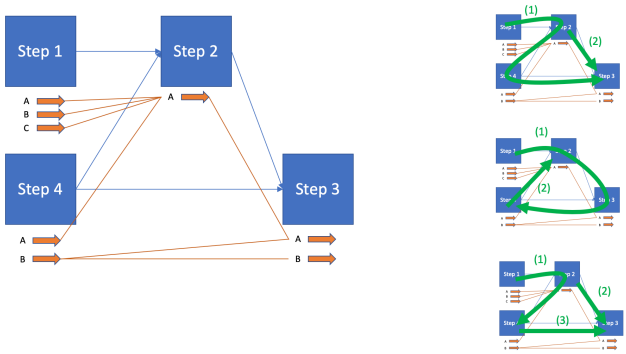
Experiment Table Assembly Algorithm

28 September 2019

Tim Gardner, Sandeep Lanka



Above is an example process table (blue) with an embedded run graph (orange). The assembly algorithm should produce the same result by joining connected step data tables regardless of the order (green) of visiting step connections.

Objective: order-independent joining of step data tables into an aggregate experiment data table while satisfying both run connections and property joining rules for any process topology

Requirements for final experiment data table

1. Every run (and multivariate data rows) shall be joined with all of its connected runs in one or more rows
2. All runs (and multivalue data rows) are included in final data table, regardless of whether they are joined to other runs or not
3. Any row must satisfy all run connections and all run joining rules for any non-missing steps/columns in that row. Joining rules do not need to be satisfied for missing steps/columns in a given row.
4. Runs (and multivalue rows) that do not satisfy joining rules prescribed for them should STILL be joined to runs on any steps that are not involved in the joining rules. This leads to the need for a "modified left join" and a two part core joining algorithm as described below.
5. No rows should be duplicated.
6. All step connections should be visited once in the process to join the runs connected between them. Note that this means that cycles in the graph will be closed, but recursively joined runs will not get "unwound". Recursively joined runs form a pattern where upstream runs connect to downstream runs which connect to subsequent upstream runs in a zigzag pattern.
7. Where there are cycles in the graph, any run that is joined into the cycle will be joined to ALL matching runs in that cycle. This is illustrated in graphs 1 and 2 of the Finishing Algorithm below.

Example experiments and joins needed here to illustrate the application of above requirements

Commented [TG1]: Need to revisit this: I propose to insert "or to runs on included steps that are stripped of their joining column data and deduplicated," and update the algorithm accordingly. But this needs discussion and a think through the use cases, because I'm not entire sure yet what we want.

Main loop of algorithm – Graph walking

- The graph walking algorithm should give the same result regardless of the order in which steps are visited as the graph is walked.
- A list of all step to step connections is constructed. This list is the "incomplete connections" list. The connections can be added to the list in any random order.
- Each connection in the "incomplete connections" list is visited one by one. When a connection is visited, the data tables (the step tables) two connected steps are joined to create an output table. The output table is called the "**seed**" table.
    - These joins, in principle, should be made with standard outer joins between each pair of step tables. But a modification of this standard out join is required to address "fuzzy" (approximate value match) joining rules on the properties of runs and resources, and/or cycles that can occur in the run graph or in the composite graph of run + joining rules. The step table joining algorithms with these modifications are described below as the Core Algorithm and Finishing Algorithm.
    - Runs between two steps are matched using the run connections object, which is stored on each run of the downstream step in a connected pair of steps. The run connection object is read into an adjacency table with the run ids of upstream steps in the left column and the run ids of the downstream steps in the right column. When joining runs, the upstream runs are first joined to this adjacency table, and then the result of this join is joined to the downstream runs.
- Run connections are visited by selecting an initial step at random and then walking the graph forward and backward in a **breadth first search** of step connections. It is critical that a breadth first search is used to ensure that part 2 of the Core Algorithm functions properly. Graph walking continues until no more connections can be added.

Inner Logic of the Main loop – Step table joining

Definition: The "seed" data table is the aggregate table containing all previously joined step tables.

Implication of the experiment data table requirements stated above

The simple joining of two steps with a standard outer join is inadequate to address all possible run connection configurations due to the presence of cycles in the joining graph. Cycles can be formed by run connections or by combinations for run connections and joining rules between steps. Such cycles demand that each join accounts for information that may be added in future joins. However, the basic algorithm above is backward looking (causal), not future looking (non-causal). Thus, two joining algorithm modifications are made to correct for this issue and allow for use of backward-looking methods. The two modifications are called the "CORE ALGORITHM" and the "FINISHING ALGORITHM". The Core Algorithm is applied between any two step joins. The Finishing Algorithm is applied only after all step data tables have been consumed into the seed table and additional connections remain to be addressed.

