1 Rate of Completions

As learners engage in activities supported by a learning ecosystem, they will build up a history of learning experiences. When the digital resources of that learning ecosystem adhere to a framework dedicated to supporting and understanding the learner, such as the Total Learning Architecture (TLA), the data produced by the learning ecosystem will contribute to each learner's digital footprint. One way that footprint can be made actionable is through analysis of trends and/or patterns of activity. The following Algorithm does exactly this but scoped to:

- events describing or asserting that a learner completed a learning activity or exercise.
- events which happened within some target window of time

1.1 Alignment to DAVE Algorithm Definition

The schema $RateOfCompletions$ serves as the first formal definition of an Algorithm which implements the definition of a DAVE Algorithm presented in the section Algorithm Formal Definition(4.6) on page 15. $RateOfCompletions$ is used to introduce the alignment between the generic components of $Algorithm$ and their corresponding definitions within this domain specific use case. In general, all DAVE Algorithm definitions must reference the schema $Algorithm$ and the schemas corresponding to the different components of $Algorithm$. Within $RateOfCompletions$, both $Algorithm.algorithm.algorithmIter$ and $ROCalgorithmIter$ are fully expanded for clarity. This is not a requirement of alignment schemas, but alignment schemas should feature:

- an expanded definition of the use case specific $algorithmIter$
- binding of the use case specific $algorithmIter$ to $Algorithm.algorithm.algorithmIter$

Typically, an alignment schema would be defined after its component schemas but because $RateOfCompletions$ is the first of its kind, it is featured first to introduce the notation by example and set the stage for the following component definitions. The alignments established in $RateOfCompletions$ are further expanded upon within the corresponding definition of each individual component.

1.1.1 Components

Within each component definition, in order to connect the dots between

- $Algorithm$ and its components
- $RateOfCompletions$ and its components

the symbol $\rightsquigarrow$ is used. This establishes that the constraints defined in the more generic component formal definitions apply to the schema being binded to. This is formalized within each of the $RateOfCompletions$ component schemas via

$$\begin{array}{|l}
genericSchema.primitiveName = \langle body \rangle \\
\hline
\langle body \rangle \rightsquigarrow localSchema.primitiveName = localSchema.primitiveChain
\end{array}$$

1.2 Formal Definition

The application of the notation described above to $RateOfCompletions$ results in the following definition with respect to schemas

$$\begin{aligned}
&RateOfCompletions ::= \\
&\qquad Algorithm \mathbin{;} RateOfCompletions \Rightarrow \\
&\qquad\qquad (Init \mathbin{;} RateOfCompletionsInit) \land \\
&\qquad\qquad (Relevant? \mathbin{;} RateOfCompletionsRelevant?) \land \\
&\qquad\qquad (Accept? \mathbin{;} RateOfCompletionsAccept?) \land \\
&\qquad\qquad (Step \mathbin{;} RateOfCompletionsStep) \land \\
&\qquad\qquad (Result \mathbin{;} RateOfCompletionsResult)
\end{aligned}$$

such that the $\langle body \rangle$ within each of the generic schema definitions is substituted for the Primitive chain defined within each of the local schemas. Here, the components of $RateOfCompletions$ use a naming scheme of Container + AlgorithmComponent but this pattern is not required. It is used here strictly for additional highlighting of the syntax introduced above for connecting the generic definition of an Algorithm to an Implementation of that methedology much like the concepts underlying Java Interfaces.

$$
\begin{array}{|l}
\hline
\Delta RateOfCompletions[KV, Collection, KV] \\
Algorithm \\
RateOfCompletionsInit \\
RateOfCompletionsRelevant? \\
RateOfCompletionsAccept? \\
RateOfCompletionsStep \\
RateOfCompletionsResult \\
rateOfCompletions\_ : KV \times Collection \times KV \twoheadrightarrow KV \\
state?, opt?, state! : KV \\
S? : Collection \\
\hline
Algorithm.algorithm.algorithmIter = \langle relevant?\_, accept?\_, step\_\rangle \\
ROCalgorithmIter = \langle RateOfCompletionsRelevant?.relevant?\_, \\
\qquad RateOfCompletionsAccept?.accept?\_, \\
\qquad RateOfCompletionsStep.step\_\rangle \\
\\
Algorithm.algorithm.algorithmIter\_ \rightsquigarrow ROCalgorithmIter\_ \Rightarrow \\
\quad (Algorithm.algorithm.algorithmIter.relevant?\_ \rightsquigarrow \\
\qquad RateOfCompletionsRelevant?.relevant?\_) \land \\
\quad (Algorithm.algorithm.algorithmIter.accept?\_ \rightsquigarrow \\
\qquad RateOfCompletionsAccept?.accept?\_) \land \\
\quad (Algorithm.algorithm.algorithmIter.step\_ \rightsquigarrow \\
\qquad RateOfCompletionsStep.step\_) \\
\\
state! = rateOfCompletions(state?, S?, opt?) \equiv algorithm(state?, S?, opt?) \iff \\
\qquad (Algorithm.algorithm.init\_ \rightsquigarrow RateOfCompletionsInit.init\_) \land \\
\qquad (Algorithm.algorithm.algorithmIter\_ \rightsquigarrow ROCalgorithmIter\_) \land \\
\qquad (Algorithm.algorithm.result\_ \rightsquigarrow RateOfCompletionsResult.result\_) \\
\hline
\end{array}
$$

