Data Analytics and Visualization Environment for xAPI and the Total Learning Architecture: DAVE Learning Analytics Algorithms

Yet Analytics

September 16, 2019

Introduction

This report introduces the updated definition of learning analytics algorithms in terms of **Operations**, **Primitives** and **Algorithms** and presents an updated definition for each of the previously defined algorithms. The previous definitions will be included for reference. In a more general sense, this report establishes a set of style guidelines for the reporting of algorithms and associated visualization templates.

This document will be updated to include additional Operations, Primitives and Algorithms as they are defined by the Author of this report or members of the Open Source Community. Updates may also address refinement of existing definitions and this document should be understood to be an example of algorithm presentation and not the final state of any defined algorithm.

The structure of this documents is as follows:

1. A formal specification for xAPI written in Z
2. An Introduction to Terminology of Operations, Primitives and Algorithms
3. What is an Operation
4. What is a Primitive
5. What is an Algorithm
6. Foundational Operations
7. Example Primitives
8. An algorithm definition including
   (a) Init
   (b) Relevant?
   (c) Accept?
   (d) Step
   (e) Result
9. Previous Algorithm definitions where each consists of
   (a) an introduction for the algorithm
   (b) the structure of the ideal input data
   (c) how to retrieve input data from an LRS
   (d) the statement parameters which the algorithm will utilize
   (e) notices regarding data collected during the 2018 pilot test of the TLA
   (f) a summary of the algorithm

(g) the formal specification of the algorithm

(h) pseudocode representation of the algorithm

(i) JSONSchema for the output of the algorithm

(j) a description of the associated visualization

(k) a prototype of the visualization

(l) a collection of suggestions describing how the algorithm could be adapted to improve the quality of the visualization prototype

1 xAPI Formal Specification

The current formal specification only defines xAPI statements abstractly within the context of Z. A concrete definition for xAPI statements is outside the scope of this document.

1.1 Basic Types

$IFI ::= mbox \mid mbox\_sha1sum \mid openid \mid account$

- Type unique to Agents and Groups, The concrete definition of the listed values is outside the scope of this specification

$OBJECTTYPE ::= Agent \mid Group \mid SubStatement \mid StatementRef \mid Activity$

- A type which can be present in all activities as defined by the xAPI specification

$INTERACTIONTYPE ::= true-false \mid choice \mid fill-in \mid long-fill-in \mid matching \mid performance \mid sequencing \mid likert \mid numeric \mid other$

- A type which represents the possible interactionTypes as defined within the xAPI specification

$INTERACTIONCOMPONENT ::= choices \mid scale \mid source \mid target \mid steps$

- A type which represents the possible interaction components as defined within the xAPI specification
- the concrete definition of the listed values is outside the scope of this specification

$CONTEXTTYPES ::= parent \mid grouping \mid category \mid other$

- A type which represents the possible context types as defined within the xAPI specification

$[STATEMENT]$

- Basic type for an xAPI data point

$[AGENT, GROUP]$

- Basic types for Agents and collections of Agents

1.2 Id Schema

$$\begin{array}{|l}
\textit{Id} \\
\hline
id : \mathbb{F}_1 \#1 \\
\hline
\end{array}$$

- the schema Id introduces the component id which is a non-empty, finite set of 1 value

1.3 Schemas for Agents, Groups and Actors

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Agent}} \\
\hline
agent : AGENT \\
objectType : OBJECTTYPE \\
name : \mathbb{F}_1 \, \#1 \\
ifi : IFI \\
\hline
objectType = Agent \\
agent = \{ifi\} \cup \mathbb{P}\{name, objectType\} \\
\hline
\end{array}
$$

- The schema *Agent* introduces the component *agent* which is a set consisting of an ifi and optionally an *objectType* and/or *name*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Member}} \\
\hline
Agent \\
member : \mathbb{F}_1 \\
\hline
member = \{a : AGENT \mid \forall a_n : a_i..a_j \bullet i \leq n \leq j \bullet a = agent\} \\
\hline
\end{array}
$$

- The schema *Member* introduces the component *member* which is a set of objects a, where for every a within $a_0..a_n$, a is an *agent*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Group}} \\
\hline
Member \\
group : GROUP \\
objectType : OBJECTTYPE \\
ifi : IFI \\
name : \mathbb{F}_1 \, \#1 \\
\hline
objectType = Group \\
group = \{objectType, name, member\} \vee \{objectType, member\} \vee \\
\qquad \{objectType, ifi\} \cup \mathbb{P}\{name, member\} \\
\hline
\end{array}
$$

- The schema *Group* introduces the component *group* which is of type *GROUP* and is a set of either *objectType* and *member* with optionaly *name* or *objectType* and ifi with optionally *name* and/or *member*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Actor}} \\
\hline
Agent \\
Group \\
actor : AGENT \vee GROUP \\
\hline
actor = agent \vee group \\
\hline
\end{array}
$$

- The schema *Actor* introduces the component *actor* which is either an *agent* or *group*

1.4 Verb Schema

$$\begin{array}{|l}
\hline \textit{Verb} \\
Id \\
display, verb : \mathbb{F}_1 \\
\hline
verb = \{id, display\} \lor \{id\} \\
\hline
\end{array}$$

- The schema $Verb$ introduces the component $verb$ which is a set that consists of either id and the non-empty, finite set $display$ or just id

1.5 Object Schema

$$\begin{array}{|l}
\hline \textit{Extensions} \\
extensions, extensionVal : \mathbb{F}_1 \\
extensionId : \mathbb{F}_1 \, \#1 \\
\hline
extensions = \{e : (extensionId, extensionVal) \ | \forall e_n : e_i..e_j \bullet i \leq n \leq j \bullet \\
\qquad (extensionId_i, extensionVal_i) \lor (extensionId_i, extensionVal_j) \land \\
\qquad (extensionId_j, extensionVal_i) \lor (extensionId_j, extensionVal_j) \land \\
\qquad extensionId_i \neq extensionId_j\} \\
\hline
\end{array}$$

- The schema $Extensions$ introduces the component $extensions$ which is a non-empty, finite set that consists of ordered pairs of $extensionId$ and $extensionVal$. Different $extensionId$s can have the same $extensionVal$ but there can not be two identical $extensionId$ values
- $extensionId$ is a non-empty, finite set with one value
- $extensionVal$ is a non-empty, finite set

$$\begin{array}{|l}
\hline \textit{InteractionActivity} \\
interactionType : INTERACTIONTYPE \\
correctResponsePattern : \mathrm{seq}_1 \\
interactionComponent : INTERACTIONCOMPONENT \\
\hline
interactionActivity = \{interactionType, correctReponsePattern, interactionComponent\} \lor \\
\qquad \{interactionType, correctResponsePattern\} \\
\hline
\end{array}$$

- The schema $InteractionActivity$ introduces the component $interactionActivity$ which is a set of either $interactionType$ and $correctResponsePattern$ or $interactionType$ and $correctResponsePattern$ and $interactionComponent$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ Definition\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
InteractionActivity \\
Extensions \\
definition, name, description : \mathbb{F}_1 \\
type, moreInfo : \mathbb{F}_1\ \#1 \\
\hline
definition = \mathbb{P}_1\{name, description, type, moreInfo, extensions, interactionActivity\} \\
\hline
\end{array}
$$

- The schema $Definition$ introduces the component $definition$ which is the non-empty, finite power set of $name$, $description$, $type$, $moreInfo$ and $extensions$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ Object\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
Id \\
Definition \\
Agent \\
Group \\
Statement \\
objectTypeA, objectTypeS, objectTypeSub, objectType : OBJECTTYPE \\
substatement : STATEMENT \\
object : \mathbb{F}_1 \\
\hline
substatement = statement \\
objectTypeA = Activity \\
objectTypeS = StatementRef \\
objectTypeSub = SubStatement \\
objectType = objectTypeA \vee objectTypeS \\
object = \{id\} \vee \{id, objectType\} \vee \{id, objectTypeA, definition\} \\
\qquad\qquad \vee\{id, definition\} \vee \{agent\} \vee \{group\} \vee \{objectTypeSub, substatement\} \\
\qquad\qquad \vee\{id, objectTypeA\} \\
\hline
\end{array}
$$

- The schema $Object$ introduces the component $object$ which is a non-empty, finite set of either id, id and $objectType$, id and $objectTypeA$, id and $objectTypeA$ and $definition$, $agent$, $group$, or $substatement$
- The schema $Statement$ and the corresponding component $statement$ will be defined later on in this specification

1.6 Result Schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Score}} \\ \hline
score : \mathbb{F}_1 \\
scaled, min, max, raw : \mathbb{Z} \\ \hline
scaled = \{n : \mathbb{Z} \mid -1.0 \leq n \leq 1.0\} \\
min = n < max \\
max = n > min \\
raw = \{n : \mathbb{Z} \mid min \leq n \leq max\} \\
score = \mathbb{P}_1\{scaled, raw, min, max\} \\ \hline
\end{array}$$

- The schema *Score* introduces the component *score* which is the non-empty powerset of *min*, *max*, *raw* and *scaled*

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Result}} \\ \hline
Score \\
Extensions \\
success, completion, response, duration : \mathbb{F}_1 \#1 \\
result : \mathbb{F}_1 \\ \hline
success = \{true\} \vee \{false\} \\
completion = \{true\} \vee \{false\} \\
result = \mathbb{P}_1\{score, success, completion, response, duration, extensions\} \\ \hline
\end{array}$$

- The schema *Result* introduces the component *result* which is the non-empty power set of *score*, *success*, *completion*, *response*, *duration* and *extensions*

1.7 Context Schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Instructor}} \\ \hline
Agent \\
Group \\
instructor : AGENT \vee GROUP \\ \hline
instructor = agent \vee group \\ \hline
\end{array}$$

- The schema *Instructor* introduces the component *instructor* which can be ether an *agent* or a *group*

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Team}} \\ \hline
Group \\
team : GROUP \\ \hline
team = group \\ \hline
\end{array}$$

- The schema $Team$ introduces the component $team$ which is a $group$

$$
\begin{array}{|l}
\hline
\textit{Context} \\
Instructor \\
Team \\
Object \\
Extensions \\
registration, revision, platform, language : \mathbb{F}_1 \# 1 \\
parentT, groupingT, categoryT, otherT : CONTEXTTYPES \\
contextActivities, statement : \mathbb{F}_1 \\
\hline
statement = object \setminus (id, objectType, agent, group, definition) \\
parentT = parent \\
groupingT = grouping \\
categoryT = category \\
otherT = other \\
contextActivity = \{ca : object \setminus (agent, group, objectType, objectTypeSub, substatement)\} \\
contextActivityParent = (parentT, contextActivity) \\
contextActivityCategory = (categoryT, contextActivity) \\
contextActivityGrouping = (groupingT, contextActivity) \\
contextActivityOther = (otherT, contextActivity) \\
contextActivities = \mathbb{P}_1\{contextActivityParent, contextActivityCategory, \\
\qquad contextActivityGrouping, contextActivityOther\} \\
context = \mathbb{P}_1\{registration, instructor, team, contextActivities, revision, \\
\qquad platform, language, statement, extensions\} \\
\hline
\end{array}
$$

- The schema $Context$ introduces the component $context$ which is the non-empty powerset of $registration$, $instructor$, $team$, $contextActivities$, $revision$, $platform$, $language$, $statement$ and $extensions$
- The notation $object \setminus agent$ represents the component $object$ except for its subcomponent $agent$

1.8 Timestamp and Stored Schema

$$
\begin{array}{|l}
\hline
\textit{Timestamp} \\
timestamp : \mathbb{F}_1 \# 1 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Stored} \\
stored : \mathbb{F}_1 \# 1 \\
\hline
\end{array}
$$

- The schema $Timestamp$ and $stored$ introduce the components $timestamp$ and $stored$ respectively. Each are non-empty, finite sets containing one value

1.9 Attachements Schema

$$
\begin{array}{|l}
\hline
\textit{Attachments} \\
display, description, attachment, attachments : \mathbb{F}_1 \\
usageType, sha2, fileUrl, contexntType : \mathbb{F}_1 \, \#1 \\
length : \mathbb{N} \\
\hline
attachment = \{usageType, display, contentType, length, sha2\} \cup \mathbb{P}\{description, fileUrl\} \\
attachments = \{a : attachment\} \\
\hline
\end{array}
$$

- The schema *Attachements* introduces the componenet *attachements* which is a non-empty, finite set of the component *attachement*
- The component *attachment* is a non-empty, finite set of the componenets *usageType*, *display*, *contentType*, *length*, *sha*2 with optionally *description* and/or *fileUrl*

1.10 Statement and Statements Schema

$$
\begin{array}{|l}
\hline
\textit{Statement} \\
Id \\
Actor \\
Verb \\
Object \\
Result \\
Context \\
Timestamp \\
Stored \\
Attachements \\
statement : STATEMENT \\
\hline
statement = \{actor, verb, object, stored\} \cup \\
\qquad \mathbb{P}\{\mathrm{id}, result, context, timestamp, attachments\} \\
\hline
\end{array}
$$

- The schema *Statement* introduces the component *statement* which consists of the components *actor*, *verb*, *object* and *stored* and the optional components *id*, *result*, *context*, *timestamp*, and/or *attachments*
- The schema *Statement* allows for subcomponent of *statement* to refrenced via the . (selection) operator

$$
\begin{array}{|l}
\hline
\textit{Statements} \\
Statement \\
IsoToUnix \\
statements : \mathbb{F}_1 \\
\hline
statements = \{s : statement \mid \forall s_n : s_i..s_j \bullet i \leq n \leq j \\
\qquad \bullet convert(s_i.timestamp) \leq convert(s_j.timestamp)\} \\
\hline
\end{array}
$$

- The schema *Statements* introduces the component *statements* which is a non-empty, finite set of the component *statement* which are in chronological order.

