have that above.   
2- Write out an example of what your result should look like, based on those inputs.   
3- Code up your metaprogramming to create that result (enclose string constants in quotes, concatenate with the parameterized values)   
4- Test with different inputs to check that your code is generic enough to work correctly in many situations.   
  
in\_dml =                                                                         out\_dml =   
record                                                                                record   
   string(10) FIELD1;                                              string(“\x01”) field1;   
   string(10) FIELD2;                                                string(“\x01”) field2;   
   string(20) FIELD3;                                                string(“\x01”) field3;   
end                                                                                        string(1) newline = “\n”;   
                                                                                        end   
  
I want to write a transform to transform the data from the in format to the out format.   
I also want to make sure to trim the strings from the input (string\_trim)   
I want the result to be a transform that looks like this:   
out :: reformat (in) =   
begin   
        out.field1 :: string\_trim(in.FIELD1);   
        out.field2 :: string\_trim(in.FIELD2);   
        out.field3 :: string\_trim(in.FIELD3);   
end   
Do I need a rule to assign to the newline field in the output? No — it has a default value in the record format for the output, so it will just pick up that default value.   
  
Obviously I want a loop here to create the rules for the transform. Should I loop over the in record format fields or the out record format fields?   
If I loop over the output, I have to remember to skip the newline field because we don’t need a rule for it.   
In \*this case\* looping over the input is a better idea. (Most of the time, looping over the output fields is a better idea because — to have a valid transform, every output field must be assigned a value.)   
  
$[ begin   
        let string(int) xfr = make\_transform (“reformat”, [vector make\_arg(“in”)]);   
        let in\_fields = record\_info\_item(in\_dml, “name”);   
        let dml\_rule\_vec out\_rules = for (let fname in in\_fields) :   
                                make\_rule (“out.” + string\_downcase(fname), “string\_trim(in.” + fname + “)”);   
          
        result :: add\_rules(xfr, out\_rules);   
end ]   
  
(You could also write this code by using add\_rule() inside a for loop statement — instead of what I did [use make\_rule inside a for loop expression] — but that will be much slower than what I did. Avoid using add\_rule() or add\_field() inside loops. Much better to create a vector of fields with make\_rule() or make\_field() and then use add\_rules()/add\_fields() outside the loops.)   
  
What are the metaprogramming functions that we learned yesterday?   
        type\_info — extracts information from a data type like “string(5)”   
        read\_type — reads a record format from a file on disk and returns it as a string   
        record\_info — extracts information from a record format into a vector of fields   
                                                containing the field information (name, dml\_type, comments, etc.) for each   
                                                field   
        record\_info\_item — extracts one attribute of the fields from a record format into a vector   
                                                example: extract the names of all the fields in the record format   
                                                        OR extract the DML types of all the fields in the record format   
  
I want to loop over the names of the fields from the input record format.   
        record\_info\_item   
  
=============   
Exercise 3   
Hint — you want the finished transform to look like this:   
out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
        out.new\_field\_name :: ‘X’;   
end   
  
Except that “new\_field\_name” should be replaced with the actual field name selected as the input parameter value by the end user.   
  
In yesterday’s exercise, we added a new field to the output record format.   
Output record format = Input record format + new\_field\_name   
  
In today’s exercise, we want to create a transform for transforming the data from the input record format to the output record format. The transform needs to have a wildcard rule (for all the fields that are the same in both in & out) plus a rule to assign a value for the new output field.   
  
Approach #1 — string manipulation — works well in this situation because there is only 1 parameter/variable reference in the transform (to new\_field\_name). Very simple to do.   
$[ “out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
        out.” + new\_field\_name + “:: ‘X’;   
end” ]   
  
  
Approach #2 — use metaprogramming functions   
$[ begin   
        let string(int) xfr = make\_transform(“reformat”, [vector make\_arg(“in”)]);   
        xfr = add\_rule (xfr, “out.\*”, “in.\*”);   
        result :: add\_rule(xfr, “out.” + new\_field\_name, “‘X’”);   
end ]   
  
  
=======================   
  
        let dml\_rule\_vec  out\_rules = allocate();   
versus   
        let out\_rules = allocate(); —> this won’t work!!   
  
But this will work:   
        let out\_rules = for (let field\_name in in\_fields) : make\_rule (“out.” + field\_name, “string\_trim(in.” + field\_name + “)”);   
  
What’s the difference?   
In the second example — we are assigning a value to out\_rules using a function (make\_rule) with a known return type (dml\_rule). DML engine can figure out the data type of out\_rules based on the function that we’re using to assign a value to it. This practice is called using an “inferred type”.   
  
  
Simpler example:   
        let x = 0;   
DML engine decides that “x” is an integer because you initialized it with an integer value. Default integer type is integer(4).   
  
        let y = “abc”;   
DML engine decides that “y” is a string(3) because you initialized it with 3 character string value.   
  
  
If you wrote   
        x = x + 1;   
it would work because “1” is a valid value for an integer(4)   
  
If you wrote:   
        y = y + “x”;   
it would fail because “abcx” is \*not\* a valid string(3).   
  
Inferred types can be used \*ONLY\* if you initialize the variable with a function that returns a known type — record\_info, transform\_info — most of the metaprogramming functions do this. It’s a nice shortcut because I can never remember the built-in type names.   
  
  
In my solution for Exercise #3, I wrote this:   
        let string(int) xfr = make\_transform(“reformat”, [vector make\_arg(“in”)]);   
Why didn’t I use an inferred type for xfr? make\_transform has a known return type (string).   
  
If I had done this:   
        let xfr = make\_transform(“reformat”, [vector make\_arg(“in”)]);   
Because the DML engine would have created a fixed-size string with the fixed-size of the return value of make\_transform.   
That’s okay — until I try to do this:   
        xfr = add\_rule(xfr, “out.\*”, “in.\*”)   
This line would fail saying that the type for “xfr” is not large enough to add additional characters to it!   
  
Recommendation:   
Don’t use inferred types for strings — use length-prefixed strings   
        let string(int) xfr ….   
        let string(int) dml ….   
  
Do use inferred types for functions that return records such as…   
         record\_info, transform\_info, type\_info   
Do use inferred types for functions that return structured field or rule information…   
        make\_rule, make\_field, make\_arg   
  
Functions that return strings are:   
        add\_rule, add\_rules, add\_field, add\_fields   
Use string(int) as the type for variables that hold those values.   
  
========================   
  
Reading record formats & transforms that are “landed to disk” (files saved in your sandbox), two access methods:   
  
1- Use a function to read the contents of the file as a string   
        read\_type() —> reads & validates types (record formats)   
        read\_transform() —> reads & validates transforms   
        read\_file() —> general files   
  
OR   
2- Setup the parameter accordingly   
        Value        —> path to the file that you want to read   
        Resolved Value —> contents of the file   
  
        (a) Choose the Type of the parameter to be something like Record Format or Transform   
        (b) Set the Location attribute of the parameter to “File”   
  
  
Both of these methods — assuming you use read\_type() or read\_transform() — validate the contents of the file to make sure that they are a valid record format or transform.   
With the functional method (#1), you can store the path to the file as a separate parameter if you want…   
name                        value                                                        resolved value   
in\_dml\_path        $AI\_DML/f.dml                                /path/to/your/sandbox/dml/f.dml   
in\_dml                        $[ read\_type(in\_dml\_path)] <contents of the dml file>   
  
Now you can use both these parameters as inputs to other computations.   
    
If you don’t need to access both the contents of the file & the path to the file, then using the location (Method #2) is more convenient. Then you have just a single parameter — that is easy to use as an Input parameter (end user can easily browse for the file name in the sandbox) and the resolved value is the contents of that file.   
  
One thing to remember — make sure that the Location attribute is set correctly   
        if the value of the parameter is a path —> Location = File   
        if the value of the parameter is embedded record format or transform —> Location = Embedded   
  
        in both these cases, the \*resolved\* value of the parameter will be the actual record format or transform contents.   
  
Another reason to use method #2 (location) is that the user can specify how they want to provide the value at runtime…   
        they can provide a transform or record format from a file (Location = File)   
OR   
        they can type in (copy & paste…) a transform or record format directly (Location = Embedded)   
  
AND you can switch between them… so   
        if the end user starts with Location = File and choose a file   
        and then switch to Location = Embedded, the GDE will prompt the end user (just like it does when doing this on the Ports tab of a component) if you want to use the contents of that file as the embedded value. Enables end user to import the contents of an existing record format or transform and then make edits to it and use that as the value of the parameter!   
    
======================================   
  
You “declare” a parameter once. Declaration of a parameter consists of choosing a Name, a Type, whether or not it is an Input parameter, whether or not it is Required, whether or not it is Exported to Environment, what is its Kind.   
—> think of a parameter like a variable, you can declare a variable only once   
  
You can “define” a parameter as many times as you want. The Definition of the parameter consists of giving it a Value, choosing an Interpretation, and choosing a Location.   
—> think of a parameter like a variable, you can set the value of the parameter (and reset it) as many times as you want.   
  
You declare parameters in your project parameters (.project.pset) and in your graph (in the .mp file).   
You can define parameter values in the project parameters, in sandbox parameters (.sandbox.pset), in the graph (.mp file), in pset (my\_graph.pset), in the environment, and on the command-line.   
  
Because Location & Interpretation are attributes of the parameter \*definition\*, they can be changed by the end user in the Input Values Editor.   
  
  
Plans are the same as graphs —   
        you can declare parameters in the plan (my\_plan.plan)   
        and you can define parameter value + location + interpretation in the pset (my\_plan\_2.pset)   
Plans have an Input Values Editor just like graphs do.   
  
If you’re doing this from the command-line, there are command-line commands that enable you to change the interpretation/location/etc.   
        “air sandbox parameter”   
                can be used for sandbox parameters, graph parameters, and plan parameters   
                just give the path to the pset that you want to edit.   
  
======   
  
We want to find the most recently modified .dat file in the $AI\_SERIAL directory. Use this file as input to the graph.   
  
directory\_listing(AI\_SERIAL, “\*.dat”) —> vector   
loop over the elements in that vector, use file\_information() function to get the modification date, and keep track of which one has the maximum value so far.   
  
$[ begin   
   let file\_list = directory\_listing(AI\_SERIAL, “\*.dat”);   
   let integer(8) max\_mod\_date = 0;   
   let integer(8) cur\_mod\_date = 0;   
   let max\_index = 0;   
   let index = 0;   
  
   for (let file in file\_list) begin   
        cur\_mod\_date = file\_information(AI\_SERIAL + “/“ + file).modified;   
     if (cur\_mod\_date > max\_mod\_date) begin   
                max\_mod\_date = cur\_mod\_date;   
                max\_index = index;   
     end   
        index = index + 1;   
   result :: AI\_SERIAL + “/“ + file\_list[max\_index];   
end ]   
  
==========================================   
  
Question:   
I want to read in the contents of a file and extract some information from that file.   
  