- the . notation is used to reference components within a schema
- the $\rightsquigarrow$ represents alignment between components of $Algorithm$ and $RateOfCompletions$
- the Δ in the schema name indicates that $RateOfCompletions$ alters the state space of $Algorithm$ due to usage of $\rightsquigarrow$

1.3 Initialization

The first example of a component to component alignment is found within $RateOfCompletionsInit$ which shows how the primitive $RateOfCompletionsInit.init$ is bound to $\langle body \rangle$ within $Algorithm.algorithm.init$. Specifically, the schema $RateOfCompletionsInit$ uses the Primtive $updateAt$ such that $init_\delta$ can be used to establish the initialization logic.

1.3.1 Formal Definition

In the following, $init_\delta$ could have been a stand alone Operation referenced within $RateOfCompletionsInit$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{RateOfCompletionsInit[KV]} \\
Init, UpdateAt \\
state?\,, state! : KV \\
init\_ : KV \twoheadrightarrow KV \\
init_{\delta} : V \twoheadrightarrow KV \\
\hline
Init.init = \langle body \rangle \\
init = \langle updateAt\_ \rangle \\
Init.init \leadsto init \Rightarrow \langle body \rangle \equiv \langle updateAt\_ \rangle \\
\\
init_{\delta}! = init_{\delta}(state?_{\langle roc, completions \rangle}\,) \bullet \\
\qquad = \ (\emptyset \iff \langle roc, completions \rangle \notin state?\,) \lor \\
\qquad\quad (state?_{\langle roc, completions \rangle} \iff \langle roc, completions \rangle \in state?\,) \\
\\
state! = init(state?\,) = updateAt(state?\,, \langle roc, completions \rangle, init_{\delta}) \bullet \\
\qquad = (\langle\!\langle roc \mapsto completions \mapsto \emptyset \rangle\!\rangle \cup state? \iff init_{\delta}! = \emptyset) \lor \\
\qquad\quad (state? \iff init_{\delta}! \neq \emptyset) \\
\hline
\end{array}
$$

The output of $RateOfCompletionsInit.init$ is $state!$ which can be one of two things given the definition of $init_{\delta}$

- $state! = \langle\!\langle roc \mapsto completions \mapsto \emptyset \rangle\!\rangle \cup state?$
- $state! = state?$

This means that the result of any previous runs of $rateOfCompletions$ will not be overwritten but if this is the first iteration of the Algorithm, the necessary storage location is established within the Algorithm State such that

- $RateOfCompletionsStep.step$ can write its output to $state!_{\langle roc, completions \rangle}$
- $RateOfCompletionsResult.result$ can read from $state!_{\langle roc, completions \rangle}$

and by defining $RateOfCompletionsInit.init$ in this way, it allows for chaining of calls to $rateOfCompletion$ such that

- the Algorithm can pick back up from the result of a previous iteration
- Other Algorithms can use the result of $rateOfCompletions$ within their processing

which highlights the importance of establishing a unique path for individual Algorithms to write their results to. The example $path?$ of $\langle roc, completions \rangle$ is very simple but is sufficient for the current Algorithm. This $path?$ can be made more complex to suppport more advanced $init_{\delta}$ definitions. For example, each run of $rateOfCompletions$ could have its own unique subpath. In this scenario, $init_{\delta}$ could be updated to look for the most recent run of $rateOfCompletions$ and use it as the seed state for the current iteration among other things.