2 Operations, Primitives and Algorithms

The following sections introduce, define and explain Operations, Primitives and Algorithms generally using the Terminology presented below. Operations are the building blocks of Primitives whereas Primitives are the building blocks of Algorithms. The definitions which follow are flexible enough to support implementation across programing languages but have been inspired by the core concepts found within Lisp. The focus of these sections is to define the properties of and interactions between Operations, Primitives and Algorithms in a general way which doesn't place unnecessary bounds on their range of possible functionality with respect to processing xAPI data.

2.1 Terminology

In the subsections and sections which follow, (s) indicates one or more

2.1.1 Scalar

Singular value x of a fundamental JSON type as described by JSON Schema

2.1.2 Collection

a n-tuple of items x such that

$$X = < x_i..x_n..x_j >$$

where

$$i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j$$

2.1.3 Key

A lookup k for a v within a kv where $k = x \vee X$

2.1.4 Value

a piece of data v where $v = x \vee X$

2.1.5 Key Value pair

Association between a k and v where

$$k \mapsto v$$

such that

$$kv = k \mapsto v$$

and a collection of Key Value pair(s) is defined as

$$KV = < kv_i..kv_n..kv_j >$$

such that

$$k_n \mapsto v_n$$

and all k within KV are unique

$$i_k \neq n_k \neq j_k$$

but the same is not true for all v within KV

$$i_v = n_v \vee i_v \neq n_v \;\; i_v = j_v \vee i_v \neq j_v \;\; j_v = n_v \vee j_v \neq n_v$$

2.1.6 Statement

Immutable collection of Key Value Pair(s) conforming to the xAPI Specification as described in the previous section

2.1.7 Algorithm State

Mutable collection of Key Value Pair(s)

2.1.8 Option

Collection of Key Value Pair(s) which alter the result of an Algorithm

3 Operation

Given an input X, an Operation produces output Y

If X changes to X' then the Operation results in Y' instead of Y

3.1 Domain

Any of the following

- Key(s)
- Value(s)
- Key Value pair(s)
- Statement(s)
- Algorithm State

3.2 Range

Any of the following dependent upon the Domain and Functionality of the Operation

- Key(s)
- Value(s)
- Key Value pair(s)
- Statement(s)
- Algorithm State

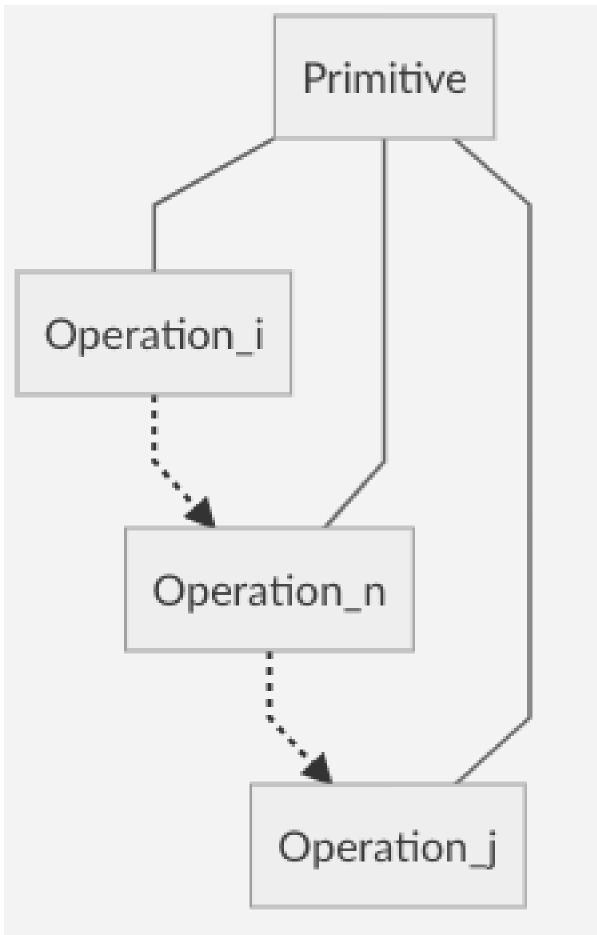
3.3 Formal Definition

A relationship between input and output data which will result in the same Y given the same X

$$Operation(X) = Y \wedge \ Operation(X') = Y'$$
$$\Rightarrow$$
$$Y = Y' \iff X = X' \wedge Y \neq Y' \iff X \neq X'$$

4 Primitive

A collection of Operations where the output of an Operation is passed as the argument to the next Operation



Primitives break the processing of xAPI data down into discrete units that can be composed to create new analytical functions. Primitives allow users to address the methodology of answering research questions as a sequence of generic algorithmic steps which establish the necessary data transformations, aggregations and calculations required to reach the solution in an implementation agnostic way.

4.1 Domain

Any of the following

- Key(s)
- Value(s)
- Key Value pair(s)
- Statement(s)
- Algorithm State

4.2 Range

Any of the following dependent upon the Domain and Functionality of the Primitive

- Key(s)
- Value(s)
- Key Value pair(s)
- Statement(s)
- Algorithm State

4.3 Formal Definition

A collection of Operation(s) O_n labeled p and defined as

$$p =< O_i..O_n..O_j >$$

where

$$i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j$$

such that the output Z is defined as the sequential composition of operation(s) O_n given arg(s) $Args$ provided to p

$$Z = p(Args) = O_j(O_n(O_i(Args)))$$

5 Algorithm

Given a collection of statement(s) $S_{<a..b..c>}$ and potentially option(s) *opt* and potentially an existing Algorithm State *state* an Algorithm A executes as follows

1. call *init*
2. for each $stmt \in S_{<a..b..c>}$
   (a) *relevant*?
   (b) *accept*?
   (c) *step*
3. return *result*

with each process within A is enumerated as

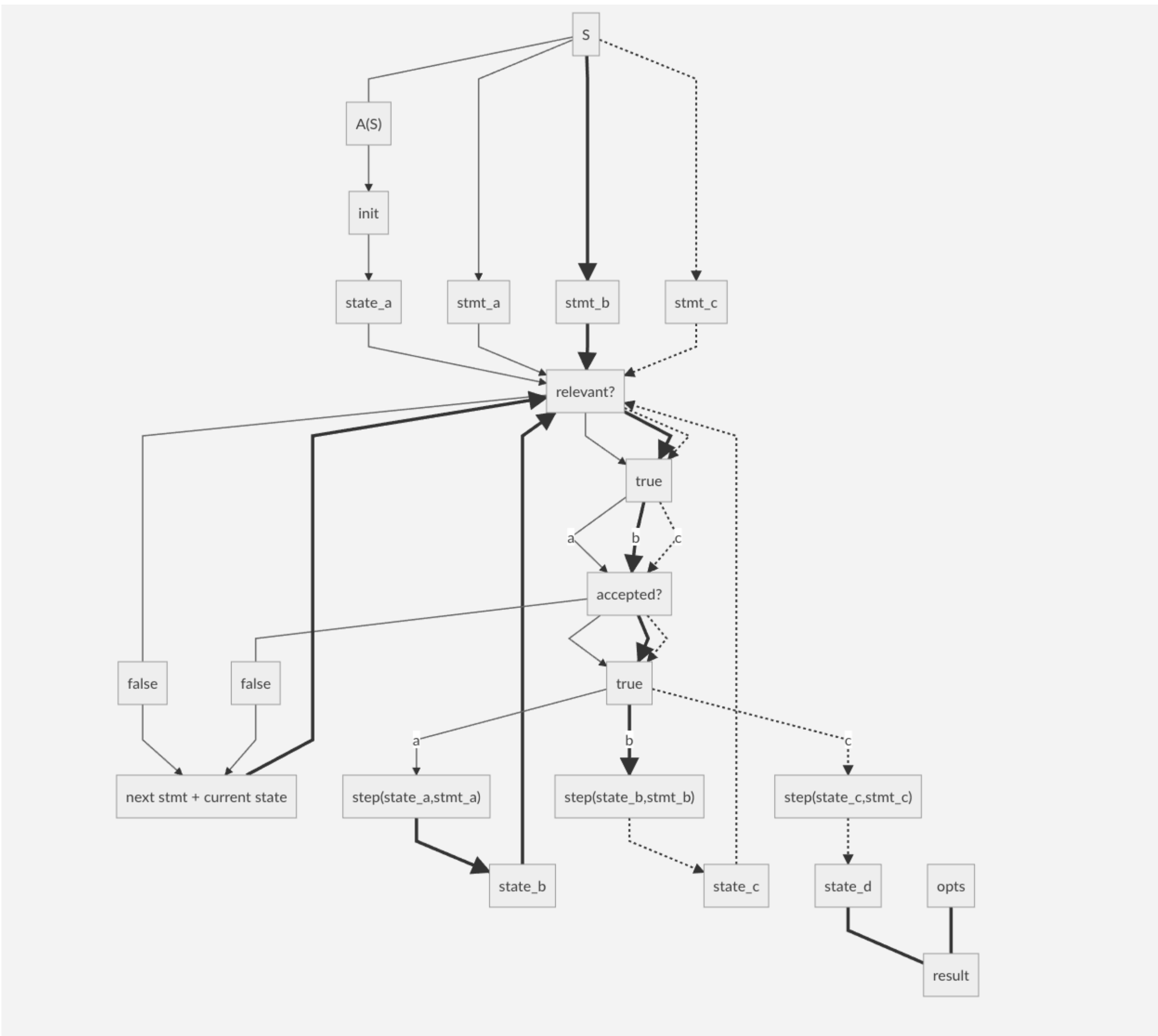
```
(init [state] body)
 - init state

(relevant? [state statement] body)
 - is the statement valid for use in algorithm?

(accept? [state statement] body)
 - can the algorithm consider the current statement?

(step [state statement] body)
 - processing per statement
 - can result in a modified state

(result [state] body)
 - return without option(s) provided
 - possibly sets default option(s)

(result [state opts] body)
 - return with consideration to option(s)
```

where

- *body* is a collection of Primitive(s) P which establishes the processing of inputs $\rightarrow$ outputs
- *state* is a mutable collection of key value pair(s) KV and synonymous with Algorithm State
- *statement* is a single statement s within the collection of statements S passed as input data to the Algorithm A

- *opts* are additional arguments passed to the algorithm A which impact the return value of the algorithm

Such that the execution of A can be described visually but not exhaustively as



5.1 Domain

Any of the following

- Statement(s)
- Algorithm State
- Option(s)

5.2 Range

- Algorithm State

5.3 Initialization

First process to run within an Algorithm which returns the starting Algorithm State $state_0$

$$init() = init(state) \vee init() \neq init(state)$$

where $state_0$ does not need to be related to its arguments

$$init() \rightarrow state_0$$

but $state_0$ can be derived from some other *state* passed as an argument to *init*

$$init(state) \rightarrow state'_0$$

such that

$$state_0 \neq state'_0$$

but this functionality is dependent upon the *body* of an Algorithms' *init*

5.3.1 Domain

- Algorithm State

5.3.2 Range

- Algorithm State

5.4 Relevant?

First process that each *stmt* passes through such that

$$relevant? \prec accept? \prec step$$

resulting in an indication of whether the *stmt* is valid for use within algorithm A

$$relevant?\,(state, stmt) = true\ \vee false$$

The criteria which determines validity of *stmt* within A is defined by the *body* of *relevant?*

