Ramesh

Q — An Analytics Framework

Version 0.1

January 29, 2014

Contents

Q — An Analytics Framework

1 Introduction

Q is a vector language, designed for efficient implementation of counting, sorting and data transformations. It uses a single data structure — a table.

1.1 Motivation

I will motivate the need for Q by quoting from two of my Gods — Codd and Iverson. I could be accused of quoting scripture for my purpose (see below) and it is true that I am being selective in my extracts. However, that does not detract from their essential verity.

> The devil can cite Scripture for his purpose.
> An evil soul producing holy witness
> Is like a villain with a smiling cheek,
> A goodly apple rotten at the heart:

1.1.1 Extracts from Codd

The most important motivation for the research work that resulted in the relational model was the objective of providing a sharp and clear boundary between the logical and physical aspects of database management. We call this the *data independence objective*.

A second objective was to make the model structurally simple, so that all kinds of users and programmers could have a common understanding of the data, and could therefore communicate with one another about the database. We call this the *communicability objective*.

A third objective as to introduce high level language concepts *but not specific syntax) to enable users to express operations upon large chunks of information at a time. This entailed providing a foundation for set-oriented processing (i.e., the ability to express in a single statement the processing of multiple sets of records at a time). We call this the *set-processing objective*.

To satisfy these three objectives, it was necessary to discard all those data structuring concepts (e.g., repeating groups, linked structures) that were not familiar to end users and to take a fresh look at the addressing of data.

We have deviated from Codd's preference for the relational model. Instead, we choose to drop down one level to the table. As Codd writes:

Tables are at a lower level of abstraction than relations, since they give the impression that positional (array-type) addressing is applicable (which is not true of n-ary relations), and they fail to show that the information content of a table is independent of row order. Nevertheless, even with these minor flaws, tables are the most important conceptual representation of relations, because they are universally understood.

Lastly, in designing Q, we wanted it to be a data model as Codd defines one

A data model is a combination of at least three components:

1. A collection of data structure types (the building blocks);
2. A collection of operators or rules of inference, which can be applied to any valid instances of the data types listed in (1), to retrieve, derive, or modify data from any parts of those structures in any combinations desired;
3. A collection of general integrity rules, which implicitly or explicitly define the set of consistent database states or changes of states or both

1.1.2 Extracts from Iverson

The importance of language has been stated over the centuries. Iverson quotes the following from Whitehead:

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race

In the same vein, he quotes Babbage:

The quantity of meaning compressed into small space by algebraic signs, is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid

I would hesitate to claim that Q meets any of the following criteria that Iverson lays down for good notation. But it is definitely the guiding principle and aspirational goal for Q.

1. Ease of expressing constructs arising in problems. If it is to be effective as a tool of thought, a notation must allow convenient expression not only of notions arising directly from a problem, but also of those arising in subsequent analysis, generalization and specialization.
2. Suggestivity. A notation will be said to be suggestive if the forms of the expressions arising in one set of problems suggests related expressions which find application in other problems.
3. Ability to subordinate detail. Brevity is achieved by subordinating detail, and we will consider three important ways of doing this
   - the use of arrays
   - the assignment of names to functions and variables
   - the use of operators
4. Economy. Economy requires that a large number of ideas be expressible in terms of a relatively small vocabulary.
5. Amenability to formal proofs

1.2 Notations

Definition 1 $\delta(x, y) = 1$ *if* $x = y$ *and 0 otherwise*

Notation 1 *For convenience, we shall often blur the distinction between arrays, sets and tables. Hence,* $T.f$ *refers to field* f *of table* T*. Similarly,* $T[i].f$ *means the* i^{th} *value of field* f *of table* T*.*

Notation 2 *We shall often use* T_X *to refer to the table whose name is* X*.*

Notation 3 *Also,* $T|f(\ldots)$ *refers to the subset of* T *for which the predicate* $f(\ldots)$ *is satisfied.*

Notation 4 $Count(T)$ *is cardinality of* T

Notation 5 $NumVal(T.f, v) = \sum_i \delta(T[i].f, v)$

Notation 6 $Unique(T.f)$ *be the unique values of* $T.f$

Notation 7 $CU(T.f)$ *is the count of the unique values of* $T.f = Count(Unique(T.f))$

Notation 8 $(f_1 : v_1 \ldots f_N : v_N) \in T \Rightarrow \exists j : T[j].f_1 = v_1 \wedge \ldots T[j].f_N = v_N$

Notation 9 $(v_1 \ldots v_N) = T[i].(f_1, \ldots f_N) \Rightarrow v_1 = T[i].f_1 \ldots v_N = T[j].f_N$

Notation 10 $\exists!$ *is read as "there exists a unique"*

Definition 2 *Let* $T_1.f_1 = Join(T_2, f_2, l_2, l_1)$ *be a field in* T_1 *created by joining field* f_2 *in Table* T_2 *using* l_2 *in* T_2 *and* l_1 *in* T_1 *as the link fields.*

2 System Capabilities

2.1 Field Types

B bit

I1 1-byte signed integer

I2 2-byte signed integer

I4 4-byte signed integer

I8 8-byte signed integer

F4 4-byte float

F8 8-byte float

SC string. Fixed length.

SV string. Variable length. Must have columns len, off in same table that allow us to identify value in a particular cell.

Note that if a string can be treated as a label, then we replace it by an I4 integer which is foreign key to field `idx` in a dictionary table.

1. If an operation is expected to create a table T and a table by that name exists, then the original table is first deleted. Examples of operations that create tables are
   (a) add table — Section 3.6.1
   (b) load from CSV — Section 3.37
2. If an operation is expected to create a field with name f and a field with such a name exists, then the old field is deleted **after** the operation is performed and replaced by the newly created field.
3. A table has a non-null, unique name. must be unique.
4. A field has a non-null name that is unique within a table.
5. Most operations cannot be performed on the nn field of a primary field. When this is possible, we will say so explicitly.

Notation 11 *$alldef(f)$ means that all values of f are defined. Hence, f cannot have an* `nn` *field.*

I have disabled this invariant. Need to be sure that it is okay to do so.

Invariant 1 *SC field cannot have a nn field.*

2.2 Environment Variables

Q_DOCROOT Name of directory where meta data is stored. Needs to be unique for each table space.

Q_DATA_DIR Name of directory where data is stored Needs to be unique for each table space.

Q_RUN_TIME_CHECKS If it is set, then we execute run time.

3 Functions

3.1 Accessing meta-data

3.1.1 file_to_fld

Input is n, positive integer. Output is `<tbl>:<fld>` if there is a field with this as its file number.

3.1.2 list_files

Arguments are

1. D — data directory (if not specified, then environment variable `Q_DATA_DIR` is used)

Returns list of files in the docroot that are in this directory. Output format is:

1. is_external — 1 means that it is owned by somebody else; else, 0
2. file_name — unique, non-null string

3.1.3 dump

Arguments are

1. name of file into which tables are dumped
2. name of file into which fields are dumped

Creates CSV files of meta-data.