- $\langle roc, completions, run1 \rangle$
- $\langle roc, completions, run2 \rangle$

1.3.2 Big Picture

When Algorithms write to a unique location within an Algorithm State, high level Algorithms can be designed which chain together individual Algorithms such that the result of one is used to seed the next. Chaining together of Algorithms is a subject not yet covered within this report and its exact form is still under active development. It is mentioned here to highlight the ideal usage of Algorithm State in the context of *init*; Algorithm State is a mutable Map which serves as a storage location for a collection of Algorithm(s) to write to and/or read from such that an Algorithm can

- pick up from a previous iteration
- use the output of other Algorithm(s) to initialize the current state
- process quantities of data too large to store in local memory all at once

1.4 Relevant?

Given that the purpose of *relevant*? is to determine if the current Statement (*stmt*?) is valid for use within *step* of $rateOfCompletions$, the validation check itself can be implemented in several different ways but idealy, the predicate logic is expressed using the xAPI Profiles spec.

1.4.1 xAPI Profile Validation

The specification defines xAPI Statement Tempaltes which feature a built in xAPI property predicate language for defining the uniquely identifying properties of an xAPI Statement. These requirements are used within validation logic aligned to/based off of the Statement Template Validation Logic defined in the spec. The formal definition of Statement Template validation logic is outside the scope of this document but the following basic type is introduced to represent an xAPI Statement Tempalte

$$[TEMPLATE_{stmt}]$$

such that the following is an Operation definition for validation of an xAPI Statement *stmt*? against an xAPI Statement Template.

$$
\begin{array}{|l}
\hline
ValidateStatement[STATEMENT, TEMPLATE_{stmt}] \\
stmt? : STATEMENT \\
template? : TEMPLATE_{stmt} \\
validateStmt! : Boolean \\
validateStmt\_ : STATEMENT \times TEMPLATE_{stmt} \rightarrow Boolean \\
\hline
validateStmt! = validateStmt(stmt?, template?) = true \lor false \\
\hline
\end{array}
$$

This Operation can be composed with other xAPI Profile centered Operations to define more complex predicate/validation logic like:

- *stmt*? matches target xAPI Statement Template(s) defined within some xAPI Profile(s)
- *stmt*? matches pred (ie, any/none/etc.) xAPI Statement Template(s) defined within some xAPI Profile(s)
- *stmt*? matches target/pred xAPI Statement Template(s) within target/pred xAPI Pattern(s) defined within some xAPI Profile(s)

1.4.2 xAPI Predicates

In order to avoid brining in additional xAPI Profile complexity, the logic of $RateOfCompletionsRelevant?$ is implemented using predicates which correspond to checks which would happen during *validateStmt* given Statement Templates containing the following constraints.

- is the Object of the Statement an Activity?
- is the Verb indicative of a completion event?
- is Result.completion used to indicate completion?

In general, each of these Primitives navigates into a Statement to retrieve the value at a target *path*? and check it against the predicate defined in the schema. This generic functionality is defined as the Primitive *stmtPred*.

$$
\begin{array}{l}
\textit{StatementPredicate}[\textit{STATEMENT}, \textit{Collection}, (\_ \nrightarrow \_)] \\
\hline
\textit{GetIn} \\
\textit{stmt}? : \textit{STATEMENT} \\
\textit{path}? : \textit{Collection} \\
fn_{pred}! : \textit{Boolean} \\
fn_{pred}? : (\_ \nrightarrow \_) \\
\textit{stmtPred}\_ : \textit{STATEMENT} \times \textit{Collection} \times (\_ \nrightarrow \_) \rightarrow \textit{Boolean} \\
\hline
\textit{stmtPred} = \langle \textit{getIn}\_, fn_{pred}?\,(\textit{stmt}?_{\textit{path}?}) \rangle \\
\\
fn_{pred}! = \textit{stmtPred}(\textit{stmt}?, \textit{path}?, fn_{pred?}) \\
\qquad = fn_{pred}?\,(\textit{getIn}(\textit{stmt}?, \textit{path}?)) \bullet \\
\qquad = \textit{true} \vee \textit{false} \\
\hline
\end{array}
$$

This Primitive covers the most basic kind of check performed when validating an xAPI Statement against an xAPI Statement Template; does the Statement property found at $stmt?_{path?}$ adhere to the expectation(s) defined within the provided predicate. The next three schemas will define the statement predicates used within $RateOfCompletionsRelevant?$ but these predicates could have been contained within some number of xAPI Statement Template(s).