Sometimes the format of the file is simple — max\_id used by some other graph.   
123   
  
Sometimes the format of the file is kind of like a lookup file — where you have a different key (and its related value) on each line.   
Example is a list of error codes   
0,no error   
1,file not found   
2,file is not readable   
3,file is too large   
  
We want the end user to specify the key value and have the value of a parameter be the related value for that key.   
parameter        value                                                        resolved   
key\_number        2                                                                <input value specified by the end user>   
error\_msg        $[ ???? ]                                                “file is not readable”   
Equivalent of doing a lookup — only you’re doing it in the parameters!   
  
CAUTION: this is an extremely bad idea if the file is large. You should design the graph to use a regular lookup.   
But if the file is quite small, this is a simple & effective way to lookup a value from that file.   
  
We know that read\_file() will read the contents of a file on disk.   
read\_file(AI\_REFERENCE + “/error\_codes.dat”) —> contents of the file as a string value   
  
Now I have a string — how can I locate a specific line in that file.   
—> convert it to a vector   
—> use something like vector\_search to locate the matching key value.   
  
I want to have this:   
error\_code\_lkup =        [vector [record key 0 error\_msg “no error”],   
                                                                [record key 1 error\_msg “file not found”],   
                                                                [record key 2 error\_msg “file is not readable”],   
                                                                [record key 3 error\_msg “file is too large”]]   
  
vector\_search(error\_code\_lkup, [record key ERROR\_CODE error\_msg “”], {key}) —> this will tell me which element in the vector matches that key   
  
error\_code\_lkup[vector\_search(error\_code\_lkup, [record key ERROR\_CODE error\_msg “”], {key})].error\_msg  —> this will give me the error\_msg for the matching key   
  
  
I want to take this string:   
0,no error   
1,file not found   
2,file is not readable   
3,file is too large   
  
and reinterpret it as this vector:   
[vector [record key 0 error\_msg “no error”],   
                                                                [record key 1 error\_msg “file not found”],   
                                                                [record key 2 error\_msg “file is not readable”],   
                                                                [record key 3 error\_msg “file is too large”]]   
  
What type should I reinterpret as?   
Each line is a comma-delimited key, followed by a newline-delimited error\_msg   
0,no error   
  
We use this record format to describe each line in the string:   
record   
  string(“,”) key;   
  string(“\n”) error\_msg;   
end   
  
Each line will become an element in the vector — but we don’t know how many elements are in the vector to start with — so we’ll use something called a “self-sized vector”   
A self-sized vector is written like this:   
        <data type> [ ] fieldname;   
  
In transforms, using length-prefixed vectors is the recommended approach:   
        <date type> [int] variable\_name;   
But in reinterpret\_as — the issue is that you don’t have a length-prefix in your data. So a self-sized vector — it keeps adding elements to the vector until it runs of out of elements, works well with reinterpret\_as.   
  
reinterpret\_as( record string(“,”) key; string(“\n”) error\_msg; end [ ], error\_code\_str)   
—> this won’t work — it will be weird output   
  
read\_file returns a length-prefixed string — length-prefix of the string is how many chars in the string. The length-prefix data type is integer(4). We need to skip over the first 4 bytes in the value that read\_file returns because that is the length-prefix and it will mess up the reinterpret\_as.   
reinterpret\_as( record string(“,”) key; string(“\n”) error\_msg; end [ ], error\_code\_str, 4)   
  
the third argument (4) is called the offset — it tells reinterpret\_as to skip over the first 4 bytes in the string and start converting to the vector after that.   
  
  
Put this all together into a single block-expression…   
$[ begin   
        let string(int) error\_msg\_str = read\_file(PROJECT\_DIR + “/ref/error\_codes.dat”);   
        let error\_msg\_vec = reinterpret\_as( record string(“,”) key; string(“\n”) error\_msg; end [ ],   
                                                                                                error\_code\_str, 4);   
        let index = vector\_search(error\_msg\_vec, [record key KEY error\_msg “”], {key});   
  
        result :: null\_if\_error(error\_msg\_vec[index].error\_msg, “KEY NOT FOUND!”);   
end ]   
  
This is a template that you can use to do simple lookups in your Parameters Editor.   
The file path that you read will be different.   
The data type that you use for reinterpret\_as might be different   
The dummy record that you use for vector\_search will depend on the data type that you used for reinterpret\_as   
Which field you choose as output in the result will also depend on your data type.   
  
  
===============================

**DAY 4:**

Exercise 3 — working with files   
Open 03.RMF.mp   
  
1- Write a transform for REFORMAT that uses the directory\_listing() function to output the list\_of\_filenames output field.   
Hint: use the Help Library examples for directory\_listing() to find an example of how to write the pattern matching argument of directory\_listing() so that you do \*not\* output dot files (e.g., “.” and “..”) in the list   
  
<You should be able to run the graph at this point to test your work so far.>   
  
2- Replace the current “out” port format for READ MULTIPLE FILES. Use metaprogramming to add a new field (the filename) to the RMF\_input\_type record format.   
  
3- Modify the transform of READ MULTIPLE FILES   
        — uncomment the line “type input\_input” and edit it to look like this:   
                type input\_type = ${RMF\_input\_type};   
  
        — edit the reformat() function to add a rule to output the filename (you do not need to use metaprogramming for this)   
  
        — make sure to set the Interpretation of the transform so that it can evaluate the ${ } reference used for the input type   
  
4. Run the graph to test your work. Don’t worry if the transaction\_amt field looks weird in the output data — that’s a problem with the data that we supplied for this exercise.   
  
====================================   
  
If you plan to use READ MULTIPLE FILES to reformat the data that it reads from the files, then you have three record formats to deal with:   
        — in port record format —> which is a list of filenames,   
                typically has a record format like this:   
                record   
                        string(int) filename;   
                end   
  
        — out port record format —> which is the result of reformatting the data inside   
                        READ MULTIPLE FILES   
                in our case, the out port record format = original contents of the files + filename   
                we want to do this:   
                $[ add\_field (RMF\_input\_type, “filename”, “string(int)”) ]   
  
        — record format that describes the contents of the files as they are read \*before\* they   
                are reformatted (before the filenames are added — the filenames are \*not\* present   
                in the original files — we’re using the reformat() transform inside RMF to add those   
                filenames to the data)   
                you specify the record format for the \*contents\* of the original files using the   
                transform in RMF. You have a line like this:   
                type input\_type = ${RMF\_input\_type};   
  
===============   
  
In earlier versions of the Co>Operating System   
        the only way to get information from the database into a parameter value was to use   
        a Korn shell interpreted parameter and use a utility like “m\_db”   
  
In recent versions of the Co>Operating System   
        we have been adding metaprogramming functions that enable you to access the database   
        using functions + $[ ] syntax (PDL interpretation)   
        advantage: significantly faster than the earlier method   
                when you use the database metaprogramming functions —> we collect all the   
                ones that you are using together into a single connection to the db   
                when you use m\_db utility, each use of m\_db is a separate database connection   
  
Example:   
        $[ db\_get\_int (AI\_DB + “/training.dbc”, “select max(id) from transactions”) ]   
  
================   
  
Some parameters need context — parameters that use a record format   
        parameters that have Type set to Transform (because the transform needs an input record format + output record format to validate it!), Key Specifier (because you need to know the input record format to choose the key), etc.   
  
Context comes from the \*graph\* design.   
  
If you have a SORT component and you Export the “key” parameter of the SORT component to use the parameter name “sort\_key”   
then in your parameter editor, if you have sort\_key Type = Key Specifier, then when you edit the sort\_key value with the Edit pencil, it will show the “in” port record format from the SORT component in the graph.   
  
1- Create a parameter in the Parameters Editor, set its type accordingly (for keys — Key Specifier).   
2- In a component in the graph, choose a component parameter — such as “key” for SORT — and click the “Export” button.   
3- In the Export dialog, change the name of the parameter to match the parameter that you created in step #1.   
  
Step #2 — the export — links the parameter to the context of that component. “Context” — record formats that component uses in the graph design.   
  
Enables the end user to be able to click the Edit pencil & see a Key Specifier Editor populated with the fields from the record format of the component that uses that parameter.   
  
Shortcut   
1- Start with the component in the graph — choose the component parameter, click Export button, and create a new parameter name.   
In the Parameters Editor, a new parameter will be created with the correct Type + context.   
  
=====================   
  
Key metaprogramming functions (these are not our most consistent moment)   
  
Design patterns you might use with add\_key()   
        sort\_key = {first\_name descending; last\_name descending}   
        more\_fields = {cust\_id}   
        $[ add\_key (sort\_key, key\_info(more\_fields)[0]) ]   
  
make\_key() is different than make\_field() and make\_rule()   
        make\_field & make\_rule take string arguments and create a structured output   
        make\_field (“fieldname”, “string(int)”) —> dml\_field\_info record   
        make\_rule(“out.\*”, “in.\*”) —> dml\_rule\_info record   
  
        make\_key   
                input argument is a dml\_key\_info\_vec   
                output result is a string (key string that can be used in a component)   
  
          
Because the key functions are confusingly (and inconsistently) named and because you often don’t use modifiers in the key values, you can get away with a simple string + vector manipulation trick.   
  
Simple key with no modifiers, might look like this:   
some\_key        { first\_name; last\_name}   
  
1- Strip off the spaces and curly braces using string\_filter\_out   
                key\_filtered          $[ string\_filter\_out (some\_key, “{ }”) ]   
          
        “{ }” is left curly brace followed by a space character followed by a right curly brace   
  
2- Use string\_split to convert the string into a vector   
                key\_vec         $[ string\_split(key\_filtered, “;”) ]   
  
        result —> [vector “first\_name”, “last\_name”]   
        Now I have a vector of strings & I can manipulate it as a vector. I can add additional   
        fields…   
                key\_vec\_all          $[ vector\_append (key\_vec, “cust\_id”) ]   
        result —> [vector “first\_name”, “last\_name”, “cust\_id”]   
          
Now I might want to convert that vector back into being a key string. Reverse the steps.   
3- Use string\_join to convert the vector into a string   
                key\_vec\_all\_str   $[ string\_join( key\_vec\_all, “;”) ]   
  
        result —>  first\_name;last\_name;cust\_id   
  
4- Add the curly braces back into   
                final\_key\_str                “{“ + key\_vec\_all\_str + “}”   
          
        result —> {first\_name;last\_name;cust\_id}   
  
  
IMPORTANT: This method does not work if you have modifiers in the key.   
If you were more careful about how you remove spaces — loop over the vector elements to trim them — then it could work even if your keys have modifiers.   
  