3.1.4 list_tbls

Returns list of tables in the docroot as a 3-column CSV text file

1. tbl ID
2. name
3. display name

3.1.5 orphan_files

Arguments are

1. D — data directory (if not specified, then environment variable `Q_DATA_DIR` is used)

Returns list of files in the data directory that are not in meta data.

3.1.6 is_tbl

Arguments

1. T — table

If table T exists, output is $1, tbl\_id)$. Else, output is $(0, -1)$

3.1.7 is_fld

Arguments

1. T — table
2. f — field

If table T exists and field f exists in table T, output is $1, fld\_id)$. Else, output is $(0, -1)$

3.1.8 is_aux_fld

Arguments

1. T — table
2. f — field
3. $auxtype$ — can be `nn, len, off`

Returns $1, fld\_id)$ if

1. f exists in T
2. f is a primary field
3. f has an auxiliary field of type auxtype

Else, returns $0, -1)$

3.1.9 describe

Arguments

1. T — table
2. f — field
3. $attribute$ —

There are 3 invocation types

1. Only T specified. For each field, we print
   (a) ID
   (b) name
   (c) field type — Section 2.1
   (d) is external?
   (e) is fk? If so, print tbl ID; else, print -1
2. T, f specified. For that field, we print
   (a) name
   (b) fname
   (c) fldtype
   (d) parent ID
   (e) nn fld ID
   (f) dict tbl ID
   (g) srttype
   (h) is external?
   (i) count — only if field is I1 or B; else, -1
3. All three specified. Prints that attribute of that field of that table

3.2 get_nR

Arguments

1. T — table

Returns number of rows in table T

3.3 Modifying meta-data

3.3.1 rename

Two styles of invocation.

1. to rename a table
   (a) T_1 — old name of table
   (b) T_2 — new name. If such a table exists, it is deleted
2. to rename a field
   (a) T_1 — name of table
   (b) f_1 — old name of field
   (c) f_2 — new name of field. If such a field exists, it is deleted

3.3.2 drop_nn_fld

1. T
2. f

If $T.f$ has an `nn` field, it is deleted i.e., all values of f are now defined. It is important to be sure that you know the value of the field in its undefined state. Usually, this will be 0, but you had better be damn sure!

3.3.3 del_tbl

Arguments

1. T — table

Restrictions

1. No other field should depend on this table

Deletes table T and all fields in T
Can also be invoked as `q del_tbl tbl1:tbl2`
Can also be invoked as `q delete tbl1:tbl2`

3.3.4 del_fld

Arguments

1. T — table
2. f — field

Deletes field f in table T. If f is a primary field, then all auxiliary fields (if any) are deleted as well. If $is\_external = 0$, then the corresponding file is **not** deleted; else, it is deleted.
Can also be invoked as `q del_tbl tbl fld1:fld2`
Can also be invoked as `q delete tbl fld1:fld2`

3.3.5 set_meta

Arguments

1. T
2. f
3. $attr$
4. $value$

Sets attribute a of field f in table T to value v. If f is null, then it sets attribute a of table T. Options for a are

cnt — sets count. $f.fldtype \in \{B, I1\}$

srttype — can be set to `unknown, ascending, descending, unsorted`

fldtype — can be set to `I1, I2, I4, I8, F4, F8`

is_dict_tbl — can be set to true or false (table attribute)

3.4 import

Arguments are

1. D_{from} — from docroot
2. T_{from}
3. T_{to}

Copies the table T_{from} from the docroot to the current docroot, D_{to} Notes

1. $D_{from} \neq D_{to}$
2. If T_{to} exists in D_{to}, then it is deleted
3. If we delete a field in D_{to} or the entire table itself, there is no impact on T_{from} in D_{from}. (As an implementation note, this is because we mark the imported fields with an `is_external` flag and we do not allow modifications of these files. This is specially important for operations which modify the underyling storage like Section 3.14.)

3.5 Accessing Data

3.5.1 pr_fld

Arguments

1. T — table
2. f — field
3. selection is f_c or $lb : ub$ — condition field or range. If null, all rows printed
4. output file name. If null, writes to stdout

Prints all rows of field f of table T. Behavior of print can be modified by specifying third parameter as either

1. f_c — printing of $T[i].f$ is suppressed if $T[i].f_c = false$.
2. $lb : ub$ — we print rows with lb as lower bound inclusive and ub as upper bound exclusive

If range is specified, following must be satisified

1. $lb > 0$
2. $ub \leq n_R$
3. $lb < ub$

3.5.2 bindmp

Arguments are

1. T — input table
2. $f_1 : f_2 : \ldots f_N$ — list of fields separated by colon. Must have at least one field
3. f_c — boolean field in T_1 optional
4. F — output file to be created
5. D — output directory in which file is to be created. If null, current working directory is used.

Creates file F by dumping the desired fields, a row at a time. The i^{th} row is output if either (i) $f_c = \perp$ or (ii) $f_c \neq \perp \wedge f_c[i] = 1$

Restrictions are

1. $fldtype(f_i) \in \{I1, I2, I4, I8, F4, F8\}$
2. $alldef(f_i)$
3. $f_c \neq \perp \Rightarrow fldtype(f_c) = I1$

3.6 Creating Data

3.6.1 add_tbl

Arguments

1. T — table
2. n_R — number of rows. Must be specified.

Creates a table with name T. If a table with such a name exists, it will be deleted.

3.6.2 add_fld

Creates field f in table T, whose values are stored in specified data file. Arguments are

1. T — table
2. f — field
3. attributes — string. This is a colon separated concatenation of `name=value` pairs where the names are
   (a) `file` — name of file which contains data for field
   (b) `dir` — directory in which file exists
   (c) `fldtype` — see Section 2.1 (however, string is not allowed)

Error conditions

1. $T.n_R = 0$
2. If $T.f$ exists, it will be over-written.
3. name of file must not be similar to internal files i.e., first character is underscore and others are digits

3.6.3 dup_fld

Arguments are

1. T — table
2. f_1 — field in T
3. f_2 — newly created field in T

Creates f_2 in$T$ which is a clone of f_1. Requires

1. $f_1 \neq f_2$
2. $f_1.fldtype \notin \{B, S\}$

3.6.4 mk_idx

Arguments are

1. T — table
2. f — field
3. $fldtype \in \{I1, I2, I4, I8\}$

Creates a field f of specified type which starts at 0 and increments by 1.