$$
\begin{array}{|l}
\hline
\textit{ActivityObject?}\,[STATEMENT] \\
\textit{StatementPredicate} \\
stmt? : STATEMENT \\
path? : Collection \\
fn_{pred}! : Boolean \\
fn_{pred\,\_} : V \rightarrow Boolean \\
activityObject?\,\_ : STATEMENT \rightarrow Boolean \\
\hline
activityObject? = \langle stmtPred\_\rangle \\
\\
path? = \langle object, objectType\rangle \\
\\
fn_{pred}! = activityObject?\,(stmt?\,) \\
\qquad = stmtPred(stmt?\,, path?\,, fn_{pred}) \\
\qquad = fn_{pred}(stmt?_{path?}\,) \\
\qquad = true \iff stmt?_{path?} = Activity \lor \emptyset \\
\hline
\end{array}
$$

- Determine if the Object of *stmt?* is an Activity

$$
\begin{array}{|l}
\hline
\textit{CompletionVerb?}\,[STATEMENT] \\
\textit{StatementPredicate} \\
stmt? : STATEMENT \\
path? : Collection \\
fn_{pred}! : Boolean \\
fn_{pred\,\_} : V \rightarrow Boolean \\
completionVerb?\,\_ : STATEMENT \rightarrow Boolean \\
\hline
completionVerb? = \langle stmtPred\_\rangle \\
\\
path? = \langle verb, id\rangle \\
\\
fn_{pred}! = completionVerb?\,(stmt?\,) \\
\qquad = stmtPred(stmt?\,, path?\,, fn_{pred}) \\
\qquad = fn_{pred}(stmt?_{path?}\,) \\
\qquad = true \iff stmt?_{path?} = \\
\qquad\qquad http : //adlnet.gov/expapi/verbs/passed \lor \\
\qquad\qquad https : //w3id.org/xapi/dod-isd/verbs/answered \lor \\
\qquad\qquad http : //adlnet.gov/expapi/verbs/completed \\
\hline
\end{array}
$$

- Determine if the Verb id within *stmt?* is one of
  - passed
  - answered
  - completed
- List of target Verb ids can be expanded as needed

$$
\begin{array}{|l}
\hline
\textit{CompletionResult?}\,[STATEMENT] \\
\textit{StatementPredicate} \\
stmt? : STATEMENT \\
path? : Collection \\
fn_{pred}! : Boolean \\
fn_{pred\,\_} : V \rightarrow Boolean \\
completionResult?\ \_ : STATEMENT \rightarrow Boolean \\
\hline
completionResult? = \langle stmtPred\_\rangle \\
\\
path? = \langle result, completion\rangle \\
\\
fn_{pred}! = completionResult?\,(stmt?) \\
\qquad\quad = stmtPred(stmt?\,, path?\,, fn_{pred}) \\
\qquad\quad = fn_{pred}(stmt?_{path?}) \\
\qquad\quad = true \iff stmt?_{path?} = true \\
\hline
\end{array}
$$

- Determine if completion is set to true within result field of an xAPI Statement

1.4.3 Formal Definition

The xAPI Predicates defined above are used within $RateOfCompletionsRelevant?$ to establish the logic which decides if $stmt?$ is

- passed on to the next step
- discarded for the next Statement in the batch passed to $rateOfCompletions$

$$
\begin{array}{|l}
\hline
\Xi RateOfCompletionsRelevant?\,[KV, STATEMENT] \\
Relevant? \\
state? : KV \\
stmt? : STATEMENT \\
relevant! : Boolean \\
relevant?\ \_ : KV \times STATEMENT \rightarrow Boolean \\
\hline
Relevant.relevant? = \langle body\rangle \\
\langle body\rangle \leadsto relevant? = \langle\, activityObject?\ \_\,, \langle\, completionVerb?\ \_\,,\ completionResult?\ \_\rangle\rangle \\
\\
relevant! = relevant?\,(state?\,, stmt?) \\
\qquad\quad = true \iff (activityObject(stmt?) = true) \wedge \\
\qquad\qquad\qquad\qquad ((completionVerb?\,(stmt?) = true) \vee \\
\qquad\qquad\qquad\qquad (completionResult?\,(stmt?) = true)) \\
\hline
\end{array}
$$

The schema prefix Ξ is used to indicate that here, $relevant?$ does not modify $state?$. Regardless, in order for $relevant?$ to return true

- The object of *stmt*? must be an activity
- The Verb of *stmt*? has an id which matches one of the target IDs
- The Result of *stmt* indicates that a completion happened