================================   
  
The graph for exercise 4 has a SORT component in the graph that is “conditioned out” — on the Condition tab — the value of the condition is always False.   
This component is always disabled.   
  
The key parameter is set to use the parameter value $fields\_to\_drop   
$fields\_to\_drop is an Input graph parameter.   
  
This is a roundabout way to enable the end user to select one or more fields from the list of input fields.   
We’re not using $fields\_to\_drop as a key — we’re taking advantage of the fact that the Key Specifier Editor enables you to select one or more fields from the record format. We’re linking the $fields\_to\_drop parameter with a record format + editor (Key Specifier Editor) as a user-interface decision — easy way for the end user to see a list of the input fields & choose one or more of them to drop from the data.   
  
There’s a REFORMAT component in the graph — why not associate the parameter with REFORMAT?   
REFORMAT doesn’t have a key parameter. There’s no way to get the same user interface interaction (an editor that enables the end user to select one or more fields from the input record format).   
  
SORT is not the only option for the workaround. You could use any component that has a key parameter and does not transform the data —> another option would be DEDUP SORTED. Again, in both situations, the component does NOT run. It is simply there to provide a link between the input record format + graph parameter so that the end user has a nice user interface.   
  
If you didn’t care about the user interface — you don’t need this.   
You would simply set fields\_to\_drop to be a string parameter and expect that the user would enter a delimited list of fieldnames. (That is not too difficult to do — you would need to do some input validation to make sure that the user formatted the list correctly. You should document the parameter to indicate what format you want for the list of fields.)   
  
====================   
  
Exercise 4   
        two input parameters:   
                in\_dml — value is a path to a dml file, supplied by the end user   
                                        resolved value is a record format (Location = File)   
  
                fields\_to\_drop — value is a key supplied by the end user using the Key Specifier Editor   
  
  
Option #1 — string + vector manipulation to remove fields\_to\_drop from the in\_dml   
        $[ remove\_fields (in\_dml, string\_split(string\_filter\_out(fields\_to\_drop, “{ }”), “;”)) ]   
  
        even better: use null\_if\_error() around remove\_fields to do error handling   
                        remove\_fields fails if the specified fields are not found in the in\_dml   
  
Option #2 — use metaprogramming functions + loop   
        $[ remove\_fields (in\_dml, for (let field in key\_info (fields\_to\_drop)) : field.field\_name ]   
          
        disadvantage of this method: more difficult to write   
        advantage of this method: works correctly even if the key fields have modifiers on them   
        even better: add null\_if\_error() to do error handling   
  
=====================================   
  
In a typical transform (no metaprogramming being used), if you have a operation (that has several steps) that you need to do repeatedly — you might create a user-defined function and save that function in an external file (utilties.xfr, string\_cleansing\_functions.xfr, etc.)   
  
in utilities.xfr, I might create a simple function that concatenates a person’s given name & family name — but also deals with blank & NULL values.   
  
out :: full\_name (string(int) given\_name, string(int) family\_name) =   
begin   
        let string(int) fname = if (not is\_null(given\_name) and not is blank(given\_name))   
                                                                 given\_name + “ “   
                                                        else “”;   
        let string(int) lname = first\_defined(family\_name, “”);   
  
        out :: fname + lname;   
end   
  
  
How do I use this function in the transform of a component in a graph?   
Include the file where I’ve saved the function in….   
  
include “~$AI\_XFR/utilities.xfr”;   
  
out :: reformat (in) =   
begin   
        out.customer\_name :: full\_name (in.first\_name, in.last\_name);   
end   
  
  
In metaprogramming — what if we want to use a user-defined function to compute the value of a parameter?   
In a block expression   
$[ begin   
        result :: ….   
end ]   
you cannot have global variables or include statements.   
  
Solution: define the value of a parameter AB\_DML\_DEFS and include the file containing the user-defined functions or define the user-defined function as the parameter value.   
  
Value of AB\_DML\_DEFS is \*not\* a file path. It is the functions themselves — as an embedded value or the contents that are read from a file.   
  
============   
  
Really neat example of a highly generic graph that uses a significant amount of metaprogramming…   
        Help > Examples > Differencing graph   
  
Great resource to inspect for ideas on how to do things, best practices, etc.   
  
Graph compares two files of any type — and generates a report with information about the differences between the files. Very customizable — you can ignore differences in certain fields, specify how to deal with NULL values, specify how to deal with differences in element values in a vector, etc.   
============

**DAY 5:**

In a typical design pattern — the end user supplies a source record format and chooses the key fields for the Rollup plus also choose which fields to aggregate.   
        in\_dml   
        rollup\_key   
        fields\_to\_aggregate   
  
In the “anti-rollup” design pattern — the end user supplies a source record format and choose which fields to aggregate (usually with min/max functions).   
        in\_dml   
        fields\_to\_aggregate   
The key for the Rollup is \*all\* of the remaining fields in the in\_dml that are not being aggregated. This is design pattern that analysts sometimes use — they are more interested in finding min/max values over a group than what the group key is.   
  
In Exercise 5 — we’ll be doing an anti-rollup.   
        We’ll compute the key for the Rollup component by subtracting the list of fields to aggregate from the list of fields in the input\_dml — with metaprogramming.   
  
=====   
  
Extremely common design pattern for metaprogramming — one that is used well in the exercise 5 graph (05.aggregate\_fields.mp)   
is the idea that vectors are easier to work with.   
Convert things to vectors — lists of items — and then you can use vector functions & loops to work with those lists.   
And then convert back to strings — transforms, keys, record formats — at the end.   
  
When combining multiple parameters into a single parameter computation — either using functional composition or using block expression — you only want to combine parameters that are \*not\* used elsewhere in the parameter computations.   
  
If fields\_to\_aggregate\_\_vec is used to compute both the record format and the transform —> then it needs to be a separate parameter so that you don’t have to recompute it both times.   
If aggregate\_info is used only to compute the record format, then it can be combined with the computation for the record format — it shouldn’t be a separate parameter.   
  
How can you find out where a particular parameter is referenced?   
Find!   
        Edit > Find in the Parameters Editor   
  
  
It is helpful to combine multiple computations into a single parameter using a block expression because you can use the PDL debugger (in the Parameters Editor, Debug > Debug PDL) to step through the computation, evaluate expressions in the context of the variables in the expression, and so on.   
  
========================   
  
Example use case   
  
We have an input record format (in\_dml) that is supplied by the end user as an Input parameter value, and it has a value like this:   
record   
  string(“”) first\_name;   
  string(“”) last\_name;   
  string(“”) address;   
end   
  
We want to check if the value of the input field in each record is valid & add a flag (true or false) to indicate whether or not the value is valid.   
  
We want to use metaprogramming to compute an output record format that looks like this:   
record   
  string(“”) first\_name;   
  string(“”) last\_name;   
  string(“”) address;   
  integer(1) first\_name\_is\_valid;   
  integer(1) last\_name\_is\_valid;   
  integer(1) address\_is\_valid;   
end   
  
How would you approach this problem? Any ideas??? How would you use metaprogramming to create the output record format? You’ll need to use add\_fields to in\_dml — how do you get the vector to indicate what fields to add?   
1- Extract the field names from the in\_dml in vector form. What function would we use? record\_info\_item   
2- Loop over those field names and add \_is\_valid to the name of each —> new vector.   
3- Use add\_fields to add the new vector to the existing in\_dml.   
  
$[ begin   
        let in\_fields = record\_info\_item(in\_dml, “name”);   
        let validation\_fields = for (let field in in\_fields) : make\_field(field + “\_is\_valid”, “integer(1)”);   
  
        result :: add\_fields (in\_dml, validation\_fields);   
end ]   
  
  
Let’s write the transform — we want the transform to look like this:   
out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
          out.first\_name\_is\_valid :: is\_valid(in.first\_name);   
          out.last\_name\_is\_valid :: is\_valid(in.last\_name);   
          out.address\_is\_valid :: is\_valid(in.address);   
end   
  
What steps do we need to do in metaprogramming to create this transform?   
1. Loop over the input fields and create an is\_valid output rule for each input field   
2. Add those rules to a simple reformat function that contains a wildcard rule.   
  
$[ begin   
        let in\_fields = record\_info\_item(in\_dml, “name”);   
        let rules\_vec = for (let field in in\_fields) : make\_rule (“out.” + field + “\_is\_valid”,   
                                                                                        “is\_valid(in.” + field + “)”);   
        let simple\_xfr = add\_rules (make\_transform (“reformat”, [vector make\_arg(“in”)]), “out.\*”, “in.\*”);   
        result :: add\_rules (simple\_xfr, rules\_vec);   
end]   
  
  
Question is:   
        can we combine these two block expressions into a single block expression that computes both the record format AND the transform?   
        that way, we need to loop over the in\_fields only once.   
  
        yes!   
  
        in my original code, I used for loop expressions   
                for loop expression works well when you are generating one vector   
  
        I want to use one loop to generate \*two\* vectors — one vector containing fields for   
                the output record & the other vector containing the rules for the transform   
        I need to use a loop statement to do that   
        In a loop statement, you use vector\_append to add new elements to an existing vector.   
  
$[ begin   
        let in\_fields = record\_info\_item(in\_dml, “name”);   
        let dml\_field\_info\_vec validation\_fields = allocate(); // must explicitly declare the type here   
        let dml\_rules\_vec rules\_vec = allocate(); // must explicitly declare the type here as well   
        let simple\_xfr = add\_rules (make\_transform (“reformat”, [vector make\_arg(“in”)]),   
                                                                        “out.\*”, “in.\*”);   
  
        for (let field in in\_fields) begin   
                validation\_fields = vector\_append (validation\_fields,   
                                                                                                make\_field(field + “\_is\_valid”, “integer(1)”));   
                rules\_vec = vector\_append (rules\_vec,   
                                                                                make\_rule (“out.” + field + “\_is\_valid”,   
                                                                                                       “is\_valid(in.” + field + “)”));   
        end   
          
        result :: [record dml add\_fields (in\_dml, validation\_fields)     
                              xfr   add\_rules (simple\_xfr, rules\_vec) ];   
end ]   
  
Does anyone know how to return two (or more) values from a block expression or a function?   
I want this block expression to return \*BOTH\* at the same time, a record format and a transform.   
I can’t use if-else — that only returns 1 thing.   
What kind of data type can I used to store two or more values that have different data types?   
Vector stores a list of values that all have the same data type.   
Record stores a set of fields, each fields can have a different type!   
  