3.6.5 get_val

Arguments are

1. T — table
2. f — field
3. i — index

Returns $T[i].f$.
Restrictions are

1. $0 \leq i < |T|$
2. $fldtype(f) \in \{I1, I2, I4, I8, F4, F8\}$

3.6.6 set_val

Arguments are

1. T — table
2. f — field
3. $l : u$ — range, such that $0 \leq l < u < nR$
4. v — value

$\forall i : l \leq i < u, T[i].f = v$
Restrictions are

1. $fldtype(f) \in \{I1, I2, I4, I8\}$

3.6.7 copy_tbl

Creates T_2 with as many rows as in T_1 and copies all fields from T_1 to T_2 using Section 3.6.8. Note that string fields do not get copied. Also, you cannot specify a conditional field.
LIMITATIONS: SC and SV fields do not get copied over

3.6.8 copy_fld

Creates field f_2 table T_2 by copying $T[i].f$ to T_2 if $T_1[i].f_c = true$. The order is preserved in the copy operation. Arguments are

1. T_1 — input table
2. f_1 — input field
3. f_c or $l : u$ — optional condition field or range
4. T_2 — output table
5. f_2 — newly created output field

Restrictions are

1. If T_2 does not exist, it will be created
2. If T_2 exists
   (a) If no selection specified, $|T_1| = |T_2|$.
   (b) If selection specified as f_c, then $NumVal(T_1.f_c, true), = |T_2|$
   (c) If selection specified as $l : u$, then $|T_2| = u - l$
3. $Type(f_1)$ must be one of `I1, I2, I4, I8, F4, F8`

3.6.9 copy_fld_ranges

Simiar to Section 3.6.8 but uses a set of ranges instead of a condition field. Arguments are

1. T_1 — input table
2. f_1 — input field in T_1
3. T_r — range table
4. f_l — lower bound field in T_r
5. f_u — upper bound field in T_r
6. T_2 — output table
7. f_2 — output field in T_2

Restrictions are

1. If T_2 does not exist, it will be created
2. $0 \leq T_r[i].lb < T_r[i].ub \leq |T_1|$
3. If T_2 exists, $\sum(T_r[i].ub - T_r[i].lb) = |T_2|$
4. Field type of f_1 must be one of `I1, I2, I4, I8, F4, F8`
5. $fldtype(f_l) = fldtype(f_u) = I8$
6. $alldef(f_l), alldef(f_u)$

3.6.10 mv_fld

Arguments are

1. T_1 — input table
2. f_1 — input field
3. T_2 — output table

Moves field f from table T_2 to T_1

Restrictions

1. $|T_1| = |T_2|$
2. $T_1 \neq= T_2$
3. $fldtype(f_1) \in \{I1, I2, I4, I8, F4, F8\}$

3.6.11 mk_nn_fld

Arguments are

1. T_S — input table
2. f_S — input field
3. T_D — output table
4. f_D — output field

Makes field f_S the `nn` field of field f_D. Note that f_S will cease to exist.

Restrictions

1. $|T_S| = |T_D|$
2. It is okay for $T_S = T_D$
3. $fldtype(f_S) = \{B, I1\}$
4. $alldef(f_S)$
5. f_S must not be in use by any other field

3.6.12 break_nn_fld

Arguments are

1. T_S — input table
2. f_S — input field
3. T_D — output table
4. f_D — output field

Makes field f_D from the `nn` field of field f_S. At end,$alldef(f_S) = true$
Restrictions

1. $alldef(f_S) = false$
2. It is okay for $T_S = T_D$
3. $fldtype(f_D) = \{B, I1\}$
4. $f_S \neq f_D$

3.7 Operations

3.7.1 regex_match

Arguments are

1. T
2. f
3. r — regular expression
4. k — kind of match
5. f_C — newly created condition field in T

1. $fldtpye(f_C) = I1$
2. $fldtpye(f) \in \{SC, SV\}$
3. $k \in \{exact\}$ (other kinds of matches to be supported)

3.7.2 stride

Arguments are

1. T_S — source table
2. f_S — input field
3. i_0 – starting position
4. m – stride
5. T_D — destination table
6. f_D — count field
7. n_D — number of rows in destination table.

Selects every m^{th} value of $T_S.f_S$ starting from index i_0. Creates T_D with a single column f_D and n_D rows as $T_D[i].f_D = T_S[i * m + n].f_S$.

- $0 \leq n < n_S$
- $0 \leq m < n_S$
- n_D will be reduced from the desired value so that $(n_D - 1) * m + n < |T_S|$

1. $fldtype(f_S) \in \{I4, I8\}$
2. $alldef(f_S)$

3.7.3 count

Arguments are

1. T_S — source table
2. f_S — input field
3. f_c – optional condition field in T_S.
4. f_D — destination table
5. f_D — count field

$T_D[j].f_D = |T_S[i].f_S = j|$
$f_c \neq \perp \Rightarrow T_D[j].f_D = |T_S[i](f_S = j, f_C = 1)$
Restrictions are

1. f_S has no `nn` field
2. fldtype of f_D is `I1, I2, I4, I8`
3. $|T_D| \leq I4_{max}$. For good peformance, it should be quite small.
4. $|T_S| \leq I4_{max}$.
5. $fldtype(f_D) = I4$
6. $0 \leq f_S < |T_D|$ — f_S must be thought of as an index into T_D

3.7.4 countf

Arguments are

1. T_S — source table
2. f_I — field that serves as index into T_D
3. f_V — field whose values are to be accumulated
4. f_c – optional condition field in T_S.
5. f_D — destination table

6. f_D — count field

7. whether to run safe or not, optional field, default true. Specified as `safe_mode=[true]`

Section 3.7.3 is a special case of this operator where $f_V = 1$.

1. $f_C = \perp \Rightarrow T_D[j].f_D = \sum_i \delta(j, T_S[i].f_I) \times T_S[i].f_V$

2. $f_C \neq \perp \Rightarrow T_D[j].f_D = \sum_i f_C[i]\delta(j, T_S[i].f_I) \times T_S[i].f_V$

Restrictions are

1. f_S has no `nn` field

2. fldtype of f_D is `I1, I2, I4, I8`

3. $|T_D| \leq I4_{max}$. For good peformance, it should be quite small.

4. $|T_S| \leq I4_{max}$.

5. $fldtype(f_D) = I4$

6. $0 \leq f_S < |T_D|$ — f_S must be thought of as an index into T_D

7. If safe mode is false, then we do not check each value of f_I to see whether it is in the range $[0, |T_D| - 1]$. Can lead to segfaults if you do not know what you are doing!

3.8 count_ht

The function `count` in Section 3.7.3 is useful when the range of values is $0, 1, \ldots, |T_2| - 1$. When one does not know the range of values, `count_ht` is useful.

Arguments are

1. T_S — source table

2. f_S — input field

3. f_C — condition field (optional)

4. f_D — destination table

Creates table T_D with 2 fields, `value` and `count`.

1. $|T_D| < 2^{21}$. Important limitation of current implementation.