1.5 Accept?

The *Accept*? component of a DAVE Algorithm is a secondary validation check prior to the potential passing of *stmt*? off to *Step*. At this point, *stmt*? has been validated to be relevant to the execution of an Algorithm so the final check is based off of the current Algorithm State *state*?. In many cases this check will not be necessary but this step matters when the ability to process *stmt*? is dependent upon some property of *state*?. This component of an Algorithm could be used to establish the placeholder mapping within *state*! if it doesn't exist for the current *stmt*? but this can also be handled within *step* as done in the schema *ProcessCompletionStatement* defined later on.

$$
\begin{array}{|l}
\hline
\Xi RateOfCompletionsAccept?\,[KV, STATEMENT] \\
Accept? \\
state? : KV \\
stmt? : STATEMENT \\
accept! : Boolean \\
fn_{pred} : KV \times STATEMENT \rightarrow Boolean \\
accept?\ \_ : KV \times STATEMENT \rightarrow Boolean \\
\hline
Accept?.accept? = \langle body \rangle \\
accept? = \langle fn_{pred}\ \_ \rangle \\
Accept?.accept? \rightsquigarrow accept? \Rightarrow \langle body \rangle \equiv \langle fn_{pred}\ \_ \rangle \\
\\
accept! = accept?\,(state?, stmt?) \\
\qquad\quad = fn_{pred}(state?, stmt?) = true \\
\hline
\end{array}
$$

The Algorithm *rateOfCompletions* does not need to check *state*? before passing *stmt*? to *step* so fn_{pred} will always return true. If this was not the case, fn_{pred} would be defined as a predicate describing the relationship between *state*? and *stmt*? which determines if true or false is returned. Additionally, if false would be returned, *Accept* can take the appropriate steps to ensure *state*! is updated such that $accept?\,(state!, stmt?) = true$.

1.6 Step

The actual processing of *stmt*? happens within *step* and may or may not result in an updated Algorithm State *state*!. In the case of *rateOfCompletions*, each call to *step* is expected to return an altered state such that $state! \neq state?$ and the schema *RateOfCompletionsStep* is prefixed with Δ accordingly. The updated *state*! will either have an existing mapping for $objectId \in state?$ altered or a completely new mapping for *objectId* added to *state*?.

1.6.1 Processing Summary

The execution of $step$ can be summarized as:

1. parse the relevant information from $stmt$
   - $currentTime$
   - $objectName$
   - $objectId$
2. resolve previous state (if it exists) given $objectId$
3. update the range of time to include $currentTime$ if not already within the existing interval for $objectId$
4. update the counter tracking the number of times $objectId$ has been in a $stmt?$ passed to $step$
5. add $objectName$ to the set of names associated with $objectId$ if not already a member.

1.6.2 Helper Functions

The following Operations and Primitives are defined for abstracting the functionality of each process within $step$ in order to reduce the noise within $RateOfCompletionsStep$.

$$
\begin{array}{l}
\textit{ParseCompletionStatement}[STATEMENT] \\
\hline
GetIn \\
stmt? : STATEMENT \\
currentTime : TIMESTAMP \\
objectName, parseStmt! : KV \\
objectId : STRING \\
parseStmt\_ : STATEMENT \rightarrowtail KV \\
\hline
parseStmt = \langle getIn\_,\ associate\_\rangle^2 \\
\\
currentTime = getIn(stmt?, \langle timestamp\rangle) \\
objectName = getIn(stmt?, \langle object, definition, name\rangle) \\
objectId = getIn(stmt?, \langle object, id\rangle) \\
\\
parseStmt! = parseStmt(stmt?) \bullet \\
\quad let\ \ withTime == associate(\langle\!\langle\rangle\!\rangle, currentT, currentTime) \\
\qquad withName == associate(withTime, objName, objectName) \\
\qquad\quad = associate(withName, objId, objectId) \Rightarrow \\
\quad \langle\!\langle currentT \mapsto currentTime,\ objName \mapsto objectName,\ objId \mapsto objectId\rangle\!\rangle \\
\hline
\end{array}
$$

- parse timestamp, object name and object id from $stmt?$

$$
\begin{array}{l}
\text{---}\ ResolvePreviousCompletionState[KV, KV]\ \text{---}\\
GetIn \\
state?, parsed?, prevState! : KV \\
getPreviousState\_ : KV \times KV \rightarrow\!\!\!\!\!+ KV \\
\hline
getPreviousState = \langle getIn\_,\ getIn\_ \rangle \\
objectId = getIn(parsed?, objId) \\
prevState! = getPreviousState(state?, parsed?) = getIn(state?, \langle roc, completions, objectId \rangle)
\end{array}
$$