5.4.1 Domain

- Statement
- Algorithm State

5.4.2 Range

- Scalar

5.5 Accept?

Second process that each *stmt* passes through such that

$$relevant? \prec accept? \prec step$$

resulting in an indication of whether the *stmt* can be sent to *step* given the current *state*

$$accept?\,(state, stmt) = true\ \vee false$$

The criteria which determines usability of *stmt* given *state*‘ is defined by the *body* of *accept?*

5.5.1 Domain

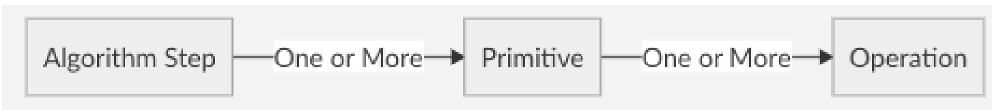
- Statement
- Algorithm State

5.5.2 Range

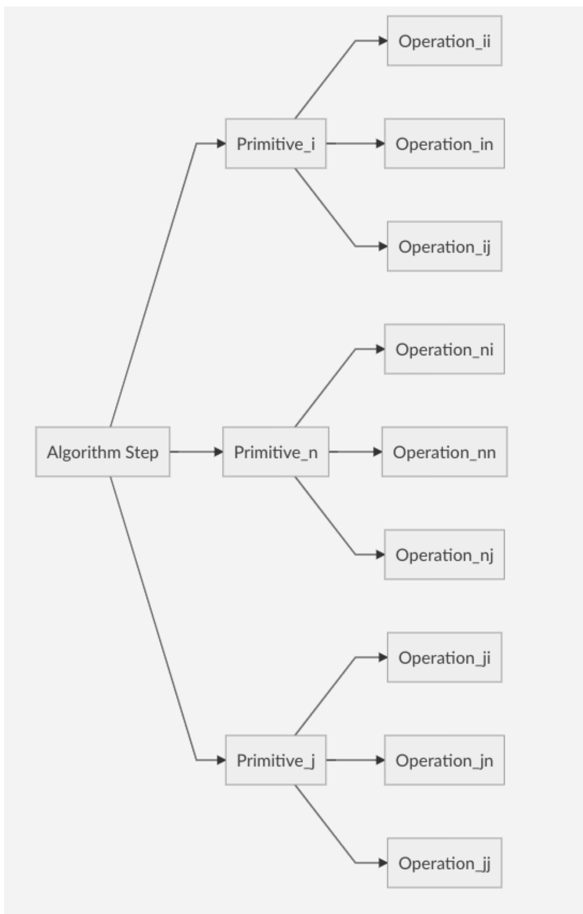
- Scalar

5.6 Step

An Algorithm Step consists of a collection of Primitive(s) and therefore collection(s) of Operation(s)



which expands to



$$i \leq n \leq j \Rightarrow i \prec n \prec j$$

$$i_i \leq i_n \leq i_j \leq n_i \leq n_n \leq n_j \leq j_i \leq j_n \leq j_j \Rightarrow i_{<i..n..j>} \prec n_{<i..n..j>} \prec j_{<i..n..j>}$$

where the output of a Primitive is passed as the argument to the next Primitive

The selection and ordering of Operation(s) and Primitive(s) into an Algorithmic Step determines how the Algorithm State changes during iteration through Statement(s) passed as input to the Algorithm.

5.6.1 Domain

- Statement
- Algorithm State

5.6.2 Range

- Algorithm State

5.6.3 Formal Definition

A collection of Primitive(s)

$$P =< p_i..p_n..p_j >$$

where

$$i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j$$

and

$$Z_i = p_i(Args) \Rightarrow O_{ij}(O_{in}(O_{ii}(Args)))$$

where

$$ii \leq in \leq ij \Rightarrow ii \prec in \prec ij \iff ii \neq in \neq ij$$

such that for each $stmt_b$ within a collection of Statement(s) S defined as

$$S =< stmt_a..stmt_b..stmt_c >$$

where

$$a \leq b \leq c \Rightarrow a \prec b \prec c \iff a \neq b \neq c$$

and

$$a \not\mapsto i \ \wedge b \not\mapsto n \ \wedge c \not\mapsto j$$

The output of *step* given a $stmt_b$ and $state_b$ is defined as

$$step(state_b, stmt_b) = p_j(p_n(Z_{ib}))$$

where

$$Z_{ib} = p_i(Args) \Rightarrow p_i(state_b, stmt_b) \Rightarrow O_{ij}(O_{in}(O_{ii}(state_b, stmt_b)))$$

and subsequently

$$Z_{nb} = p_n(Z_{ib})$$

which establishes that

$$Z_{jb} = p_j(Z_{nb}) \Rightarrow p_j(p_n(p_i(state_b, stmt_b)))$$

such that for a given $stmt_b$, $P_{<i..n..j>}$ will always result in a Z_{jb} but

$$Z_{ib} = state_b \vee state'_{ib} \iff state_b \neq state'_{ib}$$

which means

$$Z_{nb} = p_n(state_b, stmt_b) \vee p_n(state'_{ib}, stmt_b)$$

$$\Rightarrow$$

$$Z_{nb} = state_b \ \vee state'_{ib} \ \vee state'_{nb}$$

$$\Rightarrow$$

$$Z_{nb} = Z_{ib} \vee state'_{nb} \iff state_b \neq state'_{ib} \neq state'_{nb}$$

and concludes with

$$Z_{jb} = p_j(state_b, stmt_b) \vee p_j(state'_{ib}, stmt_b) \vee p_j(state'_{nb}, stmt_b)$$

$$\Rightarrow$$

$$Z_{jb} = state_b \vee state'_{ib} \vee state'_{nb} \vee state'_{jb}$$

$$\Rightarrow$$

$$Z_{jb} = Z_{nb} \vee state'_{jb} \iff state_b \neq state'_{ib} \neq state'_{nb} \neq state'_{jb}$$

such that

$$Z_{jb} \equiv state'_b$$
$$\Rightarrow$$
$$state'_b = state_b \vee state'_{ib} \vee state'_{nb} \vee state'_{jb} \iff state_b \neq state'_{ib} \neq state'_{nb} \neq state'_{jb}$$

the impact being that iteration through all $stmt \in S< a..b..c >$ results in a return of Z_{jc} such that

$$Z_{ja} = step(state_a, stmt_a) \Rightarrow state'_a \equiv Z_{ja} = state_a \vee state'_{ia} \vee state'_{na} \vee state'_{ja}$$

and

$$Z_{jb} = step(Z_{ja}, stmt_b) \Rightarrow state'_b \equiv Z_{jb} = Z_{ja} \vee state'_{ib} \vee state'_{nb} \vee state'_{jb}$$

meaning

$$Z_{jc} = step(Z_{jb}, stmt_c) \Rightarrow state'_c \equiv Z_{jc} = Z_{jb} \vee state'_{ic} \vee state'_{nc} \vee state'_{jc}$$

such that each $stmt \in S< a..b..c >$ may not result in a mutation of $state$ from $state \rightarrow state'$

$$state'_c = Z_{jc}$$
$$\Rightarrow$$
$$state'_c = state_a \vee state_{ia} \vee state'_{na} \vee state'_{ja} \vee state'_{ib} \vee state'_{nb} \vee state'_{jb} \vee state'_{ic} \vee state'_{nc} \vee state'_{jc}$$
$$\Rightarrow$$
$$state'c = state_a \vee state'c \neq statea$$

The no-op scenario described above is only a possibility of $step(state_a, stmt \in S_{<a..b..c>})$ but can be predicted to occur given

- The definition of individual Operations O which constitute a Primitive p

$$Operation(X) = Y \wedge Operation(X') = Y' \Rightarrow Y = Y' \iff X = X'$$

- The ordering of $O_{<i..n..j>}$ within p

$$i \prec n \prec j$$

- The Primitive(s) p chosen for inclusion within $P_{<i..n..j>}$

$$Z_i = p_i(Args) \Rightarrow O_{ij}(O_{in}(O_{ii}(Args)))$$

$$Z_j = p_j(Args) \Rightarrow O_{jj}(O_{jn}(O_{ji}(Args)))$$

$$\forall Args\ \exists\ Z_i = Z_j \iff O_{ij}(O_{in}(O_{ii}(Args))) \equiv O_{jj}(O_{jn}(O_{ji}(Args)))$$

$$< p_i, p_j > \equiv < p_j, p_i > \iff Z_i = Z_j$$

- The ordering of $p \in P_{i..n..j}$ which implies the ordering of $O \in p_{<i..n..j>} \in P_{<ii..ij..ni..nj..ji..jj>}$

$$i \prec n \prec j \Rightarrow ii \prec in \prec ij \Rightarrow ii \prec ij \prec ni \prec nj \prec ji \prec jj$$

$$P_{i..n..j} = P_{x..y..z} \Rightarrow < p_i, p_n, p_j > \equiv < p_x, p_y, p_z > \iff p_i \equiv p_x \wedge p_n \equiv p_y \wedge p_j \equiv p_z$$

$$\Rightarrow$$

$$P_{i..n..j} = P_{x..y..z} \iff i \mapsto x \wedge n \mapsto y \wedge j \mapsto z \wedge Z_i = Z_x \wedge Z_n = Z_y \wedge Z_j = Z_z$$

- The Key Value pair(s) $kv \in stmt \in S_{<a..b..c>}$
- The ordering of Statement(s) $stmt \in S_{<a..b..c>}$ such that $a \prec b \prec c$

5.7 Result

Last process to run within an Algorithm which returns the Algorithm State $state$ without preventing subsequent call of A

$$relevant? \prec accept? \prec step \prec result \prec relevant? \iff S \neq \emptyset$$

$$\Rightarrow$$

$$relevant? \prec accept? \prec step \prec result \iff S = \emptyset$$

such that if $S(t_n) = \emptyset$ and at some future point j within the timeline $i..n..j$ this is no longer true $S(t_j) \neq \emptyset$ then

$$A(state_{n-1},\ S(t_{n-1})) = state_n = A(init(),\ S(t_{n-i})) \iff A(state_n,\ S(t_n)) = state_n$$

such that the statement(s) added to S between t_i and t_n is

$$S(t_{n-i})$$

and the statement(s) added to S between t_n and t_j be

$$S(t_{j-n})$$

such that

$$S(t_{n-i})\ \cup\ S(t_{j-n}) = S(t_{j-i})$$

which means

$$A(init(),\ S(t_{j-i})) = state_j$$

and establishes that A can pick up from a previous $state_n$ without losing track of its own history.

$$A(result(state_n),\ S(t_{j-n})) = A(init(),\ S(t_{j-i})) = state_j$$

$$\iff$$

$$result(state_n) = A(init(),\ S(t_{n-i})) = state_n$$

Which makes A capable of taking in some $S_{<i..n..j..\infty>}$ as not all $s \in S_{<i..\infty>}$ have to be considered at once. In other words, the input data does not need to persist across the history of A, only the effect of s on $state$ must be persisted.