2. $T_S \neq T_D$

3. $f_S \neq f_C$

4. $fldtype(f_S) \in \{I1I2, I4, I8\}$

5. $T_D.value \subseteq T_S.f_S$

6. $T_D[j].count = |T_S[i].f_S = T_D[j].value$

7. $fldtype(value) = fldtype(f_S)$

8. $fldtype(count)$ = smaller of `I4, I8`. Note that I8 used only if some value occurs more than $I4_{max}$ times.

Restrictions are

1. Number of distinct values of f_S must be less than 2^{20}. This limitation should be lifted.

2. For any particular value of f_S, the number of rows with that value must be $leqI4_{max}$

3. Note that T_D might be empty. Caller needs to check.

3.9 s_to_f

Produces a field from a scalar. Arguments are

1. T

2. f

3. how to create the field

The following options are supported

CONST In this case a value must be a provided. Restrictions

1. fldtype is `I1, I2, I4, F4, I8`

SEQ In this case, a starting value and an increment must be provided. Restrictions

1. fldtype is `I2, I4, I8`
2. No overflow/underflow of values e.g., if fldtype is I1, then values produced must be in range $[-128, 127]$

Operations supported for Section 3.9 are in Table 1

Operation	B	I1	I2	I4	I8	F4	F8
const		✓	✓	✓	✓	✓	
seq			✓	✓	✓		
set_labels				✓			

Table 1: Supported Operations for s_to_f

3.10 pack

Arguments are

1. T

2. $f_1 : f_2 : f_3 \dots f_N$ — input fields

3. $n_1 : n_2 : n_3 \dots f_N$ — non-negative integers indicating shift amount

4. $fldtype$ — type of output field f_o
5. f_o — output field created

Creates f_o in T by performing an “or” of (i) f_1 shifted left by n_1, (ii) f_2 shifted left by n_2, . . .
Restrictions

1. $2 \leq N \leq 8$
2. $fldtype(f_o) \in \{I4, I8\}$
3. all input fields must be alldef
4. fldtypes of all input fields must be in $\{I1, I2, I4, I8\}$
5. $f_i \geq 0$ — no negative values in input
6. Output field not one of input fields
7. $fldtype(f_o) = I4 \Rightarrow fldtype(f_i) \neq I8$

3.11 unpack

Arguments are

1. T
2. f_I — input field
3. $n_1 : n_2 : n_3 \ldots n_N$ — non-negative integers indicating shift amount
4. $m1 : m2 : m3$ — number of bits for each input field
5. $t1 : t2 : t3$ — types of output fields
6. $f_1 : f_2 : f_3 \ldots f_N$ — output fields

Creates field f_i in T of type t_i by (unsigned) shifting f_I n_i bits to the right and then “and-ing” it with a mask which is all zeroes with the m_i least signicant bits being 1.
Restrictions

1. $2 \leq N \leq 8$
2. $fldtype(f_I) \in \{I4, I8\}$
3. input field must be alldef
4. fldtypes of all output fields must be in $\{I1, I2, I4, I8\}$
5. Output field not one of input fields
6. $fldtype(f_i) = I1 \Rightarrow m_i \leq 7$
7. $fldtype(f_i) = I2 \Rightarrow m_i \leq 15$
8. $fldtype(f_i) = I4 \Rightarrow m_i \leq 31$
9. $fldtype(f_i) = I8 \Rightarrow m_i \leq 63$
10. $fldtype(f_I) = I4 \Rightarrow fldtype(f_i) \neq I8$

3.12 subsample

1. T_1 — input table (has n_1 rows)
2. f_1 — input field
3. n_2 — number of rows in output table
4. T_2 — output table
5. f_2 — output field

Creates table T_2 with single field f_2 which contains $0 < n_2 < n_1$ rows of $T_1.f_1$ selected at random.

1. $fldtype(f_1) \in \{I4, I8\}$
2. $fldtype(f_2) = fldtype(f_2)$
3. $alldef(f_1)$
4. This is not completely uniformly at random because of need to cilk-ify the loop. So, what we do is to break the input into blocks and then select uniformly at random from each block.

3.13 wisifxthenyelsez

1. T — table
2. w — output field
3. x — boolean field
4. y — input field (or scalar)
5. z — input field (or scalar)

Creates field w, whose type is same as that of field y or z, depending on which is specified. Algorithm in Figure 3.13. Restrictions

1. $fldtype(y) = fldtype(z)$
2. $fldtype(y) \in \{I1, I2, I4, I8, F4\}$
3. It is possible for one of y or z (but not both) to be a scalar. In that case, the type of w is the type of the field that is not a scalar.

```
if $T_x[i] = \perp$ then
    $T_w[i] \leftarrow \perp$
else
    if $T_x[i] = true$ then
        $T_w[i] \leftarrow T_y[i]$
    else
        $T_w[i] \leftarrow T_z[i]$
    endif
endif
```

Figure 1: Pseudo-code for wisifxthenyelsez

3.14 fop

Arguments are

1. T — table
2. f — field in T. Modified in situ
3. $\circ$ — operation on f

Restrictions are

1. $alldef(f)$
2. $internal(f)$

Operations supported for Section 3.14 are in Table 2

Operation	B	I1	I2	I4	I8	F4	F8
permute				✓	✓		
sort				✓	✓		
saturate		✓	✓	✓	✓		
zero_after_n	✓						

Table 2: Supported Operations for fop

1. permute `op=[permute]` Field types supported are `I4`. Does random permutation.
2. sort ascending, `op=[sort]:order=[asc]`. Field types supported are `I4, I8`
3. sort descending `op=[sort]:order=[dsc]` Field types supported are `I4, I8`
4. saturate `op=[saturate]:maxval=[user specified]` Field types supported are `I1, I2, I4, I8`.
   $T[i].f \leftarrow max(T[i].f, maxval)$
5. zero_after_n `op=[zero_after_n]:n=[M]`

- Field types supported are B.
- $0 < M < |T|$
- Starting with $i = |T| - 1$ and decrementing i by 1, set $T[i].f \leftarrow 0$ as long as $\sum_{j=1}^{j=i-1} T[i].f > n$. Note that we might include $m \geq n$ 1's but $m < n + 64$.