Let’s say that the block expression above is assigned to a parameter named “validation”   
So in my components, when I want to refer to the record format that is computed for the validation parameter, I write:   
        $validation.dml   
  
and when I want to refer to the transform that is computed for the validation parameter, I write:   
        $validation.xfr   
  
In a PDL expression   
        $[ validation.dml ]   
        $[ validation.xfr ]   
  
  
Option #1   
In the expression above, I used vector\_append and make\_rule/make\_field to create a vector of rules or a vector of fields. Then I used add\_fields/add\_rules at the end to create record format or transform.   
        for (let field in in\_fields) begin   
                validation\_fields = vector\_append (validation\_fields,   
                                                                                                make\_field(field + “\_is\_valid”, “integer(1)”));   
                rules\_vec = vector\_append (rules\_vec,   
                                                                                make\_rule (“out.” + field + “\_is\_valid”,   
                                                                                                       “is\_valid(in.” + field + “)”));   
        end   
          
        result :: [record dml add\_fields (in\_dml, validation\_fields)     
                              xfr   add\_rules (simple\_xfr, rules\_vec) ];   
  
  
Option #2   
You can use add\_field or add\_rule inside the loop.   
        let string(int) out\_dml = in\_dml;   
        let string(int) out\_xfr = simple\_xfr;   
        for (let field in in\_fields) begin   
                out\_dml = add\_field(out\_dml, field + “\_is\_valid”, “integer(1)”);   
                out\_xfr = add\_rule(out\_xfr, “out.” + field + “\_is\_valid”,   
                                                                                                       “is\_valid(in.” + field + “)”);   
        end   
          
        result :: [record dml out\_dml   
                              xfr   out\_xfr ];   
  
  
Option #2 is \*significantly\* slower when you have a large number of fields. Page #129 in the PPT slides shows the difference in performance.   
Option #1 is the preferred option:   
inside the for loop   
        create a vector of fields using vector\_append and make\_field   
        create a vector of rules using vector\_append and make rule   
  
at the end, add\_rules (xfr, rules\_vec)   
        and add\_fields (dml, fields\_vec)   
  
  
  
==========================================   
Workflow for building & testing a reusable component (linked subgraph)   
  
1- Build the graph with hardcoded inputs & outputs, and test it to make sure that it works.   
2- Parameterize the inputs & outputs   
                inputs are typically input parameters, values supplied by the end user   
                outputs are typically computed with metaprogramming   
Test it with various combinations of inputs to verify that it is truly generic and works in all cases.   
3- Take the components that you want to reuse, put them in a subgraph, and move the appropriate parameters to the properties of the subgraph. Now the parameters that the components in the subgraph uses are properties of that subgraph.   
Test again to make sure it still works   
4- Save the subgraph in a separate file (Select subgraph, then File > Save Subgraph … As). Then you’ve got a parameterized subgraph that can be reused (as a component) in other graphs.   
  
In Exercise #3 (the one with the Read Multiple Files), we’ve already done steps #1 and #2. The directory & the input record format are parameterized, and the output record format is computed using metaprogramming.   
I’m going to demonstrate how to do steps #3 and #4.   
  
  
If you need to make changes to the component that you’ve saved   
—> You right click the subgraph in the Sandbox View (in the components folder inside your sandbox) and choose “Edit”. Making changes to the subgraph itself.   
—> For any instances of that subgraph that are used in other graphs, you need to update the instances with those changes.   
        option #1 — open the graph, right-click the linked subgraph and choose Update   
        option #2 — “air sandbox update” — you can update multiple graphs & components at the   
                        same time   
  
==========   
  
What is the difference between PDL and metaprogramming?   
  
PDL is a parameter evaluation syntax   
        superset of constant, $-substitution, ${ } substitution, and shell interpretation   
        plus it has a “inline DML computation” syntax $[ ]   
  
        inside $[ ], you can write Ab Initio DML expressions and have them evaluated by the   
        Co>Operating System DML engine.   
  
        there are things that you can use PDL for that don’t involve metaprogramming   
        e.g., simple computations like today’s date as part of the file path.   
  
metaprogramming — design technique for generic graphs   
        typical design pattern for generic graph :   
                end user supplies input information (data set information + record formats) plus   
                some amount of information about what they want to do (keys, functions to use,   
                which fields to drop, etc.)   
  
                rest of the metadata — record formats, transforms, key — are computed with   
                metaprogramming functions   
  
        set of functions that you can use to compute record formats, transforms, and keys for   
        the graph to use   
  
        you can do metaprogramming without PDL — way back before PDL became available in   
                V2.14, you could have two graphs: first graph computes the transforms and record   
                formats + second graph uses those transforms & record formats   
  
        PDL enables you to compute the record formats and transforms as parameters so that they   
        can be computed in the same graph that uses them.   
  
========   
  
[option #1] You don’t have to create the whole parameter value like this:   
$[ “record string(10) first\_name; string(10) last\_name; “ + new\_field\_type + “ “ + new\_field\_name + “;\nend ]   
  
[option #2 ]You could also do this:   
record   
        string(10) first\_name;   
        string(10) last\_name;   
        $[ new\_field\_type + “ “ + new\_field\_name + “;”]   
end   
  
[option #3] And of course, you could also do this:   
$[ add\_field (“record string(10) first\_name; string(10) last\_name; end“, new\_field\_type, new\_field\_name) ]   
  
Always lots of ways to do things — think about the complexity.   
The more complex it is — the more likely it is better that you use metaprogramming functions like add\_field, add\_rule, etc.   
The simpler it is — the more likely it is better to just use simple string functions or to embed the computation inside a hardcoded string, like option #2 above   
  
  
======================================   
  
Is AB\_DML\_DEFS specific to each graph?   
Yes. A graph can only see the AB\_DML\_DEFS that is defined in that graph.   
  
However, you can define a value for AB\_DML\_DEFS at the project level — in your projects parameters — then all the graphs in that project can see that definition of AB\_DML\_DEFS.   
  
It gets tricky if you want to define AB\_DML\_DEFS at the project level and then also for a specific graph. The graph value overrides the value of AB\_DML\_DEFS from the project level.   
Help Library has a lot of information on the interaction of this.   
  
========================================   
  
After class today, I’ll email instructions for a case study exercise — if you want to try that on your own. Exercise is to build a generic graph on your own from the beginning based on requirements given to you.   
If you do try the case study, let me know & I’ll be happy to take a look at your design OR give you advice if you get stuck at some point.   
  
Your accounts on the training server are active until February 28 2016.   
These materials (PPT slides on your desktop, your sandboxes, etc) are available at any time, the server is up 24/7 (except for routine maintenance for about 1 hour on Monday nights EST).   
  
Definitely encourage you to take a look at the Differencing graph example — very nice example of good design for a generic graph, uses AB\_DML\_DEFS, lots of metaprogramming, etc.   
        Help > Examples > Differencing Graph   
  
If you later think of any general questions that you have — I’m happy to help you:   
        rbuchheit@abinitio.com   
  
For customer projects, you should talk to support@abinitio.com — remember just let them know what customer this is for.

**CONDUCT IT:**

**DAY 1:**

4:30 — 8:30 PM IST   
        ~20 minute break at about 6:30 PM   
        Friday — ~2 hours on lookup overview   
                                dynamic lookups, interval lookups, created lookups   
                                use cases   
  
Plan>It —> earlier name for Conduct>It   
We changed the name from Plan>It to Conduct>It about 5 years ago.   
Same thing.   
  
======================================   
  
Conduct>It (plans) is \*not\* a scheduler — missing the main feature that schedulers have which is the ability to say “run this job at 2 AM”.   
Schedulers —> “run this job at 2 AM”   
Conduct>It —> can have triggers, executable dependencies, etc. (which many schedulers also have)   
  
Later — more on when it makes sense to use scheduler versus Conduct>It   
  
If you want to schedule a plan (written in Conduct>It) to run at 2 AM, you still need a scheduler — full featured scheduler such as Control Center, Autosys, Control M or something simple like cron   
  
=======================================   
  
When you have a dependency that is connected to the “success” connection point of a task   
        Task 1 ——>  Task 2   
  
We need to get past Task 1 (without a failure) before Task 2 starts.   
Two ways that we can “get past” Task 1   
        —> Task 1 runs and completes successfully   
        or   
        —> Task 1 is disabled for some reason   
                                due to a condition   
                                due to a configuration such that if it fails then it is disabled   
                                etc.   
  
If Task 1 is configured so that it is disabled if it fails (non standard, not the default behavior)   
        when you run the plan     
        if Task 1 fails —> disabled   
        then Task 2 starts   
  
        if Task 1 succeeds —> then Task 2 starts   
  
  
Another design option (standard default)   
        if you connect both the “success” and “failure” connection points of a Task   
        then you wouldn’t need to disable on failure —   
        the Task connected to the failure point would run if the previous task fails   
  
        Task 1 ——> Task 2   
                |   
                |   
            \/   
        Task 3   
  
        If Task 1 runs + succeeds or is disabled then Task 2 runs   
        If Task 1 runs + fails then Task 3 runs   
  
============================   
  
What is “Stdout” for a graph?   
        text that shows up in the Application Output window when you run graph in GDE   
        text that shows up on the command line when you run graph from command-line   
        “Execution starting”, …. “Done”   
        no data involved   
  
What is “Stderr” for a graph?   
        error message that you get if the graph fails — again what you would see in the   
                Application Output window in the GDE   
  
=============================   
  
  
Use the properties of a task to control which connection point is activated under various circumstances…   
        Timeout —> can either be a failure (failure point is activated) or disable (success point is activated)   
        Failure —> can either be a failure (failure point is activated) or disable (success point is activated)   
  
  
Default behavior for the start connection point for a task is “Any”   
        if any of the upstream branches are enabled, then the task runs   
                if one of the upstream branches is disabled for some reason, that’s okay   
                as long as at least one upstream branch is enabled   
  
        If you choose “All” (not default behavior) then all of the upstream branches must   
                be enabled in order for the task to run.   
  
  
        Typical design pattern for a conditional branch   
  
        < Condition > ——> Task 1 ——>   
                |                                                                     Task 3   
                |                                                                  >   
                \/                                                                  |   
                Task 2——————————   
  
        Task 3 runs either way when you have its “enabled branches” setting to “Any”   
        Either Task 1 will run + Task 2 is disabled   
        OR        Task 1 is disabled + Task 2 will run   
  
        If you change Task 3’s “enabled branches” setting to “All”   
        then it cannot run because there will never be a situation where \*all\* of its upstream   
                dependencies are enabled. It’s impossible.   
  