Additionally, the effect of $opts$ is determined by the $body$ within $result$ such that

$$A(result(state_n),\ S(t_{j-n}),\ opts)$$

$$\equiv$$

$$A(init(),\ S(t_{j-i}))$$

$$\equiv$$

$$A(init(),\ S(t_{j-i}),\ opts)$$

$$\equiv$$

$$A(result(state_n),\ S(t_{j-n}))$$

Which implies that $opts$ may have an effect on $state$ but not in a way which prevents backwards compatibility of $state$

5.7.1 Domain

- Algorithm State
- Option(s)

5.7.2 Range

- Algorithm State

6 Foundational Operations

6.1 Collections

Operations which expect a Collection of Scalar(s)

$$X =< x_i..x_n..x_j >$$

6.1.1 Array?

The operation *array*? will return a boolean which indicates if the passed in argument is a Collection

$$array?\,(Arg) = true \vee false$$

such that if X is a collection $< x_0, x_1, x_2, x_3, x_4 >$ where

$$x_0 = 0$$

$$x_1 = foo$$

$$x_2 =< baz,\ qux >$$

$$x_3 =< abc \mapsto 123,\ def \mapsto 456 >$$

$$x_4 =<< ghi \mapsto 789,\ jkl \mapsto 101112 >,\ < ghi \mapsto 131415,\ jkl \mapsto 161718 >>$$

then the following return true

$$array?\,(X) = true$$

$$array?\,(x_2) = true$$

$$array?\,(x_4) = true$$

and the following return false

$$array?\,(x_0) = false$$

$$array?\,(x_1) = false$$

$$array?\,(x_3) = false$$

Indicating that *array*? will return true when

- Arg is a Collection or Scalar(s)

but will return false when

- Arg is a non-array Scalar

which emphasises the difference between

- an *object*

$$x_3 =< abc \mapsto 123,\ def \mapsto 456 >= KV$$

- a Collection of *object*(s)

$$x_4 =<< ghi \mapsto 789,\ jkl \mapsto 101112 >,\ < ghi \mapsto 131415,\ jkl \mapsto 161718 >>=< KV_0, KV_1 >$$

6.1.2 Append

The operation *append* will return a Collection with a Value added at a specified numeric Index.

$$append(coll, v, idx) \to X'$$

such that if X is a collection $< x_0, x_1, x_2 >$ where

$$x_0 = 0$$

$$x_1 = foo$$

$$x_2 =< a, b, c >$$

then the v will be added to X to create X'

$$append(X, bar, 0) =< bar, 0, foo, < a, b, c >>= X'$$

$$append(X, bar, 1) =< 0, bar, foo, < a, b, c >>= X'$$

$$append(X, bar, 2) =< 0, foo, bar, < a, b, c >>= X'$$

$$append(X, bar, 3) =< 0, foo, < a, b, c >, bar >= X'$$

6.1.3 Remove

The operation *remove* will return a Collection minus the Value removed from the specified Numeric Index

$$remove(coll, idx) \to X'$$

such that if X is a collection $< x_0, x_1, x_2 >$ where

$$x_0 = 0$$

$$x_1 = foo$$

$$x_2 = baz$$

then

$$remove(X, 0) =< foo, baz >= X'$$

$$remove(X, 1) =< 0, baz >= X'$$

$$remove(X, 2) =< 0, foo >= X'$$

and if idx does not exist within X, *remove* will return X unaltered

$$remove(X, 3) =< 0, foo, baz >= X$$

6.1.4 At Index

The operation $atIndex$ will return the Value at

- a numeric Index
$$atIndex(X, idx) \rightarrow v$$
- some depth of numeric indexes
$$atIndex(X, < idx_i..idx_n..idx_j >) \rightarrow v$$

such that if X is a collection $< x_0, x_1, x_2 >$ where

$$x_0 = 0$$

$$x_1 = foo$$

$$x_2 =< a, b, c >$$

then

$$atIndex(X, 0) = 0$$

$$atIndex(X, 1) = foo$$

$$atIndex(X, < 1, 0 >) = f$$

$$atIndex(X, < 1, 2 >) = o$$

$$atIndex(X, 2) =< a, b, c >$$

$$atIndex(X, < 2, 1 >) = b$$

and if idx does not exist within X, $atIndex$ will return the representation of nothingness

$$atIndex(X, 3) = nil$$

$$atIndex(X, < 2, 3 >) = nil$$

6.2 Key Value Pairs

Operations which expect a Collection of Key Value pair(s)

$$KV =< k_i v_{k_i}..k_n v_{k_n}..k_j v_{k_j} >$$

6.2.1 Object?

The operation *object*? will return a boolean which indicates if the passed in argument is a mapping of Key(s) to Value(s)

$$object?\,(Arg) = true \vee false$$

such that if KV is a collection of Key Value pair(s) $< k_0 v_{k_0}, k_1 v_{k_1}, k_2 v_{k_2}, k_3 v_{k_3} >$ where

$$k_0 = abc \ \wedge v_{k_0} = 123$$
$$\Rightarrow$$
$$k_0 v_{k_0} = abc \mapsto 123$$

and

$$k_1 = def \ \wedge v_{k_1} = xyz \mapsto 456$$
$$\Rightarrow$$
$$k_1 v_{k_1} = def \mapsto xyz \mapsto 456$$

and

$$k_2 = ghi \ \wedge v_{k_2} =< 7, 8, 9 >$$
$$\Rightarrow$$
$$k_2 v_{k_2} = ghi \mapsto < 7, 8, 9 >$$

and

$$k_3 = k_0 v_{k_0} \ \wedge v_{k_3} = v_{k_2}$$
$$\Rightarrow$$
$$k_3 v_{k_3} =<< abc \mapsto 123 > \mapsto < 7, 8, 9 >>$$

then the following will return true

$$object?\,(KV)$$
$$object?\,(k_0 v_{k_0}) \Rightarrow object?\,(abc \mapsto 123)$$
$$object?\,(k_1 v_{k_1}) \Rightarrow object?\,(def \mapsto xyz \mapsto 456)$$
$$object?\,(k_2 v_{k_2}) \Rightarrow object?\,(ghi \mapsto < 7, 8, 9 >)$$
$$object?\,(k_3 v_{k_3}) \Rightarrow object?\,(<< abc \mapsto 123 > \mapsto < 7, 8, 9 >>)$$
$$object?\,(v_{k_1}) \Rightarrow object?\,(xyz \mapsto 456)$$
$$object?\,(k_3) \Rightarrow object?\,(abc \mapsto 123)$$

and the following return false

$$object?\,(k_0) \Rightarrow object?\,(abc)$$
$$object?\,(k_1) \Rightarrow object?\,(def)$$
$$object?\,(v_{k_0}) \Rightarrow object?\,(123)$$
$$object?\,(v_{k_2}) \Rightarrow object?\,(< 7, 8, 9 >)$$

6.2.2 Associate

The operation *associate* will return a modified KV' with a mapping of

- Key to Value added at a specified Key.

$$associate(KV, k, v) \rightarrow KV'$$

- Key to Value added at a specified nesting of Key(s).

$$associate(KV, < k_i..k_n..k_j >, v) \rightarrow KV'$$

such that if KV is a collection of Key Value pair(s) $< k_0 v_{k_0}, k_1 v_{k_1} >$ where

$$k_0 = abc \ \wedge v_{k_0} = 123$$

$$\Rightarrow$$

$$k_0 v_{k_0} = abc \mapsto 123$$

and

$$k_1 = def \ \wedge v_{k_1} = xyz \mapsto 456$$

$$\Rightarrow$$

$$k_1 v_{k_1} = def \mapsto xyz \mapsto 456$$

such that

$$KV =< abc \mapsto 123, \ def \mapsto xyz \mapsto 456 >$$

When k is a single Key

- $k \notin KV$, $k \mapsto v$ is added to KV to create KV'

$$associate(KV, baz, foo) = KV'$$

$$=$$

$$< abc \mapsto 123, \ def \mapsto xyz \mapsto 456, \ baz \mapsto foo >$$

- $k \in KV$, the previous mapping is overwritten to create KV'

$$associate(KV, abc, 789) = KV'$$

$$=$$

$$< abc \mapsto 789, \ def \mapsto xyz \mapsto 456 >$$

$$associate(KV, def, 456) = KV'$$

$$=$$

$$< abc \mapsto 123, \ def \mapsto 456 >$$

When k is a Collection of Key(s) $K =< k_i, k_j >$

- $K \notin KV$, $k_i \mapsto k_j \mapsto v$ is added to KV to create KV'

$$associate(KV, < baz, bar >, foo) = KV'$$
$$=$$
$$< abc \mapsto 123,\ def \mapsto xyz \mapsto 456,\ baz \mapsto bar \mapsto foo >$$

- $k_i \in KV \ \wedge k_j \notin v_{k_i} \ \wedge \ object?\,(v_{k_i}) = false$, the previous mapping at k_i is overwritten to create KV'

$$associate(KV, < abc, cba >, 789) = KV'$$
$$=$$
$$< abc \mapsto cba \mapsto 789,\ def \mapsto xyz \mapsto 456 >$$

- $k_i \in KV \ \wedge k_j \notin v_{k_i} \ \wedge \ object?\,(v_{k_i}) = true$, $k_j \mapsto v$ is added to k_i

$$associate(KV, < def, zyx >, fizbuz) = KV'$$
$$=$$
$$< abc \mapsto 123,\ def \mapsto < xyz \mapsto 456, zyx \mapsto fizbuz >>$$

- $k_i \in KV \ \wedge k_j \in v_{k_i} \Rightarrow object?\,(v_{k_i}) = true$, v_{k_j} is replaced with v

$$associate(KV, < def, xyz >, 654) = KV'$$
$$=$$
$$< abc \mapsto 123,\ def \mapsto xyz \mapsto 654 >$$

6.2.3 Dissociate

The operation *dissociate* will remove some $k \mapsto v$ from KV given $k \in KV$

$$dissociate(KV, k) \rightarrow KV'$$

such that if KV is a collection of Key Value pair(s) $< k_0 v_{k_0}, k_1 v_{k_1} >$ where

$$k_0 = abc \ \wedge v_{k_0} = 123$$
$$\Rightarrow$$
$$k_0 v_{k_0} = abc \mapsto 123$$

and

$$k_1 = def \ \wedge v_{k_1} = xyz \mapsto 456$$
$$\Rightarrow$$
$$k_1 v_{k_1} = def \mapsto xyz \mapsto 456$$

such that

$$KV =< abc \mapsto 123,\ def \mapsto xyz \mapsto 456 >$$

Which means $disocciate(KV, k)$ results in

- $KV \neq KV' \iff k \in KV$ where $k \mapsto v_k$ is removed from KV

$$dissociate(KV, abc) =< def \mapsto xyz \mapsto 456 >= KV'$$

$$dissociate(KV, def) =< abc \mapsto 123 >= KV'$$

- $KV = KV' \iff k \notin KV$ where nothing is removed from KV

$$dissociate(KV, cba) =< abc \mapsto 123,\ def \mapsto xyz \mapsto 456 >= KV' = KV$$

6.2.4 At Key

The operation $atKey$ will return the Value v at some specified

- Top level Key k within KV

$$atKey(KV, k) \to v$$

- Nested location $K =< k_i..k_n..k_j >$ within KV

$$atKey(KV, K) \to v$$

such that if KV is a collection of Key Value pair(s) $< k_0 v_{k_0}, k_1 v_{k_1} >$ where

$$k_0 = abc\ \wedge v_{k_0} = 123$$

$$\Rightarrow$$

$$k_0 v_{k_0} = abc \mapsto 123$$

and

$$k_1 = def\ \wedge v_{k_1} = xyz \mapsto 456$$

$$\Rightarrow$$

$$k_1 v_{k_1} = def \mapsto xyz \mapsto 456$$

such that

$$KV =< abc \mapsto 123,\ def \mapsto xyz \mapsto 456 >$$

When k is a single Key

- $k \notin KV$, $atKey$ will return the representation of nothingness

$$atKey(KV, cba) = nil$$

- $k \in KV$, $atKey$ will return v_k

$$atKey(KV, k_0) \Rightarrow atKey(KV, abc) = 123$$

$$atKey(KV, k_1) \Rightarrow atKey(KV, def) = xyz \mapsto 456$$

When K is a Collection of Key(s)

- $K \notin KV$, $atKey$ will return the representation of nothingness

$$atKey(KV, < cba, 321 >) = nil$$

- $k_i \in KV \ \wedge k_j \notin KV$, $atKey$ will return the representation of nothingness

$$atKey(KV, < def, abc >) = nil$$

- $k_i \in KV \ \wedge k_j \in KV$, $atKey$ will return the nested value

$$atKey(KV, < def, xyz >) = 456$$

6.3 Utility

6.3.1 Map

The *map* operation accepts the following arguments

- Operation or Primitive
- A Collection
- Additional Arguments passed to the Operator or Primitive

and returns a collection of Operation $Operation(x_n, args) \vee Primitive(x_n, args)$ respectively

Given an input Collection X and a Operation o or Primitive p where

$$X =< x_i..x_n..x_j >$$

and that collection consists of one or more members x_n within the range $i..j$

$$i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j$$

then

$$map(o,\ X,\ args) = Y \ \wedge \ map(p,\ X,\ args) = Y'$$

such that

$$Y =< o(x_i, args)..o(x_n, args)..o(x_j, args) >$$

and

$$Y' =< p(x_i, args)..p(x_n, args)..p(x_j, args) >$$

which establishes both Y and Y' are a Collection where each member y_n or y'_n is the result of passing x_n and $args$ to o or p respectively.