3.15 num_in_range

Arguments are

1. T_1
2. f_1 — field in T_1
3. T_2
4. f_{lb} — field in T_2
5. f_{ub} — field in T_2
6. f_C — newly created field in T_2

Consider the i^{th} row of T_2. Let $l = T_2[i].lb, u = T_2[i].ub$. Then, $T_2[i].cnt$ counts the number of rows of T_1 where $l \leq f_1 \leq u$

Restrictions

1. More efficient when $srttype(f_1) = asc$
2. $f_{lb} \neq f_{ub}$
3. ranges specified by f_{lb}, f_{ub} are non-overlapping
4. $fldtype(f_1) = fldtype(f_{lb}) = fldtype(f_{ub}) = I4$
5. $alldef(f_1), alldef(f_{lb}), alldef(f_{ub})$
6. $T_2[j].f_{lb} \leq T_2[j].f_{ub}$
7. $|T_1| < I4_{max}$
8. $fldtype(f_C) = I4$
9. Recommend usage of this only when $|T_2|$ is small. In particular, $|T_2| \leq 4096$
10. $srttype(f_{lb}) = srttype(f_{ub}) =$ ascending

3.16 f1s1opf2

Arguments are

1. T_1
2. f_1
3. scalar (or colon separated list of scalars)
4. f_2

Operations supported for Section 3.16 are in Table 3

3.17 f1f2_to_s

Produces a scalar from 2 fields. Arguments are

1. T — table
2. f_1 — field in T.
3. f_2 — field in T.
4. $\circ$ — operation on f

$circ = sum$. Add up the values of f_2 where $f_1 = 1$

1. $fldtype(f_1) = I1$
2. $fldtype(f_2) = \{I1, I2, I4, I8\}$
3. $alldef(f_1), alldef(f_2)$

3.18 f_to_s

Produces a scalar from a field. Arguments are

1. T — table
2. f — field in T.
3. $\circ$ — operation on f

Operations supported are in Table 4. The acronym mms stands for min, max, sum. Depending on the operation, output is as follows

1. is sorted — ascending, descending, unsorted, unknown
2. get_idx — call looks like `op=[get_idx]:val=[v]`

   Returns i such that $T[i].f = v$. If no i satisifes the inequality, -1 is returned. If more than 1 satisifies it, an aribtrary one is returned.

   Current restrictions are

   (a) $Type(f)$ = `I1, I2, I4, I8`
   (b) f has no **nn** field
3. min — $n_1 : n_2$ where n_1 is min value and n_2 is number of non-null values
4. max — $n_1 : n_2$ where n_1 is max value and n_2 is number of non-null values
5. sum — $n_1 : n_2$ where n_1 is sum and n_2 is number of non-null values
6. mms — $n_1 : n_2 : n_3 : n_4$ where n_1 is min, n_2 is max, n_3 is sum and n_4 is number of non-null values

3.19 f1opf2

Arguments are

1. T_1 — table
2. f_1 — field in T_1
3. $\circ$ — operation to b performed on f_1
4. f_2 — newly created field in T_1

Creates field f_1 in T_1 by performing operation $\circ$ on f_1 in T_1. Operations supported are in Table 5. Detailed explanations in Section 3.19.1.

3.19.1 Details for f1opf2

We explain each of the operations below

CONV Converts from one type to another. Supported conversions are in Table 6

BITCOUNT Counts the number of bits. Example `op=bitcount` $Type(f_2) = I4$

SQRT Operation is $\sqrt{x}$. Example `op=sqrt` $Type(f_2) = Type(f_1)$

NEGATION Toggles bits. Example is `op=!` $Type(f_2) = Type(f_1)$

ONE's COMPLEMENT Takes one's complement. Example is `op=~` $Type(f_2) = Type(f_1)$

INCREMENT Increments by 1 Example is `op=++` $Type(f_2) = Type(f_1)$

DECREMENT Decrements by 1 Example is `op=--` $Type(f_2) = Type(f_1)$

CUM Accumulates the values of f_1. Notes

1. $i = 0 \Rightarrow f_2[i] = f_1[i]$; else, $f_2[i] = f_1[i] + f_2[i-1]$
2. $alldef(f_1)$
3. user has to specify argument `newtype` which specifies $fldtype(f_2)$ and which can be `I4, I8`
4. $fldtype(f_1) = \{I1, I2, I4, I8\}$

SHIFT Shifts the values up or down up to a maximum of 16 Example is `op=shift:val=N`. In this case, $f_2[i] = f_1[i+N]$. If $N > 0$, then it like a shift down of the column i.e., the first N values of f_2 are set to `newval`. If $N < 0$, then it is like a shift up and the last N values will be set to `newval`. Note that $N \neq 0$. Currently, $|N| \leq 16$.

SMEAR "smears" the selection by n_R to the right and n_L to the left i.e.,

- $f_1[i] = 1 \Rightarrow \forall j : 1 \leq j \leq n_R, f_2[i+j] \leftarrow 1$
- $f_1[i] = 1 \Rightarrow \forall j : 1 \leq j \leq n_L, f_2[i-j] \leftarrow 1$

1. user has to specify argument `plus` which is the number of positions to the right that the selection should be smeared

2. user has to specify argument `minus` which is the number of positions to the right that the selection should be smeared
3. $n_R \geq 0$, $n_L \geq 0$, n_R and n_L cannot both be 0
4. There is a maximum value for n_R, n_L
5. $fldtype(f_1) = fldtype(f_2) = I1$
6. $alldef(f_1)$

There are 2 arcane statistical functions, for which $fldtype(f_1) = fldtype(f_2) = F8$. They are

1. normal cdf inverse
2. calculation of p-value from z-value

3.20 xfer

Arguments are

1. T_1
2. f_S — must exist in T_1
3. T_2
4. f_{idx} — must exist in T_2
5. f_D — newly created in T_2

Creates f_D in T_2 with same type as f_S as follows. $T_2[i].f_D \leftarrow T_1[T_2[i].f_{idx}].f_S$
Restrictions

1. $0 \leq T_2[i].f_{idx} < |T_2|$
2. $fldtype(f_S) \in \{I1, I2, I4, I8, F4, F8, SC\}$
3. $fldtype(f_{idx}) = I4$

3.21 is_a_in_b

1. T_1 — table
2. f_1 — field in T_1.
3. T_2 — table
4. f_2 — field in T_2.
5. f_c — newly created condition field in T_1.
6. f_S — source field in T_2
7. f_D — destination field in T_1

One of

1. $\forall i : T[i].f_1 \in T_2.f_2 \Rightarrow T[i].f_c \leftarrow true$
2. $\exists j : T_2[j].f_2 = T_1[i].f_1 \Rightarrow T_1[i].f_D \leftarrow T_2[j].f_S$

Restrictions

1. $Type(f_c) = I1$
2. Either specify f_c or specify f_S, f_D but not both
   (a) $f_c = \perp \Rightarrow f_S \neq \perp \wedge f_D \neq \perp$
   (b) $f_c \neq \perp \Rightarrow f_S = \perp \wedge f_D = \perp$
3. $fldtype(f_1) = fldtype(f_2)$
4. $fldtype(f_1) =$ `I1, I2, I4, I8`
5. f_2 has no null values
6. f_2 is sorted in ascending order

3.22 f1opf2f3

elabelf1opf2f3
Arguments are

1. T_1 — table
2. f_1 — field in T_1
3. $\circ$ — operation to be performed on f_1
4. f_2 — field in T_1
5. f_3 — newly created field in T_1