1. **CORE ALGORITHM** It is necessary to join each new step data table with the following algorithm that satisfies the joining requirements and accounts for "cycles" in the graph composed of the run graph + joining rules as illustrated in the example below. This **core algorithm for step joining** is akin to an outer join. It produces an output table that is the union of a two-part join of the new step with the "seed" data table. The two joins are a "modified left join" and a right join

of unjoined rows of the new step. **Note that it is assumed that the seed table (or a subset of it) is always the left table and the new step (or a subset of it) is always the right table in all joins.**

1. Modified left join of the "seed" data table with the new step while satisfying all run connections and joining rules.
   - Definition of the "modified left join". First an inner join of tables is performed in Core part 1. The unjoined rows from that inner join are split into two rows types – those that have data from the steps involved in join rules (type 1 rows), and those that do not have data from the join-rule steps (type 2 rows). The type 1 rows are appended to the result table of the inner join. The type 2 rows are set aside and then joined in as the final step after Core part 2, only if those rows are not already picked-up in the right join of Core part 2 (below). The rationale for this is a bit complicated:
     1. 'Unjoined' rows can't actually be determined until both parts 1 and 2 of the core algorithm is complete. Unjoined rows are those that are not captured by inner joins in either part 1 or part 2.
     2. To address this issue, you need to set-aside the unjoined rows from part 1 for analysis and inclusion if they are not already captured in part 2. This works for type 2 rows (which have no data on the steps involved in joining rules), but not for type 1 rows. Type 1 rows have data on the joining-rule steps that cannot be captured in Part 2 (where joining rule steps are excluded). Thus if those type 1 rows get captured in Part 2, they will have different information than those same rows which came from Part 1. Thus type 1 rows must be included from parts 1 *even* if they are also captured in part 2 (because they will differ by the data in the joining-rule steps and thus are not duplicate rows).
     3. Thus Type 1 rows are immediately appended to the result of core part 1, so that they are not lost. Those same rows may also be joined via part 2 but without the joining-rule step data on them.
2. Right join of (1) a subset of the seed data table that excludes the steps involved in the joining rules with (2) the remaining rows of the new step that were not joined in phase 1.

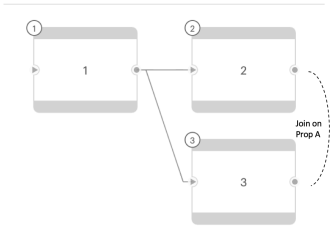
Commented [TG2]: Except if that step is one of the connected steps in the join

In the toy illustrations below, the modified left join is not actually needed (and is illustrated as a left join for simplicity). But in the actual implementation, the modified left join should always be used. If it is not needed, then unjoined type 1 or type 2 rows will simple be null and have no impact.

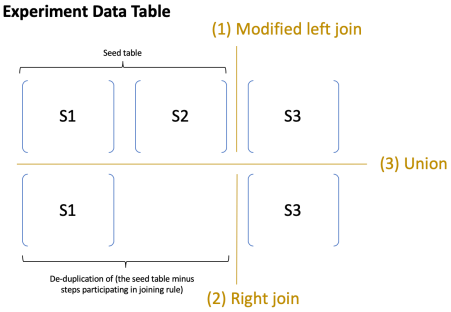
CORE ALGORITHM FOR STEP JOINING

1) Modified left join of 'seed' table with new step
2) Right join of unjoined runs of new step with a de-duplication of (the seed table minus steps participating in joining rule)
3) Union of results of 1 and 2 and unjoined type 2 runs from the modified left join

Process

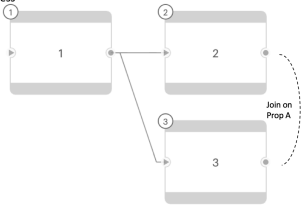


Experiment Data Table



EXAMPLE

Process



Runs on Experiment

S1

Run
A

S2

Run	Prop A
B	0
C	1

S3

Run	Prop A
D	1
E	2
F	3

(no cxn)

Cxn 1: Join S1 to S2

Modified Left Join S1 to S2 (run cxn) => TABLE 1

S1 Run	S2 Run	S2 Prop A
A	B	0
A	B	1

CORE ALGORITHM

(1) **Modified Left Join TABLE 1 to S3 (run cxn + join rule) => TABLE 2A**

S1 Run	S2 Run	S2 Prop A	S3 Run	S3 Prop A
A	B	0		
A	B	1	D	1

(2) **Duplicate TABLE 1, and remove S2 (because S2-S3 is part of join rule) => TABLE 2B**