        Obviously a bad idea for this design. But there are other designs where it might make   
        sense (ones that don’t use the branch-out-then-branch-back-in design)   
        For example, it might make sense for the “Mailing Graph” task on pg 18 of PPT   
  
===================================================   
  
If I were to run the check\_dirs.ksh script on the command-line, I would run it like this:   
        ./check\_dirs.ksh both   
  
“both” is a command-line argument as input to the script, script refers to it as $1   
  
If I had a Java executable that took two input arguments:   
        ComputeDaysWorked.java   
                        start date   
                        end date   
  
On the command-line, it would look like this:   
        java ComputeDaysWorked.java “2015-01-01” “2016-01-12”   
  
In a Program Task, I would set the executable to be:   
        java ComputeDaysWorked.java   
  
I would set the Arguments properties to be:   
                “2015-01-01” “2016-01-12”   
  
In a Program Task, I could also do this as the Executable (using “Embed” option)   
        java ComputeDaysWorked.java “2015-01-01” “2016-01-12”   
  
So why have a separate Arguments property if you can just give the arguments as part of the Executable property?   
        Notice that each property in a Task has an Interpretation.   
        By having two separate properties (Executable/Script + Arguments) you can have   
        different Interpretations for each property.   
        The Script could be $-substitution, Arguments could be “constant”   
          
  
In my demonstration, I have a Program Task that uses Script   
        Script (constant, embedded)                $AI\_BIN/check\_dirs.ksh both   
  
And it worked! How can “constant” interpretation work if I am using a parameter reference $AI\_BIN?   
  
When you run a program task —> we open up a Korn shell and we send a command to the shell. What command you send depends on the value of the Script property + its Interpretation.   
  
Let’s say that the value of AI\_BIN is /home/ggx/ggx\_rbuchheit/ConductIt/bin   
  
        Script (constant, embedded)                $AI\_BIN/check\_dirs.ksh both   
        The value of the Script property is sent to the Korn shell as-is   
        The Korn shell sees: “$AI\_BIN/check\_dirs.ksh both”   
                and the Korn shell interprets the reference to $AI\_BIN   
  
        If I do this instead:   
        Script ($-substitution, embedded)                $AI\_BIN/check\_dirs.ksh both   
        Ab Initio (the plan) resolves the value of $AI\_BIN and sends this to the Korn shell:   
                /home/ggx/ggx\_rbuchheit/ConductIt/bin/check\_dirs.ksh both   
    
For a program task, the Interpretation of the Script/Executable value is what \*Ab Initio\* does to the value. The interpreted value from Ab Initio is then sent to the Korn shell to be executed.   
  
=============================================   
  
When you use a program as the Conditional Task   
        if a program runs successfully —> exit status in Korn shell of 0 —> TRUE   
        if the program fails —> nonzero exit status in Korn shell —> FALSE   
  
  
if I run   
        ls /home   
the command returns a list of the contents of the /home directory —> ignored by the conditional task   
the command has an exit status (echo $?) —> what the conditional task looks at   
  
  
We want to have a conditional task that checks if a file exists…   
        PROGRAM                        ls /path/to/this/file.xls   
        EXPRESSION                $[ file\_information(“/path/to/this/file.xls”).found ]   
  
                for the EXPRESSION in this example, I am using PDL interpretation   
                $[ ] are called “DML inline computation” — enables me to use DML functions to compute   
                a value —> using the Ab Initio file\_information() function to check if the file exists   
  
  
        Even better —> you can check the size of the file (empty, non empty, > 10000 bytes, etc.)   
        EXPRESSION        $[ file\_information(“/path/to/this/file.xls”).size > 10000 ]   
  
        In Version 3.2.something, we introduced DML functions that connect to a database!!!   
        You can use DML functions to check information in a database as a conditional task!   
  
        You are using V3.2.2, which does have these function   
  
        Functions versus command-line utilities   
                db\_get\_int()   
                db\_get\_datetime()   
                db\_get\_string()   
                —> these are all functions, they need to be run inside $[ ] using PDL interpretation   
  
                m\_db   
                m\_eval   
                —> these are all command-line utilities, they are programs (not expressions)   
  
  
        EXPRESSION   
        $[ db\_get\_datetime(AI\_DB + “/postgres.dbc”, “select max(update\_date) from customers”) > (today() - 1)]   
                —> condition is true only if there has been an update in this table since yesterday   
                —> using the function is much much easier   
  
  
        PROGRAM   
        m\_db unload $AI\_DB/postgres.dbc -select “select max(update\_date) from customers”   
                returns a date, in the Korn shell — I would need to write command to compare this                 value to yesterday’s date (= difficult)   
  
  
To do comparisons to data in the table —> using the functions in a DML expression is easier   
If you just want to check if the database is up + accepting connections —> using the command-line utility is better   
  
        PROGRAM   
        m\_db version $AI\_DB/postgres.dbc   
                (recommended function to use to test if a database is available)   
  
=========================   
  
On the Input Values tab for graph + plan tasks, you can override the value of any project parameter from private project that the graph/plan belongs to + any common projects that it includes — if the parameter is overridable.   
If you expand the project name on the Input Values tab, you get a list of overridable parameters + current value —> you can change the value of that project parameter for this task only.   
  
If I edit the properties of the “Move and Reformat Graph Task” and on the Input Values tab, I change AI\_MFS\_DEPTH to 4. Then any components in \*that\* job run 4-ways parallel if their layout is AI\_MFS.   
  
If I have second task that runs the same graph with different input parameter values, that task would run graph with components 2-ways parallel (default value for AI\_MFS\_DEPTH) for the AI\_MFS layouts.   
  
You can have the same graph/program/plan run multiple times (with different tasks) in the same plan. Obvious reason to do that — run with different parameter values for each task.   
  
===============================================   
  
Wait for File has two connection points   
        success — file arrives within the Timeout limit   
        failure — file doesn’t arrive within the Timeout limit   
  
(default Timeout limit is “forever” — you probably want to change this!)   
  
  
================================================   
  
Events that you create and wait for with   
        Signal Event   
        Wait for Event   
  
are just labeled signals. You can name them whatever you want.   
  
If I have a plan that has a “Signal Event” task with the event name “becky”   
then my other plan will have a “Wait for Event” task with the \*same\* event name “becky”   
This synchronizes those two plans — the plan with Wait for Event will wait at that point until the “becky” event is signaled (when the Signal Event task runs).   
  
        resource-admin signal        —> same as the Signal Event task (sets the signal)   
        resource-admin unsignal —> unsets the signal   
        resource-admin events —> lists all of the currently active/signaled events   
        resource-admin check-event <name> —> returns status of that event   
  
  
  
I have two plans:   
        plan1        —> runs a Signal Event to signal “becky”   
        plan2        —> runs a Wait for Event to wait for “becky”   
  
First I did this:   
        run plan2                —> nothing runs because “becky” event is not signaled yet   
                        WAITS   
        run plan1                —> as soon as the Signal Event task runs, then plan2 runs because Wait for   
                        Event found its event   
        now both plans have completed successfully   
        run plan 2 again        —> runs immediately because the “becky” event is still active/signaled   
                        NO WAIT   
  
  
If I had done this instead:   
        run plan1                —> runs Signal Event task, now the “becky” event is signaled   
        run plan2                —> runs immediately because the becky event has already been signaled   
                        NO WAIT   
  
  
What if I want to unsignal that event — deactivate it?   
                resource-admin unsignal becky   
  
Now when I run plan2 —> it waits until the becky event is re-signaled.   
  
resource-admin commands are just command-line programs…   
        so you can run them from Program Task   
        you can run them from a method   
        you can run them manually from the command-line   
  
  
If we never turn off the signal, will it be there always?   
        Yes, until the server is rebooted.   
        As long as the server stays up — the event is signaled.   
  
Yes, this does work across plans in different sandboxes.   
The only restriction is that both plans (the one that signals the event & the one that waits for the event) must be started on the same server.   
You can have multiple plans that wait for the same event. (It doesn’t have to be a one-to-one relationship.)   
You could have multiple plans that also signal the same event — but that would be weird and unlikely. I can’t think of many good reasons to do that.   
  
================================================

**DAY 2:**

Trigger method is different than all the other methods.   
All the other methods just run at the point in the task execution when they are supposed to run.   
Trigger method must be an expression or program that returns True / False —> just like a Conditional Task.   
        Expression could be a DML inline computation, like   
                $[ file\_information(AI\_SERIAL + “/trans.dat”).found ]   
                in which case, the DML expression must return   
                                True —> 1 or any non-zero number   
                                                        or T or t or any string that starts with T/t   
                                False —> 0 or F or f or any string that doesn’t start with   
                                                                T/t   
  
        Program/executable can be any Korn shell command or   
                executable   
                        True —> if the program/executable exits with 0 exit status   
                        False —> if the program/executable exits with non-zero   
                                        exit status (fails)   
  
For a Trigger method, you can configure how many time it retries and how often.   
We have “Wait for File” task component — a similar idea — Trigger method is waiting for some thing to occur (file arrives, database table is online, a certain value is loaded into the table, ….?) and then once it occurs, the task proceeds with executing the rest of its methods.   
  
=======================================   
  
Let’s say that you have a task that runs a graph and has the following methods defined:   
        AtTrigger — $[ file\_information(AI\_SERIAL + “/trans.dat”).found ]   
                                        RetryInterval                60 seconds   
                                        RetryNumber                10   
                                        Action after retry limit        Fail   
        AtStart — mv $AI\_SERIAL/trans.dat $AI\_SERIAL\_TEMP/trans\_${TODAY\_DATE}.dat   
                                        Task action after failure         Fail   
        Perform — run a graph that processes that transactions file   
                                        Task action after failure        Fail   
        AtFailure — graph that sends an email to the person responsible for this plan   
  
  
Example 1:   
When the plan starts running this task, if the trans.dat file is not found within 10 minutes (60 seconds \* 10 tries), then the AtTrigger method fails (“Action after retry limit” = Fail)   
The AtStart & Perform methods would not run —> task goes straight to AtFailure method   
  
Example 2:   
The trans.dat file is found within the retry limit. But then when the AtStart method runs, the “mv” command fails because the AI\_SERIAL\_TEMP directory isn’t found. Since the “action after failure” for the AtStart method is “fail” —> task goes straight to the AtFailure method (the Perform method doesn’t run).   
  
Example 3:   
Let’s say that we find the trans.dat file and successfully move it to $AI\_SERIAL\_TEMP but then the graph fails because the data isn’t sorted & it expected sorted data. Because Task action after failure is “Fail” —> AtFailure method would run.   
  
Whenever the failure occurs — assuming that the method uses the default behavior (task action after failure = “fail”) — task immediately moves forward (skipping over the other methods) to the AtFailure method if there is one.   
  