Creates field f_2, f_3 in T_1 by performing operation $\circ$ on f_1 in T_1.
Restrictions

1. $fldtype(f_1) = I8$
2. $fldtype(f_2) = fldtype(f_3) = I4$
3. $op = unconcat$. Breaks up the input 8-byte integer into 2 4-byte integers

3.23 f1f2opf3

Arguments are

1. T_1 — table
2. f_1 — field in T_1
3. f_2 — field in T_1
4. $\circ$ — operation to be performed on f_1
5. f_3 — newly created field in T_1

Creates field f_3 in T_1 by performing operation $\circ$ on f_1 f_2 in T_1. Operations supported are in Table 7.
Most operations in Section 3.23 are self-explanatory. Here are the rest

1. concat $f_3 \leftarrow (f_1 << 32)|f_2$
2. `&&!` $f_3 \leftarrow (f_1 \wedge (\neg f_2)$

Restrictions

1. $Type(f_1) = Type(f_2)$
2. $Type(f_3) = Type(f_1)$, except when otherwise noted in last column of Table 7

3.24 crossprod

Arguments are

1. T_1
2. f_{11} — field in T_1
3. f_{12} — field in T_1
4. T_2
5. f_{21} — newly created field in T_2
6. f_{22} — newly created field in T_2
7. T_{aux} — optional
8. f_{cnt} — optional
9. $mode$

Creates a new table T_2 with fields f_{21} and f_{22}. If such a table exists, it is deleted.
Mode can be one of

1. complete
2. upper triangular

3. upper triangular minus diagonal

To understand this operation, start by assuming that $T_{aux} = \perp$. Let $n_2 = |T_2|, n_1 = |T_1|$. Assume $f_{11} = (1, 2, 3)$ and $f_{21} = (4, 5, 6)$

1. mode = `COMPLETE`. $n_2 = n_1 \times n_1$. $(f_{21}, f_{22}) = \{(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)\}$
2. mode `UPPER TRIANGULAR` . $n_2 = \frac{n_1 \times (n_1+1)}{2}$. $(f_{21}, f_{22}) = \{(1, 4), (1, 5), (1, 6), (2, 5), (2, 6), (3, 6)\}$
3. mode = `UPPER TRIANGULAR MINUS DIAGONAL`. $n_2 = \frac{n_1 \times (n_1-1)}{2}$. $(f_{21}, f_{22}) = \{(1, 5), (1, 6), (2, 6)\}$

Restrictions

1. $n_1 \geq 2$
2. $T_{aux} = \perp \Leftrightarrow f_{cnt} = \perp$
3. $alldef(f_{11}), alldef(f_{12})$
4. $fldtype(f_{11}) = fldtype(f_{12}) = I4$
5. $fldtype(f_{21}) \leftarrow I4$
6. $fldtype(f_{22}) \leftarrow I4$

3.25 is_A_in_b

1. T_A — table
2. l_A — link field in T_A
3. T_B — table
4. l_A — link field in T_B
5. f_c — newly created field in T_A

Restrictions

1. $srttype(l_A) = srttype(l_B =$ ascending
2. $alldef(l_A) = alldef(l_B =$ true
3. $fldtype(l_A) = fldtype(l_B$
4. $fldtype(l_A) \in \{I4, I8\}$
5. $fldtype(l_B) \in \{I4, I8\}$

Creates field $f_c \in T_A$, such that

1. $alldef(f_c) =$ true
2. $fldtype(f_c) = I1$
3. $T_A[i].f_c = true \Rightarrow \exists j : T_A[i].l_A = T_B[j].l_B$

Considerations

1. Current implementation best used when T_A is large and T_B is small. This could be fixed when we bring meta data to bear.

3.26 srt_join

1. T_s — table
2. l_s — link field in T_1
3. f_s — source field in T_1
4. T_d — table
5. l_d — link field in T_2
6. f_d — newly created field in T_2
7. m — type of join.

1. $m =$ `reg, min, max, sum, cnt, and, or`

Let $T_d[i].l_d = v_d$. $T'_s = T_s[l_s = v_d]$. Then, $T'_s = \phi \Rightarrow T_d[i] = \perp$. Depending on the mode,

reg $T_d[i].f_d \in T'_s.f_s$. If $|T'_s] > 0$, then an arbitrary value in it is used.

min $T_d[i].f_d = min(T'_s.f_s)$

max $T_d[i].f_d = max(T'_s.f_s)$

sum $T_d[i].f_d = \Sigma(T'_s.f_s)$

and $T_d[i].f_d = and(T'_s.f_s)$

or $T_d[i].f_d = or(T'_s.f_s)$

cnt $T_d[i].f_d = |T'_s|)$

3.27 rng_join

1. T_s — source table
2. l_s — source field in T_1
3. f_l — lower bound field in T_1
4. f_u — upper bound field in T_1
5. T_d — destination table
6. l_d — link field in T_2
7. f_d — newly created field in T_2

1. $|T_s| \leq I4_{max}$
2. $f_l \neq f_u$
3. $l_s \neq f_u$

4. $l_s \neq f_l$
5. $fldtype(f_l) = fldtype(l_u) = fldtype(l_d) = I4$
6. $srttype(f_l) = srttype(l_u) = srttype(l_d) = asc$
7. $alldef(l_s), alldef(l_u), alldef(l_d)$
8. $fldtype(f_d) = I4$
9. $l_s \neq \perp \Rightarrow alldef(l_s), fldtype(l_s) = I4$
10. $T_S[j].f_l \leq T_S[j].f_u$

Let $v = T_D[i].l_d$. If $\nexists j : T_S[j].f_l \leq v \leq T_S[j].f_u \Rightarrow T_D[i].f_d = \perp$. Else,

- $f_l = \perp \Rightarrow T_D[i].f_d = j$
- $f_l \neq \perp \Rightarrow T_D[i].f_d = T_S[j].f_l$

3.28 count_vals

Arguments are

1. T_1 — input table
2. f_1 — input field in T_1
3. f_1^c — optional counter field in T_1
4. T_2 — output table
5. f_2 — output field in T_1
6. f_2^c — counter field in T_2

Restrictions

1. $Type(f_1) = Type(f_2$
2. $Type(f_1) = \texttt{I4, I8}$
3. $Type(f_2^c) = \texttt{I4}$. This means that
   (a) if f_1^c is not provided, then the maximum number of occurrences of any value in f_1 is $I4_{max}$
   (b) if f_1^c is provided, similar restriction (see below)

We create a table T_2 where f_2 is the unique, non-null values of f_1 in T_1. If f_1^c is not specified, $f_2^c[i] = |T_1[f_1 = T_2[f^c - 2[i]]|$, the number of times the corresponding value of f_1 occurred in T_1. If it is specified, then the output is computed in much the same way but weighted by f_1^c. In other words, let $T_2[j].f_2 = V$. Then,

$$T_2[j].f_2^c = \sum_i T_1[i].f_1^c[i] \times \delta(T_1[i].f_1, V)$$

3.28.1 Enhancements

Do not create an output count field if not desired.