S1 Run
A
A

Cxn 2: Join S1-S2 to S3

De-duplicate TABLE 2B => TABLE 2C

S1 Run
A

Unjoined runs from S3 => TABLE 2D

S3 Run	S3 Prop A
E	2
F	3

Right join of TABLE 2C with TABLE 2D (run cxn) => TABLE 2E

S1 Run	S3 Run	S3 Prop A
A	E	2
	F	3

(3) **Union of TABLE 2A with TABLE 2E => FINAL TABLE**

S1 Run	S2 Run	S2 Prop A	S3 Run	S3 Prop A
A	B	0		
A	B	1		
			D	1
			E	2
A			F	3

2. **FINISHING ALGORITHM** In addition to cycles formed by run connections + joining rules, it is possible for runs alone (without any joining rules) to be connected in cycles. These cycles present difficulties for step joining when each step is joined for the first time because that first join cannot satisfy both of its step connections (because one or more of the connected steps have not yet been added to the seed table). Thus, there is missing information in this first join. This information could lead to incomplete run joins if not addressed with an additional correction to the core algorithm. The basic procedure is:

    1. Apply the core algorithm to all connections in the process until all steps have been joined into the seed table
    2. Any remaining (un-addressed step connection) must now be performed via self-joins of the seed table (since any remaining connections are between steps already in the table). An altered core algorithm should be applied in these self joins.
    3. Self joins will lead to duplicated columns and rows. Thus rows must be de-duplicated and like-named columns must be merged.
    4. Unjoined rows must be added at the end

Commented [TG3]: May not need the core algorithm. It currently doesn't use it – it just does a standard join on run connections only – because those rules have already been addressed via the prior joining in of all steps. But need to think through this a bit more – maybe it really does need the core algorithm to avoid incorrect row joins.

FINISHING ALGORITHM FOR STEP JOINING WHEN BOTH TABLES ARE ALREADY IN THE SEED

Joining mechanism

- Null values and blank values are considered NON-matching rows in joins
- Each inner join (below) should be executed with a modified core algorithm where the left and outer joins in core algorithm are replaced with inner joins. Note the current algorithm just uses an inner join on runs – not the core algorithm. Not clear it is needed to invoke the joining rules here or the core algorithm (this needs further thought)

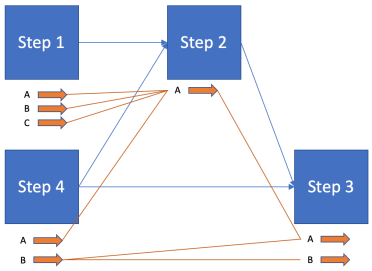
Column merge rules

- Values from any table always overwrite nulls in column merge
- Right table (the 'with' table) values will overwrite left table (the 'main' table) values in column merge

Join method

1. Duplicate the seed table (new table called dup_seed)
2. Temp table 1 = Inner join seed with dup_seed, join on upstream run ID (seed) = downstream run ID (dup_seed)
    - merge columns
    - keep track of original table indexes for the rows that were joined
3. Temp table 2 = Inner join seed with dup_seed, join on downstream run ID (seed) = upstream run ID (dup_seed), merge columns
    - merge columns
    - keep track of original table indexes for the rows that were joined
4. Temp table 3 = Union, deduplicate rows
5. Final table = Add back any unjoined rows (according to the row indexes of the original tables)

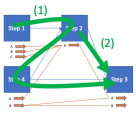
Graph 1



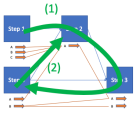
Expected result for any joining order

	1Run	2Run	4Run	3Run
1	1A	2A	4A	3A
2	1B	2A	4A	3A
3	1C	2A	4A	3A
4	1A	2A	4B	3A
5	1B	2A	4B	3A
6	1C	2A	4B	3A

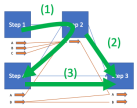
Test Result



PASSED

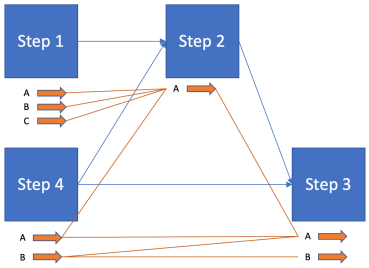


PASSED



PASSED

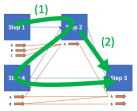
Graph 2 (adds extra connection A-A between steps 4 and 3)



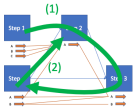
Expected result for any joining order

	1Run	2Run	4Run	3Run
1	1A	2A	4A	3A
2	1B	2A	4A	3A
3	1C	2A	4A	3A
4			4B	3B
5	1A	2A	4B	3A
6	1B	2A	4B	3A
7	1C	2A	4B	3A

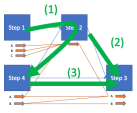
Test Result



PASSED



PASSED



PASSED

Illustrated Example

A detailed step by step application of this algorithm is illustrated starting on page 6 of this associated file titled "**Experiment Table Assembly Algorithm example**".