In otherwords

$$o(x_i, args) \mapsto y_i \wedge o(x_n, args) \mapsto y_n \wedge o(x_j, args) \mapsto y_j$$

$$p(x_i, args) \mapsto y_i' \wedge p(x_n, args) \mapsto y_n' \wedge p(x_j, args) \mapsto y_j'$$

which implies both collections Y and Y' have the same ordering as collection X

$$i_{o(x,args)} = i_y \ \wedge \ n_{o(x,args)} = n_y \ \wedge \ j_{o(x,args)} = j_y$$

$$i_{p(x,args)} = i_{y'} \ \wedge \ n_{p(x,args)} = n_{y'} \ \wedge \ j_{p(x,args)} = j_{y'}$$

When X contains non-distinct values, o and p are unaffected.

$$o(x_n, args) = y_n$$

$$o(x_{n'}, args) = y_{n'}$$
$$o(x_{n'+1}, args) = y_{n'}$$
$$\Longleftrightarrow$$
$$x_{n'} \equiv x_{n'+1} \wedge x_{n'} \not\equiv x_n$$
$$\Rightarrow$$
$$x_n \not\equiv x_{n'+1}$$
$$\Rightarrow$$
$$o(x_{n'}, args) = o(x_{n'+1}, args) \neq o(x_n, args)$$

Because p is just a composition of o's, the same property holds for primitives

$$p(x_n, args) = y_n'$$

$$p(x_{n'}, args) = y_{n'}'$$
$$p(x_{n'+1}, args) = y_{n'}'$$
$$\Longleftrightarrow$$
$$x_{n'} \equiv x_{n'+1} \wedge x_{n'} \not\equiv x_n$$
$$\Rightarrow$$
$$x_n \not\equiv x_{n'+1}$$
$$\Rightarrow$$
$$p(x_{n'}, args) = p(x_{n'+1}, args) \neq p(x_n, args)$$

6.3.2 Count

the *count* operation accepts the following arguments

- A Scalar
  - An Object
  - An Array
  - A String
- A Collection of Scalar(s)
- A Collection of Key Value pair(s)

and returns the corresponding number of items

$$count(arg) = \mathbb{R}$$

Such that

$$count(arg) \equiv count(< arg_i..arg_n..arg_j >) = j + 1$$

and with consideration to accepted argument types

- when *arg* is a Scalar String

$$x = abc$$

$$count(x) \equiv count(< x_i..x_n..x_j >) \equiv count(< a, b, c >)$$

$$count(x) = 3$$

$$\Rightarrow$$

$$index_a = 0 \ \wedge \ index_b = 1 \ \wedge \ index_c = 2$$

$$\Rightarrow$$

$$count(x) \equiv index_c + 1 = 3$$

- when *arg* is a Scalar Object or Collection of Key Value pair(s), only the Key(s) are counted

$$KV =< abc \mapsto < ABC \mapsto 123,\ cba \mapsto 321 >, xyz \mapsto 789 >$$

$$count(KV) \equiv count(< KV_{k_i}..KV_{k_n}..KV_{k_j} >) \equiv count(< abc, xyz >)$$

$$count(KV) = 2$$

$$\Rightarrow$$

$$index_{abc} = 0 \ \wedge \ index_{xyz} = 1$$

$$\Rightarrow$$

$$count(KV) \equiv index_{xyz} + 1 = 2$$

- when arg is a Scalar Array or a Collection of Scalar(s), only the members are counted

$$X =< foo, baz, 10, < 1, 2, 3 >, true >$$
$$count(X) \equiv count(< X_i..X_n..X_j >) \equiv count(< foo, baz, 10, < 1, 2, 3 >, true >)$$
$$count(X) = 5$$
$$\Rightarrow$$
$$index_{foo} = 0 \wedge index_{baz} = 1 \wedge index_{10} = 2 \wedge index_{<1,2,3>} = 3 \wedge index_{true} = 4$$
$$\Rightarrow$$
$$count(X) \equiv index_{true} + 1 = 5$$

6.3.3 First

The operation $first$ will return the first element of a

- String Scalar
- Array Scalar
- Collection

such that

$$first(arg) = arg_i$$

and when arg is a Array Scalar or Collection

$$first(X) \Rightarrow first(< x_i..x_n..x_j >) = x_i$$

and when arg is a String Scalar

$$first(abc) \Rightarrow first(< a, b, c >) = a$$

6.3.4 Last

The operation $last$ will return the last element of a

- String Scalar
- Array Scalar
- Collection

such that

$$last(arg) = arg_j$$

and when arg is a Array Scalar or Collection

$$last(X) \Rightarrow last(< x_i..x_n..x_j >) = x_j$$

and when arg is a String Scalar

$$last(abc) \Rightarrow last(< a, b, c >) = c$$

6.3.5 Iso To Unix Epoch

The $isoToUnixEpoch$ operation accepts the following arguments

- An ISO 8601 Timestamp as specified in the xAPI Specification

and returns the following

- The number of seconds that have elapsed since January 1, 1970

such that

$$isoToUnixEpoch(\$.timestamp) \rightarrow \mathbb{R}$$

which results in a conversion from the ISO string format to a numeric count of seconds.

$$ts = 2015 - 11 - 18T12 : 17 : 00 + 00 : 00 \equiv 2015 - 11 - 18T12 : 17 : 00Z$$

$$isoToUnixEpoch(ts) = 1447849020$$

6.3.6 Timeunit To Number of Seconds

the $timeunit \rightarrow seconds$ operation accepts the following arguments

- A string Scalar of
  - second
  - minute
  - hour
  - day
  - week
  - month
  - year

and returns the corresponding number of seconds such that

$$timeunit \rightarrow seconds(timeUnit) = \mathbb{R}$$

Where the following is the enumeration of $timeunit \rightarrow seconds(timeUnit)$

$$timeunit \rightarrow seconds(second) = 1$$

$$timeunit \rightarrow seconds(minute) = 60$$

$$timeunit \rightarrow seconds(hour) = 3600$$

$$timeunit \rightarrow seconds(day) = 86400$$

$$timeunit \rightarrow seconds(week) = 604800$$

$$timeunit \rightarrow seconds(month) = 2629743$$

$$timeunit \rightarrow seconds(year) = 31556926$$

7 Common Primitives

7.1 Accumulate

Performs an update at *path* within *state* using the supplied *item* or $k \wedge v$

$$accumulate(state, path, item) \rightarrow state'$$

$$accumulate(state, path, k, v) \rightarrow state'$$

7.1.1 Arguments

- *state* is an Algorithm State
- *path* is a Collection of Key(s) used to navigate into *state*
- *item* is a Scalar which should be reflected within $state'$ at *path*
- k is a Value used as the target
  - Index within some Collection
  - Key within some KV such that $k \mapsto v \in KV$
- v is a Value
  - Added to some Collection at index k
  - Mapped to k within some KV such that $k \mapsto v \in KV$

7.1.2 Relevant Operations

The primitive *accumulate* uses the operations

- array?
- object?
- append
- associate
- atKey
- count

7.1.3 Summary

accumulate will do one of the following things

- replace an existing non-array Scalar or KV

$$accumulate(state, path, item) \equiv associate(state, path, item)$$

- update an existing array Scalar or Collection

$$accumulate(state, path, item) \equiv associate(state, path, append(state_{path}, item, count(state_{path})))$$

- updates an existing Scalar object or KV

$$accumulate(state, path, k, v) \equiv associate(state, append(path, k, count(path)), v)$$

- updates an existing array Scalar or Collection

$$accumulate(state, path, k, v) \equiv associate(state, path, append(state_{path}, v, k))$$

- create a new Collection containing $state_{path}$ and v

$$accumulate(state, path, k, v) \equiv associate(state, path, append(append(<>, state_{path}, 0), v, k))$$

7.1.4 Usage of Operations

In order to update the argument *state* at *path* using *item* or $k \ \wedge \ v$ the first step is always retrieving the value at *path* using the operation $atKey$. This operation is used because by definition, *state* is a KV

$$state_{path} = atKey(state, path)$$

to determine its type

$$state_{path} = Object \ \vee \ KV \ \vee \ x \ \vee X$$

such that the following bullet points represent the beahvior of *accumulate* under various conditions

- $object?\,(state_{path}) = true$
  - and *item* passed in as argument

$$updatedState = associate(state, path, item)$$

  - and $k \ \wedge v$ passed in as argument

$$index = count(path)$$

$$fullPath = append(path, k, index)$$

$$updatedState = associate(state, fullPath, v)$$

- $array?\,(state_{path}) = true$
  - and $item$ passed in as argument

$$index = count(state_{path})$$

$$updatedArray = append(state_{path}, item, index)$$
$$updatedState = associate(state, path, updatedArray)$$

  - and $k \ \wedge \ v$ passed in as argument

$$updatedArray = append(state_{path}, v, k)$$

$$updatedState = associate(state, path, updatedArray)$$

- $array?\,(state_{path}) = false \ \wedge \ object?\,(state_{path}) = false$
  - and $item$ passed in as argument

$$updatedState = associate(state, path, item)$$

  - and $k \ \wedge \ v$ passed in as argument

$$newArray = append(<>, state_{path}, 0)$$

$$updatedArray = append(newArray, v, k)$$
$$updatedState = associate(state, path, updatedArray)$$

Which shows that $accumulate$ has common steps across all conditions

$$state_{path} = atKey(state, path)$$

$$objectAtPath? = object?\,(state_{path})$$
$$arrayAtPath? = array?\,(state_{path})$$

but then the steps deviate based $item$ vs $k \ \wedge \ v$ such that the action of $accumulate$ when $item$ is passed in results in either

- an overwrite of $state_{path}$ via $associate(state, path, item)$
  - $objectAtPath? = true$
  - $objectAtPath? = false \ \wedge \ arrayAtpath? = false$
- an updated $state_{path}$ via $associate(state, path, append(state_{path}, item, count(state_{path})))$
  - $arrayAtPath? = true$

and the action of $accumulate$ when $k \ \wedge \ v$ is passed in results in either

- $objectAtPath? = true$

– an update of $state_{path}$ to include $k \mapsto v$

$$associate(state, append(path, k, count(path)), v)$$

- $arrayAtPath? = true$

  – an update of $state_{path}$ to include v at index k

$$associate(state, path, append(state_{path}, v, k))$$

- $objectAtPath? = false \ \wedge \ arrayAtpath? = false$

  – creation of a new array which contains $state_{path}$ and v at index k

$$associate(state, path, append(append(<>, state_{path}, 0), v, k))$$

7.1.5 Example output

To demonstrate the functionality of *accumulate*, the following assumptions will be made

$$state =< a \mapsto< b \mapsto< 1, 2, 3 >, c \mapsto 4 > d \mapsto foo, e \mapsto< 4, 5, 6 >>$$

$$\Rightarrow$$

$$state_a =< b \mapsto< 1, 2, 3 >, c \mapsto 4 >$$

$$state_d = foo$$

$$state_e =< 4, 5, 6 >$$

such that

$$accumulate(state, < d >, baz) =< state_a, \ d \mapsto baz, \ state_e >$$

and

$$accumulate(state, < a >, baz) =< a \mapsto baz, \ state_d, \ state_e >$$

and

$$accumulate(state, < a, c >, baz) =< a \mapsto< b \mapsto< 1, 2, 3 >, c \mapsto baz >, \ state_d, \ state_e >$$

and

$$accumulate(state, < e >, 7) =< state_a, \ state_d, \ e \mapsto< 4, 5, 6, 7 >>$$

and

$$accumulate(state, < e >, < 7, 8, 9 >) =< state_a, \ state_d, \ e \mapsto< 4, 5, 6, < 7, 8, 9 >>>$$

and

$$accumulate(state, < a >, b, < 3, 2, 1 >) =< a \mapsto< b \mapsto< 3, 2, 1 >, c \mapsto 4 >, \ state_d, \ state_e >$$

and

$accumulate(state, < a >, q, baz) =< a \mapsto < b \mapsto < 1, 2, 3 >, c \mapsto 4, q \mapsto baz >, \ state_d, \ state_e >$

and

$accumulate(state, < a, q >, r, baz) =< a \mapsto < b \mapsto < 1, 2, 3 >, c \mapsto 4, q \mapsto r \mapsto baz >, \ state_d, \ state_e >$

and

$$accumulate(state, < e >, 1, 7) =< state_a, \ state_d, \ e \mapsto < 4, 7, 5, 6 >>$$

and

$$accumulate(state, < d >, 0, baz) =< state_a, \ < baz, foo >, \ state_e >$$

and

$accumulate(state, < d >, 1, < baz, bar >) =< state_a, \ < foo, < baz, bar >>, \ state_e >$

7.2 At JSONPath

Performs a lookup at *path* within *source* similar to atKey

$$atJsonPath(source, path)$$

such that the fundamental functionality of JSONPath is covered in this definition.

