hail MatrixTables

https://hail.is/docs/0.2/hail.MatrixTable.html

MatrixTables vs Tables

A MatrixTable is a Table with an extra dimension. Tables have row fields and globals, whereas MatrixTables have row fields, column fields, entry fields, and globals. Many methods on tables have three equivalents on MatrixTables. For instance, **filter** on a Table has equivalents **filter_rows, filter_columns,** and **filter_entries** on a MatrixTable.

Globals

Global fields represent information constant across all entries.

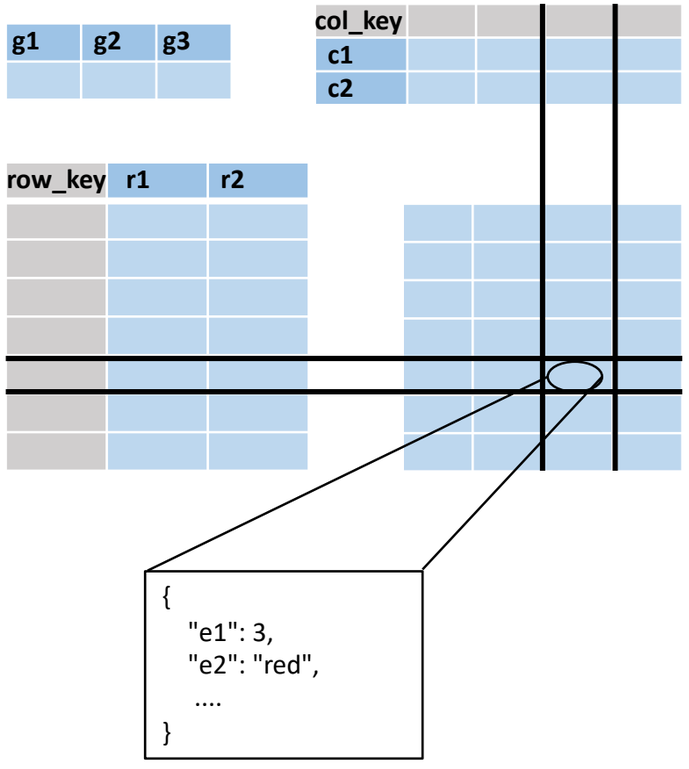
`mt.globals`
Get a struct containing global fields.

`mt.globals_table()`
Get the global fields as a single row table

`mt.annotate_globals(g4=2*mt.g2)`
Add new global fields.

`mt.transmute_globals(g2_sq=2*mt.g2)`
Like annotate_globals, but deletes referenced fields.

`mt.select_globals(mt.g1, g4="foo")`
Select existing or create new global fields, dropping the rest.



Columns

Column fields represent information constant across an entire column of entries.

`mt.cols()`
Get just the column fields as a table

`mt.annotate_cols(cf3=mt.cf1**2)`
Add new column fields.

`mt.transmute_cols(cf1_half = mt.cf1 / 2)`
Like annotate_columns, but deletes referenced fields.

`mt.filter_cols(hl.is_defined(mt.r1))`
Filters out columns/entries for which given expression is false.

`mt.select_cols(mt.c2, sum=mt.c2+mt.c1)`
Select existing or create new col fields, dropping the rest.

`mt.sample_cols(p)`
Randomly downsample columns by keeping each column with probability **p**.

Rows

Row fields represent information constant across an entire row of entries. MatrixTables are distributed by row.

`mt.rows()`
Get just the row fields as a table

`mt.annotate_rows(r3 = mt.r1 + mt.r2)`
Add new row field **r3** based on other row fields.

`mt.transmute_rows(r1_sq = mt.r1 ** 2)`
Like **`annotate_rows`**, but drops referenced fields.

`mt.filter_rows(~hl.is_nan(mt.r1))`
Filters out rows/entries for which given expression is false.

`mt.select_rows(mt.r1, mt.r2, r3=hl.coalesce(mt.r1, mt.r2))`
Select existing or create new row fields, dropping the rest.