- look in $state?$ for any previous record of $objectId$

$$
\begin{array}{l}
\text{---}\ IntervalValGiven[TIMESTAMP, TIMESTAMP(\_ \rightarrow\!\!\!\!\!+ \_)]\ \text{---}\\
IsoToUnix \\
stmt_{ts}, state_{ts}, intervalValGiven! : TIMESTAMP \\
fn_{pred} : (\_ \rightarrow\!\!\!\!\!+ \_) \\
fn_{pred}! : \mathbb{N} \\
intervalValGiven\_ : TIMESTAMP \times TIMESTAMP \times (\_ \rightarrow\!\!\!\!\!+ \_) \rightarrow\!\!\!\!\!+ TIMESTAMP \\
\hline
intervalValGiven = \langle isoToUnix\_,\ isoToUnix\_,\ fn_{pred}\_ \rangle \\
\\
nSeconds_{stmt} = isoToUnix(stmt_{ts}) \\
nSeconds_{state} = isoToUnix(state_{ts}) \\
fn_{pred}! = fn_{pred}(nSeconds_{stmt}, nSeconds_{state}) \\
intervalValGiven! = intervalValGiven(stmt_{ts}, state_{ts}, fn_{pred}) \\
\qquad = (stmt_{ts} \iff fn_{pred}! = nSeconds_{stmt}) \lor \\
\qquad\quad (state_{ts} \iff fn_{pred}! = nSeconds_{state})
\end{array}
$$

- return $stmt_{ts}$ or $state_{ts}$ based on result of fn_{pred}

$$
\begin{array}{l}
\text{---}\ ReturnIntervalStart[TIMESTAMP, TIMESTAMP]\ \text{---}\\
IntervalValGiven \\
stmt_{ts}, state_{ts}, interval_{start} : TIMESTAMP \\
fn_{\delta}! : \mathbb{N} \\
fn_{\delta} : \mathbb{N} \times \mathbb{N} \rightarrow\!\!\!\!\!+ \mathbb{N} \\
returnIntervalStart\_ : TIMESTAMP \times TIMESTAMP \rightarrow\!\!\!\!\!+ TIMESTAMP \\
\hline
returnIntervalStart = \langle intervalValGiven\_ \rangle \\
\\
fn_{\delta}! = fn_{\delta}(nSeconds_{stmt}, nSeconds_{state}) \\
\qquad = (nSeconds_{stmt} \iff nSeconds_{stmt} \leq nSeconds_{state}) \lor \\
\qquad\quad (nSeconds_{state} \iff nSeconds_{stmt} > nSeconds_{state}) \\
interval_{start} = intervalValGiven(stmt_{ts}, state_{ts}, fn_{\delta}) \\
\qquad = (stmt_{ts} \iff fn_{\delta}! = nSeconds_{stmt}) \lor \\
\qquad\quad (state_{ts} \iff fn_{\delta}! = nSeconds_{state})
\end{array}
$$

- return $stmt_{ts}$ or $state_{ts}$, which ever one is further back in the past.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ReturnIntervalEnd[TIMESTAMP, TIMESTAMP] \qquad\qquad}} \\
IntervalValGiven \\
stmt_{ts}, state_{ts}, interval_{end} : TIMESTAMP \\
fn_{\delta}! : \mathbb{N} \\
fn_{\delta} : \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N} \\
returnIntervalEnd\_ : TIMESTAMP \times TIMESTAMP \nrightarrow TIMESTAMP \\
\hline
returnIntervalEnd = \langle intervalValGiven\_ \rangle \\
\\
fn_{\delta}! = fn_{\delta}(nSeconds_{stmt}, nSeconds_{state}) \\
\qquad = (nSeconds_{stmt} \iff nSeconds_{stmt} \geq nSeconds_{state}) \lor \\
\qquad\quad (nSeconds_{state} \iff nSeconds_{stmt} < nSeconds_{state}) \\
\\
interval_{end} = intervalValGiven(stmt_{ts}, state_{ts}, fn_{\delta}) \\
\qquad\qquad = (stmt_{ts} \iff fn_{\delta}! = nSeconds_{stmt}) \lor \\
\qquad\qquad\quad (state_{ts} \iff fn_{\delta}! = nSeconds_{state}) \\
\hline
\end{array}
$$

- return $stmt_{ts}$ or $state_{ts}$, which ever one is later on chronologically

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ReturnUpdatedCount[V] \qquad\qquad}} \\
count? : V \\
count! : \mathbb{N} \\
returnUpdatedCount\_ : V \rightarrow \mathbb{N} \\
\hline
count! = returnUpdatedCount(count?) \\
\qquad = (count? + 1 \iff count? \neq \emptyset) \lor \\
\qquad\quad (1 \iff (count? = 0) \lor (count? = \emptyset)) \\
\hline
\end{array}
$$