- A more complete definition will come at a future date if/as necessary

7.2.1 Arguments

- *source* is an object Scalar, KV, Statement or an Algorithm State
- *path* is a JSONPath string which adheres to the additional requirements, clarifications, and additions placed on JSONPath by the xAPI Profile Specification

7.2.2 Relevant Operations

The primitive $atJsonPath$ uses the operations

- atKey
- atIndex
- append
- count

7.2.3 Summary

$atJsonPath$ will return a v found within $source$ after converting

$$path \rightarrow < path_{i+1}..path_j >$$

such that if

$$path = \$.a.b$$

then

$$path \rightarrow < a, b >$$

so that

$$atJsonPath(< a \mapsto b \mapsto 123 >, \$.a.b) = 123$$

7.2.4 Usage of Operations

In order to convert

$$path \rightarrow < path_{i+2}..path_j >$$

an empty Collection $keyState$ is introduced

$$keyState = <>$$

so that the relevant k'(s) can be stored in $keyState$ during iteration over $path$

$$\forall n : i..j \ \bullet i = 0 \ \wedge j = count(path) - 1$$

and the number of stored keys can be tracked using $curKeyStateIndex$

$$curKeyStateIndex = count(keyState) - 1$$

such that the current $path_n$ can be retrieved

$$curKey = atIndex(path, n)$$

and $keepKey?$ can indicate the relevance of $path_n$

$$keepKey? = true \iff curKey \neq \$ \ \wedge curKey \neq .$$

such that during each iteration n, $keyState$ will be updated if necessary

$$keyState = append(keyState,\ curKey,\ curKeyStateIndex) \iff keepKey? = true$$

so at the end of the loop

$$keyState = < path_{i+2}..path_n..path_j >$$

which provides the Collection of Key(s) necessary for calling $atKey$

$$valueInSource = atKey(source, keyState)$$

such that

$$atJsonPath(source, path) \equiv atKey(source, keyState)$$

7.2.5 Example output

Given an example *source*

$$source = < a \mapsto < b \mapsto 123, c \mapsto 456 >, d \mapsto foo >$$

then

$$atJsonPath(source, \$.a) = < b \mapsto 123, c \mapsto 456 >$$

and

$$atJsonPath(source, \$.a.b) = 123$$

and

$$atJsonPath(source, \$.a.c) = 456$$

and

$$atJsonPath(source, \$.d) = foo$$

7.3 Rate Of

Calculates the number of times something occured within an interval of time given a unit of time.

$$rateOf(nOccurances, start, end, unit)$$

Where the output translates to: the rate of occurance per unit within interval

7.3.1 Arguments

- *nOccurances* is the number of times something happened and should be an Integer
- *start* is an ISO 8601 timestamp which serves as the first timestamp within the interval
- *end* is an ISO 8601 timestamp which servers as the last timestamp within the interval
- *unit* is a String Enum representing the unit of time

7.3.2 Relevant Operations

- isoToUnixEpoch
- $timeunit \rightarrow seconds$

7.3.3 Summary

$rateOf$ determines the number of seconds within the interval $start \rightarrow end$

$$intervalSeconds = isoToUnixEpoch(end) - isoToUnixEpoch(start)$$

and resolves the numer of seconds corresponding to $unit$

$$unitSeconds = timeunit \rightarrow seconds(unit)$$

so that the interval can be converted from Seconds to $unit$

$$per = intervalSeconds/unitSeconds$$

and the rate can be calculated

$$rateOf(nOccurances, start, end, unit) \equiv nOccurances \div per$$

7.3.4 Usage of Operations

The Operations used within $rateOf$ convert from a String to a Integer

- $isoToUnixEpoch$ is used to convert

$$end \ \wedge start \rightarrow \mathbb{R}$$

- $timeunit \rightarrow seconds$ is used to convert

$$unit \rightarrow \mathbb{R}$$

The only other functionality required by $rateOf$ is supplied via basic arithmetic

7.3.5 Example output

Given an example $start$ and end

$$start = 2015 - 11 - 18T12 : 17 : 00Z$$

$$end = 2015 - 11 - 18T14 : 17 : 00Z$$

Then the Unix Epoch of each is

$$nStart = 1447849020$$

$$nEnd = 1447856220$$

Which provides an interval range (in seconds)

$$intervalSeconds = nEnd - nStart = 7200$$

which is divided by $timeunit \rightarrow seconds(unit)$ to derive per $unit$

$$per = 7200 \iff unit = second \Rightarrow 7200/1$$

$$per = 120 \iff unit = minute \Rightarrow 7200/60$$

$$per = 2 \iff unit = hour \Rightarrow 7200/3600$$

such that if

$$nOccurances = 10$$

and $unit = second$ then the output is 0.001389 occurances per second within $start \rightarrow end$

$$rateOf(nOccurances, start, end, second) \equiv 10/7200 \equiv 0.001389$$

and $unit = minute$ then the output is 0.0833 occurances per minute within $start \rightarrow end$

$$rateOf(nOccurances, start, end, minute) \equiv 10/120 \equiv 0.0833$$

and $unit = hour$ then the output is 5 occurances per hour within $start \rightarrow end$

$$rateOf(nOccurances, start, end, hour) \equiv 10/2 \equiv 5$$

Updated Algorithm Definitions

The following are examples of the new way in which Algorithms were defined. These sections are either in draft form or are a work in progress.

8 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), it becomes possible to retell their learning story through data and data visualization. One important aspect of that story is the rate of completion of the various digital resources within the learning ecosystem.

8.1 Initialization

$init(state)$ sets up an empty KV within $state$ for the Algorithm to update at each $step$

$$init(state) = state_0$$

where

$$state_0 = associate(state, < state, completions >, <>) \iff atKey(state, < state, completions >) = nil$$

otherwise

$$state_0 = state$$

such that if

$$state =< a \mapsto b >$$

then

$$state_0 =< a \mapsto b, state \mapsto completions \mapsto <>>$$

8.2 Relevant?

$relevant?(state, stmt)$ determines if $stmt$ is valid for use within $step$ of $rateOfCompletions$ and does so by looking into various $k \rightarrow v$ within $stmt$. The following Primitives are used as the $body$ of $relevant?(state, stmt)$

- is the Object of the Statement an Activity?

$$activityType = atKey(stmt, < object, objectType >)$$

$$activity?(activityType) = true \iff activityType = Activity \vee activityType = nil$$

- is the Verb indicative of a completion event?

$$verbId = atKey(stmt, < verb, id >)$$

$$completionVerb?(verbId) = true$$

$$\iff$$

$$verbId = http://adlnet.gov/expapi/verbs/passed$$

$$\vee$$

$$verbId = https://w3id.org/xapi/dod-isd/verbs/answered$$

$$\vee$$

$$verbId = http://adlnet.gov/expapi/verbs/completed$$

- does the *stmt* indicate completion using Result?

$$result = atKey(stmt, < result, completion >)$$

$$resultCompletion = true \iff result = true$$

such that the body of *relevant*? contains

$$p_a(stmt) = activity?\,(atKey(stmt, < object, objectType >))$$

and

$$p_v(stmt) = completionVerb?\,(atKey(stmt, < verb, id >))$$

and

$$p_r(stmt) = resultCompletion(atKey(stmt, < result, completion >))$$

which are used to form higher level Primitives

$$p_{continue}(stmt) = stmt \iff p_a(stmt) = true$$

and

$$p_{completed?}(stmt) = stmt \iff p_v(stmt) = true \;\vee\; p_r(stmt) = true$$

which results in a final Primitive $p_{return?}$

$$p_{return?}(stmt) = object?\,(p_{completed?}(p_{continue}(stmt)))$$

which defines the *body* of *relevant*?

$$relevant?\,(stmt) = p_{return?}(stmt) \Rightarrow object?\,(p_{completed?}(p_{continue}(stmt)))$$

and can be summarized as

$$relevant?\,(state, stmt) = true$$

$$\iff$$

$$activity?\,(activitType) = true$$

$$\wedge$$

$$completionVerb?\,(verbId) = true \;\vee\; resultCompletion = true$$

8.3 Accept?

$rateOfCompletions$ does not require further boolean logic to determine if $stmt$ and $state$ can be passed to $step$

$$accept?\,(state, stmt) = object?\,(stmt)$$

which should always return true assuming valid xAPI Statements are passed to $rateOfCompletions$

8.4 Step

8.4.1 summary

$step(state, stmt)$ updates $state$ to include

$$\$.object.id \mapsto < domain, statementCount, name >$$

where

$$domain \mapsto < start, end >$$

$$statementCount \mapsto \mathbb{R}$$

$$name \mapsto < \$.object.definition.name >$$

at

$$< state, completions, \$.object.id >$$

8.4.2 processing

$step$ starts by extracting the relevant information from $stmt$

- $currentTime$

$$currentTime = atKey(stmt, timestamp)$$

- $name$

$$name_{stmt} = atKey(stmt, < object, definition, name >)$$

- $objectId$

$$objectId = atKey(stmt, < object, id >)$$

which allows for the previous $state$ to be resolved using $objectId$

- $domain$

$$domain_{state} = atKey(state, < state, completions, objectId, domain >)$$

$$start_{state} = first(domain_{state})$$

$$end_{state} = last(domain_{state})$$

- *statementCount*

  $$statementCount_{state} = atKey(state, < state, completions, objectId, statementCount >)$$

- *name*

  $$name_{state} = atKey(state, < state, completions, objectId, name >)$$

so that the previous state can be used along side the information parsed from *stmt*

- does $start_{state}$ need to be updated to $currentTime$?

  where
  $$inSeconds_{stmt} = isoToUnixEpoch(currentTime)$$
  $$inSeconds_{start} = isoToUnixEpoch(start_{state}) \iff start_{state} \neq nil$$
  such that
  $$start(state, stmt) = currentTime$$
  $$\iff$$
  $$start_{state} = nil$$
  $$\vee$$
  $$inSeconds_{stmt} \leq inSeconds_{start}$$
  otherwise
  $$start(state, stmt) = start_{state}$$

- does end_{state} need to be updated to $currentTime$?

  where
  $$inSeconds_{stmt} = isoToUnixEpoch(currentTime)$$
  $$inSeconds_{end} = isoToUnixEpoch(end_{state}) \iff end_{state} \neq nil$$
  such that
  $$end(state, stmt) = currentTime$$
  $$\iff$$
  $$end_{state} = nil$$
  $$\vee$$
  $$inSeconds_{stmt} \geq inSeconds_{end}$$
  otherwise
  $$end(state, stmt) = end_{state}$$

- what should $statementCount$ be?

$$nStmts(state) = 1 \iff statementCount_{state} = 0 \ \vee \ nil$$

$$\vee$$

$$nStmts(state) = 1 + statementCount_{state} \iff statementCount_{state} \geq 1$$

- do we need to add a new $name$?