`mt.sample_rows(p)`
Randomly downsample rows by keeping each row with probability **p**.

Entries

Entry fields are index by row and column. Each entry is a struct of potentially many fields.

`mt.entries()`
Flatten the matrix tables entry fields, row fields, and column fields into one giant table (**expensive!!!**)

`mt.annotate_entries(e3 = mt.e1*2)`
Create a new entry field for every entry in the MatrixTable (can be based on row and column fields)

`mt.transmute_entries(e3=mt.e1*2)`
Like **annotate_entries**, but drops referenced entry fields.

`mt.filter_entries(mt.e1 > 4)`
Filters out entries for which given expression is false.

`mt.select_entries(mt.e1, e2_len=hl.len(mt.e2))`
Select existing or create new entry fields, dropping the rest.

Creating MatrixTables

`hl.read_matrix_table('path/file.mt')`
Read in a hail formatted MatrixTable file.

`hl.utils.range_matrix_table(20, 10)`
Create a MatrixTable with 20 rows and 10 columns.

`hl.from_rows_table(ht)`
Create a MatrixTable with no columns from a table.

`hl.import_vcf('path/foo.vcf.bgz')`
Import a VCF file to create a variant by sample matrix table.

Writing MatrixTables

`mt.write('path/output_file.mt', overwrite=True)`
Write out a file in hail's MatrixTable format, overwriting any already existing file (by default, doesn't overwrite).

`mt = mt.checkpoint('path/output_file.mt')`
Combines **`mt.write`** and **`hl.read_matrix_table`** into one operation by writing and then immediately reading back in. Good to break up complicated procedures.

`hl.export_vcf(mt, 'path/output.vcf.bgz')`
Exports a file keyed by locus (**tlocus**) and alleles (**tarray** of **tstr**) to a VCF file.

Exploring MatrixTables

`mt.describe()`
Print information about the types of each field

`mt.summarize()`
Basic descriptive statistics for each field

`mt.count()`
# of rows and columns in MatrixTable.

`mt.show(n)`
Print first n rows of table (forces computation!)

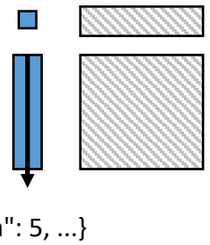
`mt.n_partitions()`
Check how many partitions are in this matrix table

`mt.head(n)`
Subset the matrix table to the first **n** rows.

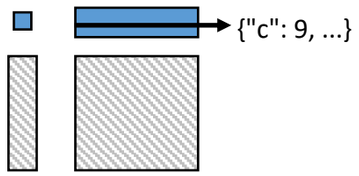
`mt.tail(n)`
Subset the matrix table to the last **n** rows.

Aggregations

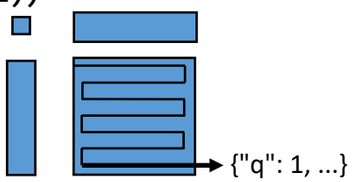
The three aggregate methods work across the matrix table and produce a local python value.



`mt.aggregate_rows(hl.agg.counter(mt.rf1))`
Aggregate over row fields, can also reference globals.

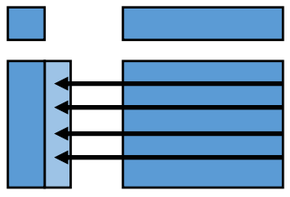


`mt.aggregate_cols(hl.agg.counter(mt.cf1))`
Aggregate over column fields, can also reference globals.

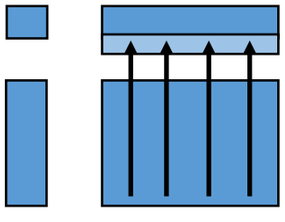


`mt.aggregate_entries(hl.agg.counter(mt.ef1))`
Aggregate over entry fields, can also reference row, column, and global fields.

The annotation methods over rows and columns also support aggregations over entries within each row/column.