- return an incremented value or 1 otherwise

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ReturnUpdatedNames[Collection, STRING] \qquad\qquad}} \\
names?, names! : Collection \\
targetName : STRING \\
returnUpdatedNames\_ : Collection \times STRING \twoheadrightarrow Collection \\
\hline
names?_{targetName} = names? \upharpoonright targetName \\
names! = returnUpdatedNames(names?, targetName) \\
\qquad = (names? \frown targetName \iff names?_{targetName} = \emptyset \Rightarrow targetName \notin names?) \lor \\
\qquad\quad (names? \iff names?_{targetName} \neq emptyset \Rightarrow targetName \in names?) \\
\hline
\end{array}
$$

- add $targetName$ to the end of $names?$ if $targetName \notin names?$

1.6.3 Formal Definition

The schema $ProcessCompletionStatement$ is used to define the core functionality of $RateOfCompletionsStep.step$ using the Primitive $replaceAt$ to produce $state!$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Delta ProcessCompletionStatement[STATEMENT, KV]} \\
\hline
ReplaceAt \\
ParseCompletionStatement \\
ResolvePreviousCompletionState \\
ReturnIntervalStart \\
ReturnIntervalEnd \\
ReturnUpdatedCount \\
ReturnUpdatedNames \\
stmt? : STATEMENT \\
state?, state!, state_{objectId} : KV \\
processStatement\_ : STATEMENT \times KV \rightarrowtail\!\!\!\!\rightarrow KV \\
\hline
processStatement = \langle\langle parseStmt\_, getPreviousState\_\rangle, \\
\qquad\qquad\qquad\qquad \langle returnIntervalStart\_, returnIntervalEnd\_, replaceAt\_\rangle, \\
\qquad\qquad\qquad\qquad \langle returnUpdatedCount\_, replaceAt\_\rangle, \\
\qquad\qquad\qquad\qquad \langle returnUpdatedNames\_, replaceAt\_\rangle\rangle \\
\\
parsed_{stmt?} = parseStmt(stmt?) \\
stmt_{timestamp} = get(parsed_{stmt?}, currentT) \\
stmt_{objName} = get(parsed_{stmt?}, objName) \\
stmt_{objId} = get(parsed_{stmt?}, objId) \\
\\
state_{objectId} = getPreviousState(state?, parsed_{stmt?}) \\
interval_{start} = getIn(state_{objectId}, \langle domain, start\rangle) \\
interval_{end} = getIn(state_{objectId}, \langle domain, end\rangle) \\
state_{nStmts} = get(state_{objectId}, nStmts) \\
state_{names} = get(state_{objectId}, names) \\
\\
interval_{start}! = returnIntervalStart(stmt_{timestamp}, interval_{start}) \\
interval_{end}! = returnIntervalEnd(stmt_{timestamp}, interval_{end}) \\
interval! = \langle\!\langle start \mapsto interval_{start}!, end \mapsto interval_{end}! \rangle\!\rangle \\
\\
nStmts! = returnUpdatedCount(state_{nStmts}) \\
names! = returnUpdatedNames(state_{names}, stmt_{objName}) \\
\\
state! = processStatement(stmt?, state?) \bullet \\
\quad let\ \ interval_{\delta} == replaceAt(state?, \langle roc, completions, stmt_{objId}, domain\rangle, interval!) \\
\qquad nStmts_{\delta} == replaceAt(interval_{\delta}, \langle roc, completions, stmt_{objId}, nStmts\rangle, nStmts!) \\
\qquad\ = replaceAt(nStmts_{\delta}, \langle roc, completions, stmt_{objId}, names\rangle, names!) \\
\hline
\end{array}
$$

- update *state*! to include a mapping with Key $stmt_{objId}$ or update an existing mapping identified by $stmt_{objId}$