$$allNames(state, stmt) = append(name_{state}, name_{stmt}, count(name_{state}))$$

$$\iff$$

$$name_{stmt} \notin name_{state}$$

otherwise

$$allNames(state, stmt) = name_{state}$$

which allows for the following primitives to be defined

$$p_{start}(state, stmt) = start(state, stmt)$$

$$p_{end}(state, stmt) = end(state, stmt)$$

$$p_{stmtCount}(state, stmt) = nStmts(state)$$

$$p_{names}(state, stmt) = allNames(state, stmt)$$

and establish relevant paths into $state$

$$K_{domain} =< state, completions, objectId, domain >$$

$$K_{stmtCount} =< state, completions, objectId, statementCount >$$

$$K_{names} =< state, completions, objectId, name >$$

which are used within higher level primitives concerned with updating $state$

$$p_{updateStart}(state, stmt)$$

$$\equiv$$

$$associate(state, K_{domain}, append(remove(domain_{state}, 0), p_{start}(state, stmt), 0))$$

and

$$p_{updateEnd}(state, stmt)$$

$$\equiv$$

$$associate(state, K_{domain}, append(remove(domain_{state}, 1), p_{end}(state, stmt), 1))$$

and

$$p_{updatedCount}(state, stmt)$$

$$\equiv$$

$$associate(state, K_{stmtCount}, p_{stmtCount}(state, stmt))$$

and

$$p_{updatedNames}(state, stmt)$$

$$\equiv$$

$$associate(state, K_{names}, p_{names}(state, stmt))$$

such that *body* of *step* is defined as

$$step(state, stmt) = p_{updateNames}(p_{updateCount}(p_{updateEnd}(p_{updateStart}(state, stmt), stmt), stmt), stmt)$$

where

$$state' = p_{updateStart}(state, stmt)$$

and

$$state'' = p_{updateEnd}(state', stmt)$$

and

$$state''' = p_{updateCount}(state'', stmt)$$

such that

$$step(state, stmt) = p_{updateNames}(state''', stmt)$$

8.5 Result

The only *opts* used by $rateOfCompletions$ is $timeUnit$

$$timeUnit = second \ \vee minute \ \vee \ \ hour \ \vee day \ \vee month \ \vee year$$

and will default to *day* if not passed to $rateOfCompletions$

$$result(state) = result(state, < timeUnit \mapsto day >)$$

which is passed to $rateOf$ along with the arguments parsed from *state*

$$unit = atKey(opts, timeUnit)$$

$$allCompletions(state) = atKey(state, < state, completions >)$$

such that

$$\forall k_n : i..n..j \in allCompletions(state)$$

the following primitives are called each iteration

$$getCount(state, k_n) = atKey(allCompletions(state), < k_n, statementCount >)$$

$$getStart(state, k_n) = atKey(allCompletions(state), < k_n, domain, start >)$$

$$getEnd(state, k_n) = atKey(allCompletions(state), < k_n, domain, end >)$$

$$getName(state, k_n) = atKey(allCompletions(state), < k_n, name >)$$

which allows for

$$rate_n(state, k_n, unit) = rateOf(getCount(state, k_n), getStart(state, k_n), getEnd(state, k_n), unit)$$

such that

$$value_n(state, k_n, unit) =< x_n, y_n >$$

where

$$name_n(state, k_n) = first(getName(state, k_n))$$

$$x_n = x \mapsto name_n(state, k_n) \iff name_n(state, k_n) \neq nil$$

otherwise

$$x_n = x \mapsto k_n$$

and

$$y_n = y \mapsto rate_n(state, k_n, unit)$$

such that

$$value_n(state, k_n, unit) =< name_n(state, k_n),\ rate_n(state, k_n, unit) >$$

and

$$value(state, unit) = \forall k_n : i..n..j \in allCompletions(state) \exists!\, value_n(state, k_n, unit) =< x_n, y_n >$$

$$\Rightarrow$$

$$value(state, unit) =< value_i(state, k_i, unit)..value_n(state, k_n, unit)..value_j(state, k_j, unit) >$$

which allows the body of *result* to be defined using

$$unit = atKey(opts, timeUnit)$$

$$K_{store} =< state, completions, values, unit >$$

so that *result* returns an updated *state* with the rate of completions per *unit* located at K_{store}

$$result(state, opts) = associate(state, K_{store}, value(state, unit))$$

9 Timeline Of Learner Success

Intro text about the Algorithm

9.1 Initialization

What does $state_0$ look like?

9.2 Relevant?

What primitives are used to determine if a Statement is relevant

9.3 Accept?

What primitives are used to determine if a Statement is accepted

9.4 Step

What primitives are used to process a Statement to update *state*

9.5 Result

What *opts* are used if any + what does the *state* look like?

10 Which Assessment Questions are the Most Difficult

Intro text about the Algorithm

10.1 Initialization

What does $state_0$ look like?

10.2 Relevant?

What primitives are used to determine if a Statement is relevant

10.3 Accept?

What primitives are used to determine if a Statement is accepted

10.4 Step

What primitives are used to process a Statement to update *state*

10.5 Result

What *opts* are used if any + what does the *state* look like?

11 How Often are Recommendations Followed

Intro text about the Algorithm

11.1 Initialization

What does $state_0$ look like?

11.2 Relevant?

What primitives are used to determine if a Statement is relevant

11.3 Accept?

What primitives are used to determine if a Statement is accepted

11.4 Step

What primitives are used to process a Statement to update *state*

11.5 Result

What *opts* are used if any + what does the *state* look like?

Previous Algorithm Definitions

The following are examples of the previous way in which Algorithms were defined.

12 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), it becomes possible to retell their learning story through data and data visualization. One important aspect of that story is the rate of completion[1] of the various digital resources within the learning ecosystem.

12.1 Ideal Statements

In order to accurately portray the rates of completion, there are a few base requirements of the data produced by a Learning Record Provider (LRP). They are as follows:

- statements describing a learner completing an activity should[2] use the verb $http://adlnet.gov/expapi/verbs/completed$
- statements describing a learner completing an activity should report if the learner was successful or not via $\$.result.success$
- statement describing a learner completing a scored activity should report the learners score via $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$
- activites must be uniquely and consistently identified across all statements
- The time at which a learner completed a learning activity must be recorded
  - The timestamp should contain an appropriate level of specificity.
  - ie. Year, Month, Day, Hour, Minute, Second, Timezone
- statements describing a learner completing an activity should report the amount of time taken to complete the activity via $\$.result.duration$

12.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource via curl. The following section contains the appropriate parameters with example values as well as the curl command necessary for making the request.[3][4][5]

[1] Completion can be defined by the presence of the verb completed or by the presence of $\$.result.completion$ set equal to true. In this algorithm, completion is defined by the presence of the verb completed regardless of $\$.result.completion$. This decision affects how statements are retrieved and filtered. In the case where completion is defined by $\$.result.completion$, the query to the LRS would not include the verb paramater and there would need to be a filtering process which looks for the presence of $\$.result.completion$ = true

[2] See footnote 4

[3] See footnote 1.

[4] See footnote 2.

[5] See footnote 3.

```
Verb = "verb=http://adlnet.gov/expapi/verbs/completed"

Since = "since=2018-07-20T12:08:47Z"

Until = "until=2018-07-21T12:08:47Z"

Base = "https://example.endpoint/statements?"

endpoint = Base + Verb + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

S = curl -X GET -H "Authorization: Auth"
         -H "Content-Type: application/json"
         -H "X-Experience-API-Version: 1.0.3"
         Endpoint
```

12.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the $

- $\$.timestamp$
- $\$.object.id$

12.4 2018 Pilot TLA Statement Problems

The initial pilot test data supports the core requirements of this algorithm but completion statements only reports completion scores via $\$.result.scaled$ instead of $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$.[6] Given that the offical 2018 pilot test is scheduled to take place on July 27th, 2018, this section may require updates pending future data review.

12.5 Summary

1. Query an LRS via a GET request to the statemetns endpoint using the paramters verb, since and until.

[6] The one potential issue with using scaled score is the calculation of scaled is not stricly defined by the xAPI specification but is instead up to the authors of the LRP. This results in the inability to reliably compare scaled scores across LRPs. if $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$ are reported for all questions, it becomes possible to reliably compare scores across LRPs by generating a scaled score in a consistent way.

2. group statements by their $\$.object.id$

3. select time range unit for use within rate calculation. Will default to day.

4. determine the amount of time between the first and last instance of a $\$.object.id$ (in seconds) and divide it by the time unit. ie if the unit is minute, you would divide by 60.

5. calculate the rate by dividing the count of a group (2) by the number of time units covered by the statements (4) so that the rate is the number of completions per activity per time unit.

12.6 Formal Specification

12.6.1 Basic Types

$TIMEUNIT ::== \{second\}|\{minute\}|\{hour\}|\{day\}|\{week\}|\{month\}|\{year\}$

12.6.2 System State

$$
\begin{array}{|l}
\hline
\textit{RateOfCompletion} \\
Statements \\
S_{completions} : \mathbb{F}_1 \\
S_{grouped}, S_{timeunit}, S_{processed} : \mathbb{F} \\
\hline
S_{completions} = statements \\
S_{grouped} = \{byId : \mathrm{seq}_1\ statement\} \\
S_{withRate} = \{byGroup : (\mathrm{seq}_1\ statement, \mathbb{N})\} \\
S_{processed} = \{rate : (id, \mathbb{N}, TIMEUNIT)\} \\
\hline
\end{array}
$$

- The set $S_{completions}$ is a non-empty, finite set and is the component $statements$ which contains the results of the query to the LRS.

- The sets $S_{grouped}$, $S_{withRate}$ and $S_{processed}$ are all finite sets

- the set $S_{grouped}$ is a finite set of objects $byId$ which are non-empty, finite sequences of the component *statement*

- the set $S_{withRate}$ is a finite set of objects *byGroup* which are ordered pairs of non-empty, finite sequences of the component *statement* and a natural number

- the set $S_{processed}$ is a finite set of objects $rate$ where each contains the component id, a natural number and the type $TIMEUNIT$

12.6.3 Initial System State

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad InitRateOfCompletion \qquad\qquad\qquad\qquad\qquad}} \\
RateOfCompletion \\
T : TIMEUNIT \\
\hline
S_{completions} \neq \emptyset \\
S_{grouped} = \emptyset \\
S_{withRate} = \emptyset \\
S_{processed} = \emptyset \\
T = \{day\} \\
\hline
\end{array}
$$

- The set $S_{completions}$ is a non-empty set which contains the results of the GET request(s) to the LRS
- The sets $S_{grouped}$, $S_{withRate}$ and $S_{processed}$ are all initially empty
- the variable T has the type $TIMEUNIT$ and the value $\{day\}$

12.6.4 Calculate Rate

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad IsoToUnix \qquad\qquad\qquad\qquad\qquad}} \\
convert : \mathbb{F}_1 \rightarrow \mathbb{N}\#1 \\
c? : \mathbb{F}_1 \\
c! : \mathbb{N}\#1 \\
\hline
c! = convert(c?) \\
\hline
\end{array}
$$

- The schema $IsoToUnix$ introduces the function *convert* which takes in a finit set of one thing (a timestamp) and converts it to a single natural number.
- the purpose of this function is to convert an ISO 8601 timestamp to the Unix epoch. The concrete definition of the conversion is outside the scope of this document
  - The Unix epoch is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT), not counting leap seconds.

$$
\begin{array}{|l}
\hline
\textit{CalcRateByUnit} \\
\textit{Statement} \\
\textit{IsoToUnix} \\
\textit{CountPerGroup} \\
unit? : TIMEUNIT \\
s?\,, s! : \mathbb{F} \\
r : \mathbb{N} \\
rate : (\mathbb{F}, TIMEUNIT) \rightarrow \mathbb{F} \\
\hline
unit? = \{second\} \Rightarrow 1 \lor \{minute\} \Rightarrow 60 \lor \{hour\} \Rightarrow 3600 \lor \\
\qquad\qquad \{day\} \Rightarrow 86400 \lor \{week\} \Rightarrow 604800 \lor \\
\qquad\qquad \{month\} \Rightarrow 2629743 \lor \{year\} \Rightarrow 31556926 \\
s? = \{g : seq_1\, statement\} \\
s! = rate(s?\,, unit?\,) \\
s! = \{s : (g, r) \mid \forall g_n : g_i .. g_j \bullet i \leq n \leq j \bullet \exists\, s_n : (g_n, r_n) \bullet \\
\qquad r_n = count(g_n) \div ((convert(last\, g_n.timestamp) - convert(head\, g_n.timestamp)) \div unit?\,)\} \\
\hline
\end{array}
$$

- The schema $CalcRateByUnit$ introduces the function $rate$ where the input $s?$ is a set of objects g which are each a non-empty, finite sequence of statements and the input $unit?$ represents a unit of time.