==========================================   
  
Continuous Flows graphs are specially designed graphs that run forever.   
Contrast this with a typical graph which is called a “batch graph”   
  
Batch graph starts up, reads the data from its data sources (files, tables, etc.), processes that data, writes the results to the targets, and then shuts down. Processes the data that is there when it starts & then when it’s done, it stops. No processes are running for this graph once it shuts down.   
Batch graphs are often scheduled to run at a particular time interval — once a month, once a week, once a day, once every hour, etc.   
  
Continuous Flows (CF) graphs are used when you want lower latency than what is possible with batch graphs. (“Lower latency” = targets are written very frequently — several times each second! or once a second or once a minute)   
CF graphs achieve this by never shutting down. Start up + shut down take time = increases the latency for getting results into the targets. By never shutting down, you don’t have to wait for the CF to start up every time you need to process more data.   
CF graphs are used as web services backends (to service requests from a web server), to process near-real time data, etc.   
CF graph is started up once, continuously polls (asks for) new data and it processes the new data as soon as it arrives.   
  
You can have a plan that runs a CF graph.   
When does the plan shut down? Never because it can’t shut down while the CF graph it still running.   
Why would you do this? You might have a situation where you need to run a few batch graphs to set up your system before the CF graph is ready to start. You might have scheduled maintenance on the CF graph & its system (sources + targets) so that you manually shut down the CF graph once a month. You could design a plan that runs the batch graphs to do the setup & then runs the CF graph.   
  
When you want to gracefully shut down a CF graph — as an example, for scheduled maintenance — you use a specific command “m\_shutdown” to do so. Stops the graph in a way that is consistent, enables it to restart without error, and is not an error (not failure).   
  
If you want to shut down the plan — again for this “scheduled monthly maintenance” in my example — on the command-line, you would run a command “plan-admin shutdown” (=gracefully shuts down the plan by stopping any tasks that are currently running). This runs the “AtShutdown” method for any tasks that are currently running.   
        —> requirement: if the plan runs a CF graph, then you should define an AtShutdown   
                        method for that Graph Task that runs “m\_shutdown” to gracefully shutdown the CF   
                        graph if the plan is stopped.   
  
==================================================   
  
If a task has its own methods + it inherits methods from a plan or subplan…   
        We work from the outside —> in on start   
        We work from the inside —> out on end   
  
Example order of execution for methods in a Task that has inherited methods   
1-        inherited AtStart from plan   
2-        inherited AtStart from subplan   
3-  own AtStart   
4-  Perform   
5-  own AtSuccess   
6-  inherited AtSuccess from the subplan   
7-  inherited AtSuccess from the plan   
  
(AtFailure acts like AtSuccess)   
  
=======================================================   
  
Typically in production — use default recovery behavior (“continue from point of failure”) from the command-line   
        — rollback any failed tasks if they haven’t rolled back yet   
        — continue the plan from the point of failure   
  
Typically in dev, when you are testing + designing a new plan, you just want to delete all the stuff & start over after a failure — default behavior in GDE is “remove job & then restart it”   
  
  
m\_rollback should be used for graphs only   
to rollback plans, you should use the command-line options for “air sandbox run”   
        check out the Help Library for “air sandbox run” command   
                —> plan-recovery-options: options for rolling back and deleting recovery files on restart   
                —> other-plan-options: include options to skip various tasks on restart   
  
  
Let’s say that you have a plan that runs a Graph Task. When this Graph Task fails, you actually get two recovery files:   
        recovery file for the Graph Task because it failed   
        recovery file for the Plan itself because it failed because one of its tasks failed   
  
What happens to the recovery file for the Graph Tasks depends on the design of the graph. When the Graph Task fails, Conduct>It tries to do an m\_rollback (by default) of the graph.   
  
If the graph fails in phase 0, m\_rollback rolls back that phase, deletes the recovery file. Now you have only 1 recovery file (for the plan itself) — that will be dealt with when you use “air sandbox run” options or restart the plan job.   
  
If the graph fails in phase 1 or later (and the phases are checkpointed, default behavior), m\_rollback only rolls back the phase that failed. Recovery file for the graph is retained. You still have two recovery files: one for the plan & one for the graph. Both of these recovery files will be dealt with when you use “air sandbox run” options (either to delete the recovery files) or restart the plan job.   
If you choose to “continue from the point of failure” in this case — graph task starts at the beginning of the phase that failed because it uses the recovery file information to skip over the phases that have already succeeded. If you don’t want that behavior (if you want to delete the recovery file & restart the graph from the beginning), you would want to edit the AtRollback method for this graph task to change that behavior.   
  
  
  
To restart a plan from the recovery file (default behavior, “continue from the point of failure”)   
All you need to do is just run the plan again “air sandbox run”   
  
        plan-admin commands have been deprecated (don’t use them) for running a plan   
        You should use “air sandbox run” to rollback, recover + run plans   
  
        You \*should\* use the plan-admin commands to \*stop\* a running plan   
                        various shutdown options   
                        various options to stop a task within a running plan   
                        various options to restart, skip, disable, etc. a task within a \*running\* plan   
  
============================================

**DAY 3:**

In our ex7 plan for connection points, we have a task (Create File) that runs this command:   
        touch $AI\_SERIAL/rec/rec.dat   
        exit 1   
  
It has an AtRollback method that runs this command:   
        rm -rf $AI\_SERIAL/rec/rec.dat   
  
Then we attached a Program Task to its failure connection point. That program task runs this command:   
        touch $AI\_SERIAL/rec/failure.dat   
  
  
The Create File task is going to fail (due to the exit 1 in its script). When I run this plan, what files do I expect to see in $AI\_SERIAL/rec???   
        failure.dat ONLY   
  
The rec.dat file was created in that directory but then it was deleted because the AtRollback method of the Create File task ran \*before\* the Program Task to create the failure.dat file. When a task fails and that task has its failure connection point enabled, the AtFailure & AtRollback methods run \*first\* before the task connected to the failure connection point.   
  
================================   
If you want to change the value of the PARAM1 parameter in tasks A, B, and C   
        then in each task, you must declare the parameter as dynamic and run the plan-admin set   
        command   
All three tasks must declare the parameter dynamic.   
  
When a task declares that a parameter is dynamic & changes the value of that parameter, ONLY tasks that are dependent on that task will see the new value. Tasks that are independent (not connected as successors) will continue to see the original value of the parameter.   
  
===============================   
  
We’ve seen conditions in Conduct>It already in a few different places:   
        — Conditional Task   
        — Trigger method   
        — While loop — “Loop condition”   
                        loop continues to run as long as the condition is True.   
                        at each iteration, we re-evaluate the condition & if it returns False, the loop stops   
  
Two kinds of conditions: Expressions + Programs   
  
        Expression is usually an Ab Initio DML expressions — going to run a DML function to compute   
        true or false   
                        True = 1 or non zero number   
                                        T or t   
                                        any string that starts with T or t   
                        False = 0   
                                                F or f   
                                        any string that does not start with T or t   
                If you are using a DML function or DML expression, it needs to be enclosed in $[ ]   
                $[ file\_information (AI\_SERIAL + “/input.dat”).found ]   
                $[ file\_information (AI\_SERIAL + “/input.dat”).size > 10000 ]   
  
        Program — can be any executable: Korn shell script, Korn shell command, Java program,   
                        C program, Ab Initio graph or plan, Python script, etc.   
                        True = program runs successfully (zero exit status)   
                        False = program fails (non zero exit status)   
  
  
=======================================================   
  
$[ not file\_information (AI\_SERIAL + “/stop.control”).found ]   
  
  
$[ ] is basically telling the graph/plan to evaluate the contents as a DML transform / expression   
        in this syntax, parameters such as $AI\_SERIAL, are treated like variables.   
  
  
Consider a reformat transform   
out :: reformat (in) =   
begin   
        let birthday = “2016-01-01”;        // declared a local variable   
        out :: date\_month((date(“YYYY-MM-DD”))birthday);   
end   
        NOTE: when we reference the value of the variable (birthday), we just write “birthday”   
        not $birthday   
  
When you use $[ ] syntax to compute a parameter value using DML expression — it acts just like a transform   
  
If I have a sandbox parameter   
Name                                                        value   
birthday                                                2016-01-01   
birth\_month                                        $[  date\_month((date(“YYYY-MM-DD”))birthday) ]   
  
NOTE: I use the exact same expression that I did in the transform to compute the birth month. Inside $[ ] parameters are treated like variables.   
  
Normally, when I want to reference AI\_SERIAL, I write:   
        $AI\_SERIAL/stop.control   
  
But when I’m writing an expression inside $[ ], I write:   
        $[ AI\_SERIAL + “/stop.control” ]   
  
  
Why don’t I do this:   
        $[ AI\_SERIAL/stop.control ]   
What does the DML engine think this means?   
        / —> division operator   
        stop.control —> assumes that you have a variable named “stop”, it’s a record that contains   
                        a field named “control”   
        complains that it cannot divide the value of AI\_SERIAL by the value of stop.control because   
                AI\_SERIAL is a string and “stop” is not a known variable / parameter value.   
  
  
stop.control is a hard coded constant value, string constant — must be enclosed in quotation marks   
        AI\_SERIAL + “/stop.control”   
the / in the string value is because the value of AI\_SERIAL doesn’t end with /   
if you leave the slash out, you get something like this:   
AI\_SERIAL + “stop.control”   
        /home/ggx/feb3/data/serial/BDS/BDS\_retailstop.control   
  
We need to add a slash in there in the string constant for the directory path…   
AI\_SERIAL + “/stop.control”   
        /home/ggx/feb3/data/serial/BDS/BDS\_retail/stop.control   
  
  
=============================   
  
For the previous exercise (While Loop), we moved the parameters (INPUT\_FILE, OUTPUT\_FILE) from the plan to the looping subplan.   
Why?   
We use INPUT\_FILE & OUTPUT\_FILE parameters to pass values to the Graph + Plan tasks that run inside the loop. Graph Task creates a new output file at the path given by the value of $OUTPUT\_FILE.   
  
If you are going to run this graph task in a loop — do you want it to write the same output file every iteration of the loop? Probably not.   
Probably you want a new output file each time the loop iterates and runs the Graph Task…   
        $AI\_SERIAL/trans\_1.dat   
        $AI\_SERIAL/trans\_2.dat   
        $AI\_SERIAL/trans\_3.dat   
        and so on   
  
We can do that with   
        $AB\_PLAN\_LOOP\_CURRENT\_VALUE   
        $AB\_PLAN\_LOOP\_INDEX   
  
These parameters are only defined (have known values) when the plan is running & only inside the looping subplan.   
We moved the parameters INPUT\_FILE and OUTPUT\_FILE into the looping subplan so that we can use the value of AB\_PLAN\_LOOP\_INDEX to compute the OUTPUT\_FILE name.   
  