3.29 mk_bins

Arguments are

1. T_1
2. f_1
3. T_2 — newly created table
4. n_2 — desired number of bins

Creates n'_2 rows in T_2 (where $1 \leq n'_2 \leq n_2$) with columns

1. `lb` Type is I4
2. `ub` Type is I4
3. `cnt` Type is I8

such that

1. $cnt > 0$
2. $lb < ub$
3. cnt is as evenly distributed as possible
4. n'_2 is as close to n_2 as possible
5. $\sum T_2[i].cnt = |T_1|$

Restrictions

1. $alldef(f_1)$
2. $srttype(f_1) = asc$
3. $fldtype(f_1) = I4$
4. Currently, we use a greedy algorithm that can result in the bin sizes being not as evenly distributed as one would like them to be.

3.30 t1f1t2f2opt3f3

Arguments are

1. T_1
2. f_1
3. T_2
4. f_2
5. $\circ$ operation to be performed
6. T_3
7. f_3

Creates field T_3 with single column f_3 of same type as input fields.
Restrictions

1. Supported operations for $\circ$ are
   (a) $A \cup B$ — union
   (b) $A \cap B$ — intersection
   (c) $A - B$ — `a_minus_b`
   (d) pvalcalc — Calculation of p-value
2. $srttype(f_1) = srttype(f_2) = asc$
3. $fldtype(f_1) = fldtype(f_2) \in \{I4, I8\}$
4. $alldef(f_1), alldef(f_2)$
5. Because of input restrictions, $fldtype(f_3) \leftarrow fldtype(f_1)$ and $srttype(f_3) = asc$

3.31 binld

Similar to bindmp in Section 3.5.2. Arguments are

1. T — table to be created
2. $f_1 : f_2 : \ldots f_N$ — list of fields separated by colon. Must have at least one field
3. $t_1 : t_2 : \ldots t_N$ — where t_i is specifier of field type for field f_i.
4. F — input file to be read
5. D — input directory from which file is to be read. If null, current working directory is used.

Creates table T from file F. Restrictions are

1. $fldtype(f_i) \in \{I1, I2, I4, I8, F4, F8\}$
2. $alldef(f_i)$

3.32 percentiles

Arguments are

1. T_I — input table
2. f — field in input table
3. T_O — output table
4. n — desired number of rows in output table

- Breaks input into n bins of equal size, last bin being potentially larger than others.
- Creates T_O, where every row corresponds to a bin, with 4 fields — min, max, avg, cnt.
  1. $min[i]$ is smallest value in bin i
  2. $max[i]$ is largest value in bin i
  3. $cnt[i]$ is number of values in bin i
  4. $avg[i]$ is average value in bin i

Restrictions

1. $fldtype(f) \in \{I4, I8, F4, F8\}$
2. $alldef(f)$
3. $srttype(f) = ascending$
4. $2 \leq n < min(|T|, 1000)$

3.33 rng_sort

Sorts f_1 in T_1 in batches specified by T_2

1. f_{lb} — lower bound field in T_2
2. f_{ub} — upper bound field in T_2
3. $mode$ — `A` for ascending, `D` for descending

The rows of T_2 denote non-overlapping ranges in T_1. These ranges are sorted independently.
Restrictions

1. $fldtype(f_1) \in \{I4, I8\}$
2. $fldtype(f_{lb}) = fldtype(f_{ub}) = I8$
3. $alldef(f_1), alldef(f_{lb}), alldef(f_{ub})$
4. $external(f_1) = false$
5. $0 < T_2[i].f_{lb} < T_2[i].f_{ub} \leq |T_1|$

3.34 app_tbl

Appends table T_2 to T_1. T_2 is untouched. Arguments are

1. T_1 — destination table
2. T_2 — source table

Restrictions are

1. Every field in T_1 must be present in T_2 and have same type in both T_1 and T_2
2. Allowable field types are `I1, I2, I4, I8, F4, F8`
3. All fields in T_1 are internal (not external)
4. Note that it is possible for $T_1 = T_2$

3.35 sortf1f2

Arguments are

1. T_1 — input table
2. f_1 — primary input field
3. f_2 — secondary input field
4. $srttype$ — see below

Restrictions are

1. $f_1 \neq f_2$
2. $alldef(f_1), internal(f_1)$
3. $alldef(f_2), internal(f_2)$
4. $fldtype(f_1) = \{I4, I8\}$
5. $fldtype(f_2) = \{I4, I8\}$
6. srttype must be one of
   (a) `A_` — f_1 primary ascending and f_2 drag along
   (b) `AA` — f_1 primary ascending and f_2 secondary ascending
   (c) `AD` — f_1 primary ascending and f_2 secondary descending
   (d) `D_` — f_1 primary descending and f_2 drag along
   (e) `DA` — f_1 primary descending and f_2 secondary ascending
   (f) `DD` — f_1 primary descending and f_2 secondary descending
7. Currently only `A_` and `D)` implemented

3.36 no_op

As expected, does nothing. Although it writes a line to the log file.

3.37 dld

Meta data file has 3 columns

1. field name. Must be alphanumeric.
2. field type. See Section 2.1, except for
   (a) `B`
   (b) `SC`
3. auxiliary information about the field. This is optional. It can contain the following
   (a) `is_load=[true|false]` Whether to load field or not. Default is true.
   (b) `is_all_def=[true|false]` Whether all values are defined or not. Default is false. If we discover that all values are defined, then we can drop the `nn` field that is created to mark undefined values.
   (c) `is_null_if_missing=[true|false]` This is used for fields that are lookups into an existing dictionary. Default is false.
      - When set to true, if we find a value that is not in the dictionary, we mark it as null.
      - When set to false, if we find a value that is not in the dictionary, we quit the dld operation.
      - `is_dict_old=[true|false]` Whether to use an existing directory or create a new one.

When `fldtype = SV`, it is converted into I4 and we use `dict_tbl_id` to denote the lookup table used to map ints to strings.