`mt.annotate_rows(sum_of_ef1_by_row=hl.agg.sum(mt.ef1))`
Aggregate along each row of entries to create a new row annotation. Can reference column and entry fields in aggregations.



`mt.annotate_cols(sum_of_ef1_by_col=hl.agg.sum(mt.ef1))`
Aggregate along each column of entries to create a new col annotation. Can reference row and entry fields in aggregations.

Keying

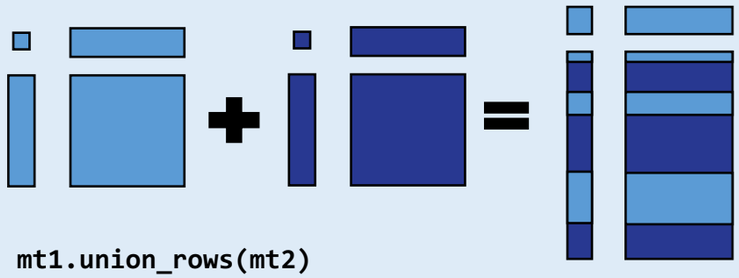
MatrixTables can be joined with tables on their row key or column key. To key:

`mt.key_rows_by(mt.rf1, mt.rf2)`
Keys the row by row fields **rf1** and **rf2**.

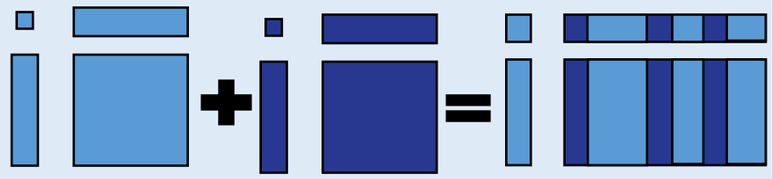
`mt.key_cols_by(mt.cf1)`
Keys the column by col field **cf1**.

Combining Datasets

Union MatrixTables

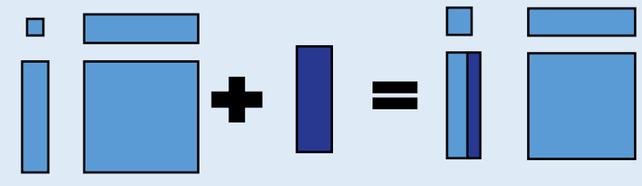


`mt1.union_rows(mt2)`
Combines rows of datasets with same column fields/keys.

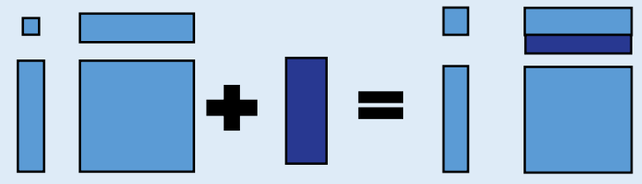


`mt1.union_cols(mt2)`
Combines cols of datasets with same row fields/keys.

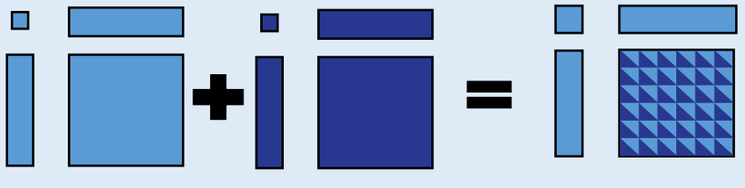
Join Tables onto MatrixTables



`mt1.annotate_rows(foo=ht2[mt1.key].foo)`
Joins the field of table **ht2** called **foo** onto **mt1**'s rows.



`mt1.annotate_cols(**ht2[mt1.key])`
Joins all of the fields of table **ht2** onto **mt1**'s columns keeping the same names they had (** is used to get all fields).



`mt1.annotate_entries(foo = mt2[mt1.row_key, mt1.col_key].ef)`
Joints the entry field named **ef** of **mt2** onto **mt1,** renaming it **foo** in the process.