For the value of the OUTPUT\_FILE parameter   
Instead of:   
        $AI\_SERIAL/trans.dat        —> same output file every time the loop runs   
we can do:   
        $AI\_SERIAL/trans\_$AB\_PLAN\_LOOP\_INDEX.dat —> each time the loop runs, the output   
                        filename is slightly different (numbered)   
  
This is only possible because we moved the OUTPUT\_FILE parameter into the subplan. If we had left OUTPUT\_FILE as plan-level parameter, it wouldn’t work because the plan cannot evaluate AB\_PLAN\_LOOP\_INDEX. Only the looping subplan can see values for that parameter.   
  
=====================================   
  
For loops & ForEach loops   
        can run either sequentially or concurrently   
  
LoopConcurrent = False        (“sequential”)   
LoopConcurrent = True          (“concurrent”)   
  
When LoopConcurrent = False (sequential)   
        previous iteration must finish successfully before the next iteration begins   
        if iteration 5 fails —> then iteration 6 never starts   
        if we want to get to iteration 6 —> then iterations 5, 4, 3, 2, 1, and 0 must have completed   
                successfully   
  
When LoopConcurrent = True (concurrent)   
        all the iterations start up at the same time   
        if iteration 5 fails —> it does not affect any of the other iterations, they continue to run!   
  
  
Advantage of concurrent looping —> faster by running the iterations in parallel   
Drawback —> your server needs to have enough resources available to run all those iterations at once (fix: use resources — more on that tomorrow)   
—> the tasks inside the looping subplan must be designed such that they can run concurrently — they can’t write to the same files at the same time. Depending on how database access is set up (doing updates locks tables, etc.), you may need to be careful there as well.   
  
Advantage of sequential looping —> simpler because you don’t have to worry about adverse interactions between tasks running in each iteration   
Also helpful when the iterations are dependent on each other — iteration 0 creates the input for iteration 1, etc.   
Disadvantage: slower because you must wait for the previous iteration to finish before the next one can begin.   
  
  
Why can’t you run While loops concurrently?   
(They must always be sequential.)   
Because the condition needs to be reevaluated for each iteration — there is no way to know ahead of time how many iterations need to run. There’s no way to know how many iterations to start up at the beginning. Number of iterations is determined as the while loop runs.   
  
With For & ForEach loops, the number of iterations is computed before the loop starts running. Conduct>It knows how many iterations need to start up at the beginning.   
  
  
========================================   
  
directory\_listing() is a DML function that returns a list of files in a directory   
  
directory\_listing(AI\_SERIAL, “\*.dat”)   
—> returns a \*vector\* that contains a list of all the files that end with .dat in the $AI\_SERIAL directory   
  
Frequently used in ForEach loops in plans.   
Common use case for looping is that you want to run a (generic) graph once for each file in a directory. Idea is that the graph processes that file as input and reformats it, filters it, sorts it, writes it to a table — that sort of thing. You want to process all the files in the directory in this way — write a generic graph that has input parameters so that you can specify the input file that the graph uses.   
  
Graph Parameters   
        INPUT\_FILE           
        INPUT\_DML   
        OUTPUT\_TABLE   
  
Then in your plan, you put a Graph Task inside a Subplan.   
The Subplan is set to loop over all the files in a directory   
        LoopType                                        ForEach   
        LoopValueVector                $[ directory\_listing(AI\_SERIAL + “/files-to-process”, “\*.dat”) ]   
  
In the Input Values for the Graph Task   
        INPUT\_FILE                        $AI\_SERIAL/files-to-process/$AB\_PLAN\_LOOP\_CURRENT\_VALUE   
  
The parameter AB\_PLAN\_LOOP\_CURRENT\_VALUE returns the current value of the vector element that the loop is processing.   
If the directory listing function returns a vector such as:   
        [“transactions.dat”, “customers.dat”, “stores.dat” ]   
then on loop iteration 0, the value of AB\_PLAN\_LOOP\_CURRENT\_VALUE is “transactions.dat”   
on loop iteration 1, the value of AB\_PLAN\_LOOP\_CURRENT\_VALUE is “customers.dat”   
  
==================================================

**DAY 4:**

While Loops — have an unknown number of iterations   
                                        runs until the loop condition evaluates to false   
                                        quite common to have loop conditions that check for   
                                                existence of a file, etc.   
  
For Loops — has a known number of iterations   
                                number can be supplied as a hard coded value, as a parameter   
                                        value, or computed with an expression   
                                because the number of iterations is known, you can run the   
                                        iterations either sequentially or concurrently   
  
ForEach Loops — has a known number of input values (LoopValueVector)   
                                        values are supplied as a vector   
                                        values are either entered manually by hand (List)   
                                                or computed with an expression that returns a vector   
                                                value (Expression)   
                                because the number of iterations is known, you can run the   
                                        iterations either sequentially or concurrently   
  
Loop behavior is configured in Subplans   
        — put all the tasks that you want to loop inside the subplan   
        — configure the looping properties of the subplan   
        — inside the subplan \*only\*, you have access to two parameter values   
                        $AB\_PLAN\_LOOP\_INDEX        — number of the current iteration   
                        $AB\_PLAN\_LOOP\_CURRENT\_VALUE — value of the iteration (ForEach loops)   
  
  
  
If I have a task named “Run XYZ Graph” inside a subplan named “ABC Tasks”, then the full name of the graph task is:   
        /ABC Tasks/Run XYZ Graph   
  
The initial / refers to the top-level plan   
        then the subplan name   
                then the task name   
  
================================   
  
  
What do we mean by “logical” resources?   
You — as the developer or capacity planner — decide what resources you want to define and how many units of that resource are available.   
  
It has ABSOLUTELY nothing to do with your actual hardware.   
  
If your server has 6 CPUs, you can define a resource named “CPUS” and give it the value 500.   
  
If your server has 500 GB of memory, you can define a resource named “memory” and give it the value “100”.   
  
You can name the resources anything you want, they can refer to anything you want, and have any number of units that you want.   
You also decide how many units of that resource each task requires.   
  
If you create a resource named “memory”, ConductIt doesn’t check how much memory is on your server or whether the memory usage is high/low. It doesn’t check the \*physical\* resource at all. The only thing ConductIt knows is:   
        — how many units of the resource \*you\* say are available   
        — how many units of the resource each task requires to run.   
  
  
Your definition of the resources   
        — names of the resources   
        — number of units available for each resource   
is saved in a .pool file (class.pool, for example) in your sandbox. This definition can be promoted with the project to test/production servers.   
  
To actually use a resource pool, it must be activated on the server   
Use   
        resource-admin   
commands to administer resources on a server.   
You can resize resources (add or remove units) from the command-line   
        — add more units if a large queue of waiting tasks is forming so that more tasks are   
                able to run   
        — subtract some units if the server is starting to be overloaded & you want to   
                run fewer concurrent tasks   
  
  
By default, tasks do not use \*any\* resources.   
In order for a task to use a resource, you must configure the Resource tab on that task.   
If a particular resources is used 100% (no resources available), then tasks that use that resource (Resource tab is configured to use some number of units of that resource) and haven’t started yet will \*wait\* until the resource becomes available.   
        In the Tracking Details, we saw an orange hourglass indicating that some of the loop   
        iterations were waiting for Pencil resources to become available. Those iterations   
        were not running yet — they were waiting for the resource.   
        Waiting for the resource is not a failure; the task is simply waiting and will run   
                once enough units of that resource are freed up for that task.   
  
  
Let’s say that I have three plans.   
Plan 1 has a graph task that loads data into a database table 8-ways parallel (requires 8 database connections).   
Plan 2 has a graph task that loads data into a database table 4-ways parallel (requires 4 database connections).   
Plan 3 has a graph task that loads data into a database table 16-ways parallel (requires 16 database connections).   
Each degree of parallelism = 1 database connection   
Our project is only allowed to use 18 database connections at the same time.   
  
Step 1:   
I create a resource pool for our project. And I define a resource named “Oracle DB Connections” and I declare that 18 units of this resource are available.   
  
Step 2:   
I edit each of the graph tasks in my 3 plans.   
        Plan 1 —> Graph Task uses 8 units of “Oracle DB Connections” resource   
        Plan 2 —> Graph Task uses 4 units of “Oracle DB Connections” resource   
        Plan 3 —> Graph Task uses 16 units of “Oracle DB Connections” resource   
  
Step 3:   
I start all three plans at the same time…   
        air sandbox run plan1.plan   
        air sandbox run plan2.plan   
        air sandbox run plan3.plan   
All three plans start execution. Any other tasks in those plans begin to run.   
Graph Task in Plan 1 asks for 8 units of the “Oracle DB Connections” resource and it gets them because 18 units are available.   
Now 10 units of “Oracle DB Connections” resource are available.   
Graph Task in Plan 2 asks for 4 units of “Oracle DB Connections” resource and it gets them because 10 units are available.   
Now 6 units of “Oracle DB Connections” resource are available.   
Graph Task in Plan 3 asks for 16 units of “Oracle DB Connections” resource and IT DOESN”T GET THEM because they’re not available.   
  
All the tasks in Plan 1 are running.   
All the tasks in Plan 2 are running.   
All the tasks in Plan 3 are running EXCEPT the Graph Task which is waiting for “Oracle DB Connections” resource (and of course, also any tasks that are dependent on the Graph Task will wait because they must wait until this graph task completes)   
  
Let’s say that the graph task in Plan 2 finishes first. That frees up 4 units of “Oracle DB Connections” resource. We had 6 units available + 4 more that are now freed up. Now there are 10 units of “Oracle DB Connections” resource available.   
  
But the Graph Task in Plan 3 requires \*16\* units of the resource, so it is still waiting.   
  
Now the graph task in Plan 1 finishes. That frees up 8 units of “Oracle DB Connections” resource. We had 10 units available + 8 more that are now freed up. Now there are 18 units of “Oracle DB Connections” resource available — now the Graph Task in Plan 3 can finally run. It acquires 16 units of the resource, there are two leftover (18 - 16 = 2).   
  