The schema *RateOfCompletionsStep* introduces the alignment with *Algorithm.step* such that $\langle body\rangle = processStatement$ as defined by *ProcessCompletionStatement*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Delta RateOfCompletionsStep[KV, STATEMENT]} \\
Step \\
ProcessCompletionStatement \\
state?, state! : KV \\
stmt? : STATEMENT \\
step\_ : KV \times STATEMENT \twoheadrightarrow KV \\
\hline
Step.step = \langle body \rangle \\
step = \langle processStatement\_ \rangle \\
Step.step \rightsquigarrow step \Rightarrow \\
\qquad \langle body \rangle \equiv \langle \langle parseStmt\_, getPreviousState\_ \rangle, \\
\qquad\qquad \langle returnIntervalStart\_, returnIntervalEnd\_, replaceAt\_ \rangle, \\
\qquad\qquad \langle returnUpdatedCount\_, replaceAt\_ \rangle, \\
\qquad\qquad \langle returnUpdatedNames\_, replaceAt\_ \rangle \rangle \\
\\
state! = step(state?, stmt?) = processStatement(stmt?, state?) \bullet \\
\quad state! \neq state? \land \\
\quad getIn(state!, \langle roc, completions, stmt_{objId} \rangle) \neq \emptyset
\end{array}
$$

For each unique $stmt_{objId}$ passed to $step$, there should be a corresponding mapping in $state_{\langle roc, completions \rangle}$ which looks like

$$
\begin{aligned}
stmt_{objId} \mapsto \langle\!\langle & domain \mapsto \langle\!\langle (start, interval_{start}!), (end, interval_{end}!) \rangle\!\rangle \\
& nStmts \mapsto nStmts! \\
& names \mapsto names! \rangle\!\rangle
\end{aligned}
$$

1.7 Result

The interval of $interval_{start}!$ to $interval_{end}!$ can be partitioned based on the passed in opt named $timeUnit$ such that for each unique $stmt_{objId}$, the metric n completions per time unit can be calculated and added to $state_{\langle roc, completions, stmt_{objId}, rate \rangle}$.

$$
\begin{array}{l}
\underline{\quad RateOfCompletionsResult[KV, KV] \quad} \\
Result \\
RateOf \\
replaceAt \\
opt?, state?, result! : KV \\
result\_ : KV \times KV \twoheadrightarrow KV \\
\hline
Result.result = \langle body \rangle \\
\langle body \rangle \leadsto result = \langle replaceAt\_ \rangle^{\# \operatorname{dom}(state_{\langle roc, completions \rangle})} \\
\\
state! = result(state?, opt?) \bullet \\
\qquad\qquad let\ timeUnit == atKey(opt?, timeUnit) \bullet \\
\qquad \forall stmt_{objId} \in state_{\langle roc, completions \rangle} \mid \exists_1 state!_{objId} \bullet \\
\qquad\qquad\qquad let\ \ nO == getIn(state?, \langle roc, completions, stmt_{objId}, nStmts \rangle) \\
\qquad\qquad\qquad\qquad start == getIn(state?, \langle roc, completions, stmt_{objId}, domain, start \rangle) \\
\qquad\qquad\qquad\qquad end == getIn(state?, \langle roc, completions, stmt_{objId}, domain, end \rangle) \\
\qquad\qquad\qquad\qquad rate_{objId} == rateOf(nO, start, end, timeUnit) \\
\qquad\qquad state!_{objId} = replaceAt(state?, \langle roc, completions, stmt_{objId}, rate \rangle, rate_{objId}) \\
\qquad = \sum state!_{objId} \bullet \forall stmt_{objId} \mid getIn(state!, \langle roc, completions, stmt_{objId}, rate \rangle) \in \mathbb{Z} \\
\hline
\end{array}
$$

- $state! = \sum state!_{objId}$ is an alternative way to describe the aggregation of all changes made to $state?$.
- $rateOf$ performs the calculation which is used to update $state?$ with consideration to $opt?$
- $timeUnit$ will default to day if not specified within $opt?$

The output of $rateOfCompletions$ is a $state!$ which contains a mapping of following shape for each unique $stmt_{objId}$ passed to $rateOfCompletions$, each of which can be found at the path $\langle roc, completions, stmt_{objId} \rangle$.

$$
\begin{array}{l}
stmt_{objId} \mapsto \langle\!\langle domain \mapsto \langle\!\langle (start, interval_{start}!), (end, interval_{end}!) \rangle\!\rangle \\
\qquad\qquad\quad nStmts \mapsto nStmts! \\
\qquad\qquad\quad names \mapsto names! \\
\qquad\qquad\quad rate \mapsto rate_{objId} \rangle\!\rangle
\end{array}
$$

Any mapping within this structure can be used within a corresponding visualization but the core piece of information the visualization should convey is $rate_{objId}$.

1.8 Conclusion

This concludes the first example of a DAVE Algorithm formal definition. The conventions established within this section should be used across all DAVE Algorithm formal definitions in this document. If any aspect of this section requires further explanation or clarification, please post an issue to the DAVE github repo describing the issue or reach out to the Author(s) of this report via some other medium.