Invariant 2 $is\_load = false \Rightarrow dict = \bot$

Invariant 3 $is\_null\_if\_missing = true \Rightarrow is\_dict\_old = true$

Invariant 4 *Consider 2 fields being loaded in a given table. If both are going to create a new table, the tables must be different.* $is\_dict\_old_i = false \wedge is\_dict\_old_j = false \Rightarrow dict_i \neq dict_j$

3.37.1 Input Restrictions

1. When reading a CSV file, the length of a text field is limited to 32767 characters.

3.38 lkp_sort

Arguments are

1. T_1
2. f_1 — input field to be sorted
3. T_2 — table with information about number of each value of f_1

4. f_{cnt} — field in T_2
5. f_{idx} — newly created field in T_1 **OPTIONAL**. $fldtype(T_1.f_{idx}) = I8, alldef(T_1.f_{idx})$
6. f_{srt} — newly created field in T_1 $fldtype(T_1.f_{srt}) = I4, alldef(T_1.f_{srt})$

This is an ascending sort which does a one-pass sort provided one knows the distribution of the data. For example, let us say that we have a field f_{cnt} in table T_2 such that the i^{th} row tells us the number of values of f_1 that are $= i$. Then f_{srt} is a permutation of f_1 in sorted order ascending and f_{idx} is created as if we did a drag-along of the index field $0, 1, 2, \ldots$

Restrictions

1. $fldtype(f_{cnt}) = I8$
2. $fldtype(f1) = I8$
3. $alldef(f_{cnt})alldef(f_1)$
4. $0 \leq T_1.f_1 < |T_2|$

4 GPU Operations

As a convention, all GPU operations will start with `g_` We will use `D` to indicate device and `H` to indicate host.

I have not fully understood how to share the GPU across multiple concurrent Q invocations. Need to think through this further.

4.1 g_load

Arguments are

1. T — string, table
2. f_H — string, field
3. f_D — string, register name

Internally, this is translated to the following load operation

1. file name — string, name of binary file which contains data to be loaded
2. field type — string, Can be any of `B, I1, I2, I4, I8, F4, F8`
3. n_R. Number of rows. Uses for book-keeping.
4. r — register number.
5. over write or not — Could be
   (a) `N` means operation fails if register is in use or
   (b) `Y` means free register if in use and then perform operation

Above operation fails if any of the validations fail e.g.,

1. file not found
2. r not a valid registe number. Valid numbers are currently $[0, 31]$
3. Insufficient space on device
4. ...

4.2 g_info

Prints following meta data to stdout for each GPU register that is in use.

1. index
2. Size in bytes
3. File name from which it was loaded
4. Table name from which it was loaded
5. Field name from which it was loaded
6. Logical name provided to it by Q

5 Compound Expressions

The motivation behind compound statements is allowing users to take get the benefits of strip-mining. The user needs to issue the command `q start_compound` to denote that subsequent commands will be part of a compound expression. None of the subsequent commands are executed (they are simply recorded) until the user issues a `q stop_compound` statement.

At this time, they are first verified and then executed.

5.1 Verification

1. Only one table should be referenced. We will relax this gradually to allow operations like count which read from one table and write to another.
2. Given above limitation, the table referenced should be the same in each statement
3. operations should come from a limited set. They are either
   (a) regular e.g., `q <op> <tbl> .......`
      i. `s_to_f`
      ii. `f1opf2`
      iii. `f1s1opf2`
      iv. `f1f2opf3`
   (b) reduction e.g., `A='q <op> <tbl> .....'`. In this case, `A` is the destination of the reduction
      i. `f_to_s`

ii. `f1f2_to_s`

4. ephemeral fields are suffixed with the prime character i.e., `['].` ephemeral fields must be created before they are referenced

5. all non-ephemeral fields that are referenced must either exist prior to the start of the compound expression or must be created in a statement preceding the one in which it is referenced

6. reduction operations, if any, should come at end

7. the destinations of reduction operations should be unique

5.2 For some later point in time

A compound statement is of the form

```
x' = a + b; y' = (int)x'; z' = x' > y'; w = z' ? e : f
```

which is the same as Figure 2. Note that any variable with a prime is a temporary variable. Hence, in this example, a column called w is created but columns (x', y', z') are not created. Some limitations of a compound statement

1. All the assignments, except the last one, must be to temporary variables

2. The last assignment must be a to a permanent variable

3. Each assignment is separated by a semi-colon

4. The maximum number of assignments is 16. This is an arbitrary number, which can be increased when we are more comfortable.

for $i \leftarrow 0$ **to** $|T| - 1$ **do**
    $x' \leftarrow a[i] + b[i]$
    $y' \leftarrow (int)x'$
    if $x' > y'$ **then**
        $z' \leftarrow true$
    else
        $z' \leftarrow false$
    endif
    if $z' == true$ **then**
        $w[i] \leftarrow e[i]$
    else
        $w[i] \leftarrow f[i]$
    endif
endfor

Figure 2: Explanation of compound statement

Operation	I1	I2	I4	I8	F4	F8
+	✓	✓	✓	✓	✓	
−			✓	✓	✓	
*			✓	✓	✓	
/			✓	✓	✓	
%			✓	✓		
&	✓		✓	✓		
|	✓		✓	✓		
^	✓		✓	✓		
>	✓		✓	✓	✓	
<	✓		✓	✓	✓	
>=	✓		✓	✓	✓	
<=	✓		✓	✓	✓	
!=	✓		✓	✓	✓	
==	✓	✓	✓	✓	✓	
<<			✓	✓		
>>			✓	✓		
<=||>=			✓			
>&&<			✓			
>=&&<=			✓			

Table 3: Supported Operations for f1s1opf2

Operation	B	I1	I2	I4	I8
min		✓	✓	✓	✓
max		✓	✓	✓	✓
sum	✓	✓	✓	✓	✓
is_sorted		✓	✓	✓	✓
get_idx		✓	✓	✓	✓
approx_uq				✓	

Table 4: Supported Operations for f_to_s

Operation	B	I1	I2	I4	I8	F4	F8
conv	✓	✓		✓	✓	✓	✓
bitcount				✓	✓		
sqrt					✓		✓
abs							✓
reciprocal					✓		✓
!		✓		✓			
++				✓			
−−				✓			
~		✓		✓			
hash				✓	✓		
shift		✓	✓	✓	✓		
cum		✓	✓	✓	✓		
smear		✓					

Table 5: Supported Operations for f1opf2

From	To
B	I1
I1	B
I1	I4
I1	I8
I2	I4
I2	I8
I4	I1
I4	I2
I4	I8
I4	F4
I8	I1
I8	I4
I8	F4
F4	I4
F4	I8
F4	F8

Table 6: Supported conversions for f1opf2

Operation	B	I1	I2	I4	I8	F4	F8	Type(f_3)
concat				✓				I8
+			✓	✓	✓	✓		
--			✓	✓	✓	✓		
*				✓	✓	✓		
/				✓	✓	✓		
%				✓	✓			
&&	✓	✓						
||	✓	✓						
>				✓	✓	✓		I1
<				✓	✓	✓		I1
>=				✓	✓	✓		I1
<=				✓	✓	✓		I1
!=				✓	✓	✓		I1
==				✓	✓	✓		I1
&&!	✓				✓			
&				✓	✓			
|				✓	✓			
^				✓	✓			
<<				✓	✓			
>>				✓	✓			

Table 7: Supported Operations for f1f2opf3