===============================   
  
Let’s say that you have a task A that uses 4 units of a resource.   
You have two other tasks B and C that are waiting for that resource.   
When Task A finishes, which task (B or C) gets to run?   
Depends on the design of the plan that they’re in. Tasks are typically added to the queue of waiting tasks in the alphabetical order of their names if they’re in the same plan and there are no dependencies. If they’re in different plans, then the plan that started first gets precedence.   
The important thing here is: it shouldn’t matter. It should not matter to you whether B or C runs first.   
If it does matter to you whether B or C runs first, you can give one of them a higher priority.   
If it matters to you whether B or C runs first because C depends on B being first, then having these two tasks run concurrently is incorrect plan design. You should have a dependency between them  B —> C   
If you just care because B is more important than C, you can configure a Priority on the Resources tab.   
  
==================================   
  
Use resource pools only if   
        — you have many concurrent tasks running on your system   
                either in the same plan or over multiple plans   
        — you need to throttle (limit) how many concurrent tasks run at the same time   
                to avoid overloading your server   
  
You would need to identify which physical resources on your server are limited (memory, db connections, etc.), define logical resources for those limitations, and then identify which tasks run concurrently & use those resources (define resource usage for those tasks).   
  
Typically resource pools are for all the plans in a project, or even all the plans in a group of related projects.   
You typically do not have one resource pool per plan.   
Pools are typically defined at the project or even environment level (e.g., you have one resource pool for your entire server).   
  
=====================================   
  
Manual for Conduct>It is very thorough.   
Help Library   
        Co>Operating System Help >   
                                Conduct>It Guide & Reference   
  
        Highly suggest that you take a look at the “Conduct>It Recipes” section of the manual.   
  
=====================================   
  
4 categories of LOOKUP FILE behavior:   
— “static” lookup   
                typical lookup, LOOKUP FILE + lookup()   
                “static” — data doesn’t change while the graph is running   
                                graph reads the data from the lookup file at the start of the phase that   
                                uses it, organizes the data in memory & then it accesses that data from   
                                components in the graph   
  
                lookup() & lookup\_first() have the same behavior   
                TRUE for recent (3.0 and later versions of the Co>Operating System)   
                In earlier versions before 3.0, the lookup() function was allowed to return   
                        any matching record from the lookup   
                It’s still good practice either way to use lookup\_first() when you know that you   
                        have multiple matches & you want the first one because it \*documents\*   
                        that requirement.   
  
                exact match —> data in the flow is matched to key values in the lookup exactly   
                                                        123 —> 123   
                interval —> data in the flow is matched to a record that contains a min+max value   
                                        (range) in the lookup   
                                                        123 —> between 100 and 200   
                regex —> data in the flow is matched to a regular expression pattern in the lookup   
                                                        123 —> \d\*   
  
— “dynamic” lookup   
                LOOKUP FILE TEMPLATE, lookup\_load() and lookup\_unload() to load the   
                        lookup file into memory   
                gives you control over when the lookup file is accessed   
                it can be loaded & unloaded   
  
                “basic” — you the graph developer are responsible for reloading the data if   
                        it changes and you want to pick up the updates   
                        typical use case: you have a lookup file that is too big to fit into memory   
                                instead of loading the entire lookup file into memory at once,   
                                you break up the lookup file into pieces & you load only the piece that   
                                you currently need   
                                typically used with sorted data   
                                        maybe you have a lookup for all customers who live in “NY”   
                                        then you have another lookup file for all customers who live in “MA”   
                                you sort the input data in your graph by {state}, then in your transform   
                                        you load the “MA” lookup file with lookup\_load(), lookup the information   
                                        for all those people & then when you get to the first person who lives in   
                                        “NY”, the transform is designed to unload the “MA” lookup and load the   
                                        “NY” lookup   
                                (not used very often any more — most people use ICFFs now instead   
                                reason: this often results in extremely complicated transform logic)   
  
                “ALF” — Appendable Lookup File (some people call these ALUs — appendable   
                                lookups)   
                                you have two graphs — one graph that adds data to the lookup file   
                                        another graph that reads data from the lookup   
                                you want the reading graph to automatically see if new records have been   
                                        added to the lookup & use them if so   
                                (typical use case: you want to access a lookup file from a Continuous Flows   
                                graph — graph that is always running. You don’t want to have to stop the   
                                CF graph every time you need to update the lookup file data. Enables the   
                                CF graph to keep running and you can have some other graph add new   
                                data to the lookup file whenever it arrives. CF graph will automatically   
                                see the new data the next time it runs the lookup() function.)   
  
— “created” lookup   
                lookup that is created by you in a transform   
                you use input records + lookup\_add() function to add records to the lookup   
                uses LOOKUP TEMPLATE + lookup\_add() + lookup()   
  
                — standard “created” lookup — which is created + used in the same transform   
                — “updatable” lookup — which is created in a transform then saved to a file on disk so   
                                that it can be used as a lookup by other components + graphs.   
  
                Equivalent of creating a “hash table” or “HashMap” in other programming languages.   
  
— ICFF (“Index compressed flat file”)   
                way to use extremely large lookup files   
                various options for having either static or dynamic (appendable) ICFFs   
  
                basic idea:   
                — you take extremely large source data and split it into smaller files that are then compressed   
                        and saved on disk (WRITE BLOCK COMPRESSED LOOKUP)   
                — this also creates a very specialized index that indicates which keys are in the lookup &   
                        which file contains the key   
                — when you run a graph that uses an ICFF, the graph automatically uncompresses the matching   
                        file + loads it into memory + finds the matching record for that key   
          
                Lot of design decisions that have to be made about how to organize the data in the files,   
                how to create the index so that it is effective but not too big, how to organize the data records   
                that flow through the graph so that you aren’t opening + uncompressed a new lookup for each   
                input record, and so on.   
  
                If you have appendable ICFFS, you also need to have infrastructure in place (additional graphs   
                + scheduled jobs) to consolidate & reindex the new records periodically   
  
  
Co>Operating System   
        > Graph Development Guide   
                > Lookup Data Processing Guide & Reference   
  
Example graphs that I showed in class will be located at   
        /home/ggx/feb1   
  
Copy these graphs into your home directory in the graph-dev-basics sandbox   
        /home/ggx/<your id>/examples/graph-dev-basics/mp   
  
You’ll be able to run them from there.   
  
========================   
  
          
Interval lookup — typically has two key fields   
        (there is a way to set it up with one key field, but it is more difficult)   
  
the key fields define a range of values that indicate an output value   
                NY                        10003                14853   
                        implies that if a zip code is between 10003 and 14853, then the state is NY   
  
In the Key Specifier for the LOOKUP FILE component, you have two key fields   
        interval bottom (field that indicates the starting/min value for the range)   
        interval top (field that indicates the ending/max value for the range)   
  
You can have non-overlapping intervals (in which case the data in the lookup file needs to be sorted on the interval bottom field) or you can have overlapping intervals (slower, but it means that the data doesn’t have to be sorted).   
  
Non-overlapping intervals (data must be sorted by interval bottom field)   
Lookup data might look like this:   
        min\_percent                max\_percent                        grade   
        0                                        50                                                F   
        51                                        60                                                D   
        61                                        70                                                C   
        71                                        80                                                B   
        81                                        100                                        A   
Notice that each interval is distinct. A particular value can match only 1 interval.   
  
Overlapping intervals (data doesn’t need to be sorted)   
Lookup data might look like this:   
        min\_postal\_code                max\_postal\_code        state   
        20000                                                30000                                NJ   
        25000                                                35000                                PA   
        10000                                                25000                                NY   
If I have an input value 24875, then it matches both “NJ” and “NY”.   
A particular value might match 1 or more intervals. (If that’s the case, remember you can use lookup\_count() & lookup\_next() to get all the matches, or functions such as lookup\_first() to return a specific match.)   
  
If you have overlapping intervals or unsorted lookup data, you would specify two key fields   
        interval bottom & overlapping                — min value of the interval   
        interval top                                                        — max value of the interval   
  
For Interval Lookups — the only difference between configuring them versus exact lookup is when you configure the lookup key.   
The functions that you use to access the lookup file (lookup(), lookup\_next(), etc.) are exactly the same — whether you are doing interval lookup or exact lookup.   
  
  
=========================   
  
Regex lookup — key values contain regular expressions   
        For example   
                        .\*\.xls(x)?   
  
        .\* —> means zero or more characters of any kind (. = any character) (\* = zero or more)   
        \. —> means a literal .  (actual period character, decimal point)   
        xls —> a literal “xls”   
        (x)? —> zero or one “x” characters ((x) means a literal “x”) (? = zero or one)   
  
        This regular expression matches values such as:   
                        abc.xls   
                        abc.xlsx   
                        .xls   
                        My Big Spreadsheet.xls   
                        My Even Bigger Spreadsheet.xlsx   
  
        It does not match   
                        abc.xls1 (because it can’t end with 1)   
                        abcxls (because it must have a “.” before xls)   
  
        You define a regex LOOKUP by setting up the Key Specifier to indicate that the   
                key is a “regex” key.   
        The lookup() function that you use in a transform is the same as what you’d use for   
                an exact lookup. The lookup() function can return all the fields from the matching   
                record or just one field from the matching record — just like we do in an exact   
                lookup.   
  
=============================   
  
LOOKUP TEMPLATE — consists of a record format + key + configuration parameters for the lookup behavior.   
It does \*not\* specify the location of the lookup file.   
  
Multiple different files in a transform can share the same LOOKUP TEMPLATE as long as they all share the same record format + key.   
  
I can do this:   
        lu1 = lookup\_load($[AI\_SERIAL + “/lookup\_A.dat”], NULL, “Value Lookup Template”, -2);   
        lu2 = lookup\_load($[AI\_SERIAL + “/lookup\_B.dat”], NULL, “Value Lookup Template”, -2);   
  
Dynamic lookups (either basic or ALFs) have a few requirements:   
        — you need to declare a lookup\_identifer\_type variable   
        — you need to use lookup\_load() to load the lookup file into memory   
                        result of lookup\_load() function should be assigned to the lookup\_identifier\_type variable   
        — you need to use a LOOKUP TEMPLATE instead of LOOKUP FILE   
        — lookup() function is different   
                        the first argument is the lookup\_identifer\_type variable   
                        second argument is path to the lookup index (if you have one, or NULL if not)   
                        third argument is the label of the LOOKUP TEMPLATE component   
                        fourth argument is the “loading behavior”   
                                        0 = basic dynamic lookup   
                                        -2 = ALF   
  
What is a lookup index?   
Automatically created for you when you use static lookups.   
        A table of key values + file positions so that the lookup() function can quickly:   
                        — determine if a particular key value is in the lookup file data   
                        — if yes, find where that record is located in memory   
  
It doesn’t make sense to use it for ALFs   
        — because you are adding new data all the time   
  
You can create your own index & save it in a file to use with the lookup() function when you do have a static lookup.   
        WRITE LOOKUP   
Input (lookup file data) —> WRITE LOOKUP —> index file