- for every g_n within the range $g_i..g_j$, there exists an associated object s_n which is an orderd pair of (g_n, r_n) where r_n is equal to the number of items within g_n divided by the number of $unit?$s within the time range of $last\, g_n.timestamp - head\, g_n.timestamp$

- the output of the function $rate$ is $s!$, the set of all s_n

12.6.5 Processes Results

$$
\begin{array}{l}
\textbf{AggergateCompletionStatements} \\
\hline
\Delta RateOfCompletion \\
GroupByActivityId \\
CalcRateByUnit \\
grouped, processed, withRate : \mathbb{F} \\
r : \mathbb{N} \\
T? : TIMEUNIT \\
\hline
T? = \{day\} \\
grouped = \emptyset \\
grouped' = group(S_{completions}) \\
S'_{grouped} = S_{grouped} \cup grouped' \\
withRate \subseteq S'_{grouped} \\
withRate' = rate(withRate, T?) \\
S'_{withRate} = withRate' \cup S_{withRate} \\
processed \subseteq S'_{withRate} \\
processed' = \{p : (id, r, T?) \mid \\
\qquad \textbf{let } \{processed_i..processed_j\} == \{b_i..b_j\} \bullet \\
\qquad \forall b_n : b_i..b_j \bullet i \leq n \leq j \bullet \exists p_n : (id_n, r_n, T?) \bullet \\
\qquad id_n = (head\,(first\,b_n)).object.id \wedge \\
\qquad r_n = (second\,b_n)\} \\
S'_{processed} = processed' \cup S_{processed} \\
\hline
\end{array}
$$

- The schema *AggergateCompletionStatements* outlines how to calculate the rate of completion per $\$.object.id$ per $second|minute|hour|day|week|month|year$

  1. $S'_{grouped}$ is the result of grouping the statements within $S_{completions}$ by their $\$.object.id$
  2. The groups from (1) are passed to the function *rate* with the variable $T?$ which controls the unit of time, ie per day vs per week
  3. the result of (2) is then processed to create a triplet of $\$.object.id$, rate, unit of time for all unique $\$.object.id$ within $S_{completions}$

12.6.6 Return

$$
\begin{array}{l}
\textbf{ReturnAggergateCompletionStatements} \\
\hline
\Xi RateOfCompletion \\
AggergateCompletionStatements \\
S_{processed}! : \mathbb{F} \\
\hline
S_{processed}! = S_{processed} \\
\hline
\end{array}
$$

- The return value $S_{processed}!$ is equal to $S_{processed}$ after the operation described by *AggergateCompletionStatements*

12.7 Pseudocode

Algorithm 1: Rate of Completions

```
Input: S_completed, timeUnit
Result: ratePerObjTu'
context = {};
ratePerObjTu = [];
while S_completion ≠ ∅ do
    foreach s ∈ S_completion do
        id ← s.object.id;
        ts ← convert(s.timestamp);
        if id ∉ context then
            do
            times = [ts];
            context' ← {id : times};
            S'_completion ← S_completion \ s;
            recur context', S'_completion;
        else
            do
            times' ← context.id ⌢ ts;
            context' ← {id : times'};
            S'_completion ← S_completion \ s;
            recur context', S'_completion;
        end
    end
end
foreach k ∈ context' do
    allTs ← context'.k;
    totalDuration ← max(allTs) − min(allTs);
    totalCount ← count(allTs);
    rate ← totalCount ÷ (totalDuration ÷ timeUnit);
    subVec = [k, rate, timeUnit];
    ratePerObjTu' ← ratePerObjTu ⌢ subVec;
    recur ratePerObjTu';
end
return ratePerObjTu'
```

- Values from Z schemas are used within this pseudocode
- the result of the algorithm is an array of arrays where each subarray contains a *statement.object.id*, the *rate* and the *timeUnit* used to calculate *rate*.

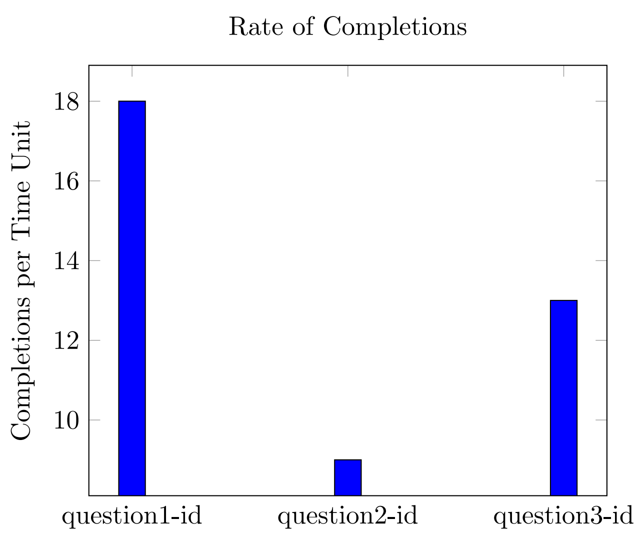
12.8 JSON Schema

```
{"type":"array",
   "items":{"type":"array",
      "items":[{"type":"string"}, {"type":"number"},
{"type":"string"}]}}
```

12.9 Visualization Description

The **Rate of Completions** visualization will be a bar chart where the domain consists of *statement.object.id* and the range is a number greater than 0 (the rate of completions for that *statement.object.id*). Every subarray within the array $ratePerObjTu$ will be a grouping within the bar chart. The pseudocode specifies an input paramter $timeUnit$ which controls the calculation of the rate (range of the visualization). $timeUnit$ could be per minute, per day, per week, etc.

12.10 Visualization prototype



12.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but they would require adding aditional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement paramters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

- use $statement.object.definition.name$ instead of $statement.object.id$ for x axis label
- populate a tooltip with the people who have completed the activity. This could also include the number of times they have completed it.
- populate a tooltip with the breakdown of which devices or platforms the activity was completed on. This would require the device type or platform to be reported within $statement.context.platform$
- populate a tooltip with the breakdown of percentage successful for all completions of the activty. This would require $statement.result.success$
- populate a tooltip with the breakdown of scores earned (if appliciable) for the completions. This would require $statement.result.score.raw$, $statement.result.score.min$ and $statement.result.score.max$
- populate a tooltip with the competency assocaited with the completed activities. The competency should be reported via $statement.context.contextActivities$
- populate a tooltip with the average duration spent to reach completions. This would require $statement.result.duration$ to be reported.

13 Timeline Of Learner Success

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), it becomes possible to retell their learning story through data and data visualization. One important aspect of that story is the' learners history of success.

13.1 Ideal Statements

In order to accurately portray a learner's timeline of success, there are a few base requirements of the data produced by a Learning Record Provider (LRP). They are as follows:

- the learner must be uniquely and consistently identified across all statements
- learning activities which evaluate a learner's understanding of material must report if the learner was successful or not
  - the grade earned by the learner must be reported
  - the minimum and maximum possible grade must be reported
- The learning activities must be uniquely and consistently identified across all statements
- The time at which a learner completed a learning activity must be recorded
  - The timestamp should contain an appropriate level of specificity.
  - ie. Year, Month, Day, Hour, Minute, Second, Timezone

13.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource via curl. The following section contains the appropriate parameters with example values as well as the curl command necessary for making the request.[7][8][9]

[7] S is the set of all statements parsed from the statements array within the HTTP response to the Curl request(s). It may be possible that multiple Curl requests are needed to retrieve all query results. If multiple requests are necessary, S is the result of concatenating the result of each request into a single set

[8] Querying an LRS will not be defined within the following Z specifications but the results of the query will be utilized

[9] If you want to query across the entire history of a LRS, omit Since and Until from the endpoint(s) and remove the associated & symbols.

```
Agent = "agent={"account":
  {"homePage": "https://example.homepage",
    "name": 123456}}"

Since = "since=2018-07-20T12:08:47Z"

Until = "until=2018-07-21T12:08:47Z"

Base = "https://example.endpoint/statements?"

endpoint = Base + Agent + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

S = curl -X GET -H "Authorization: Auth"
-H "Content-Type: application/json"
-H "X-Experience-API-Version: 1.0.3"
Endpoint
```

13.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the $

- $.timestamp$
- $.result.success$
- $.result.score.raw$
- $.result.score.min$
- $.result.score.max$
- $.verb.id$

13.4 2018 Pilot TLA Statement Problems

The initial pilot test data supports this algorithm. This section may require updates pending future data review following iterations of the TLA testing.

13.5 Summary

1. Query an LRS via a GET request to the statements endpoint using the parameters agent, since and until

2. Filter the results to the set of statements where:
   - $\$.verb.id$ is one of:
     - http://adlnet.gov/expapi/verbs/passed
     - https://w3id.org/xapi/dod-isd/verbs/answered
     - http://adlnet.gov/expapi/verbs/completed
   - $\$.result.success$ is true
3. process the filtered data
   - extract $\$.timestamp$
   - extract the score values from $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$ and convert them to the scale 0..100
   - create a pair of [$\$.timestamp$, #]

13.6 Formal Specification

13.6.1 Basic Types

$$COMPLETION ::= \\ \{http://adlnet.gov/expapi/verbs/passed\} \mid \\ \{https://w3id.org/xapi/dod-isd/verbs/answered\} \mid \\ \{http://adlnet.gov/expapi/verbs/completed\}$$

$$SUCCESS ::= \{true\}$$

13.6.2 System State

$$\begin{array}{l} \textit{TimelineLearnerSuccess} \\ \hline Statements \\ S_{all} : \mathbb{F}_1 \\ S_{completion}, S_{success}, S_{processed} : \mathbb{F} \\ \hline S_{all} = statements \\ S_{completion} \subseteq S_{all} \\ S_{success} \subseteq S_{completion} \\ S_{processed} = \{pair : (statement.timestamp, \mathbb{N})\} \\ \hline \end{array}$$

- The set S_{all} is a non-empty, finite set and is the component *statements*
- The sets $S_{completion}$ and $S_{success}$ are both finite sets
- the set $S_{completion}$ is a subset of S_{all} which may contain every value within S_{all}
- the set $S_{success}$ is a subset of $S_{completion}$ which may contain every value within $S_{completion}$
- the set $S_{processed}$ is a finite set of pairs where each contains a *statement.timestamp* and a natural number

13.6.3 Initial System State

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitTimelineLearnerSuccess}} \\
\hline
TimelineLearnerSuccess \\
\hline
S_{all} \neq \emptyset \\
S_{completion} = \emptyset \\
S_{success} = \emptyset \\
S_{processed} = \emptyset \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty set
- The sets $S_{completion}, S_{success}$ and $S_{processed}$ are all initially empty

13.6.4 Filter for Completion

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Completion}} \\
\hline
Statement \\
completion : STATEMENT \rightarrow\!\!\!\!\!\!+ \; \mathbb{F} \\
s? : STATEMENT \\
s! : \mathbb{F} \\
\hline
s? = statement \\
s! = completion(s?) \\
completion(s?) = \textbf{if } s?.verb.id : COMPLETION \\
\qquad\qquad\qquad\qquad \textbf{then } s! = s? \\
\qquad\qquad\qquad\qquad \textbf{else } s! = \emptyset \\
\hline
\end{array}
$$

- The schema *Completion* inroduces the function *completion* which takes in the variable $s?$ and returns the variable $s!$
- The variable $s?$ is the component *statement*
- $s!$ is equal to $s?$ if $\$.verb.id$ is of the type $COMPLETION$ otherwise $s!$ is an empty set

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FilterForCompletion}} \\
\hline
\Delta TimelineLearnerSuccess \\
Completion \\
completions : \mathbb{F} \\
\hline
completions \subseteq S_{all} \\
completions' = \{s : STATEMENT \mid completion(s) \neq \emptyset\} \\
S'_{completion} = S_{completion} \cup completions' \\
\hline
\end{array}
$$

- the set *completions* is a subset of S_{all} which may contain every value within S_{all}

- The set $completions'$ is the set of all statements s where the result of $completion(s)$ is not an empty set
- the updated set $S'_{completion}$ is the union of the previous state of set $S_{completion}$ and the set $completions'$

13.6.5 Filter for Success

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Success}} \\
\hline
Statement \\
success : STATEMENT \nrightarrow \mathbb{F} \\
s? : STATEMENT \\
s! : \mathbb{F} \\
\hline
s? = statement \\
s! = success(s?) \\
success(s?) = \textbf{if } s?.result.success : SUCCESS \\
\qquad\qquad\qquad \textbf{then } s! = s? \\
\qquad\qquad\qquad \textbf{else } s! = \emptyset \\
\hline
\end{array}
$$

- the schema $Success$ introduces the function $success$ which takes in the variable $s?$ and returns the variable $s!$
- the variable $s?$ is the component $statement$
- $s!$ is equal to $s?$ if $\$.result.success$ is of the type $SUCCESS$ otherwise $s!$ is an empty set

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FilterForSuccess}} \\
\hline
\Delta TimelineLearnerSuccess \\
Success \\
successes : \mathbb{F} \\
\hline
successes \subseteq S_{completion} \\
successes' = \{s : STATEMENT \mid success(s) \neq \emptyset\} \\
S'_{success} = S_{success} \cup successes' \\
\hline
\end{array}
$$

- the set $successes$ is a subset of $S_{completion}$ which may contain every value within $S_{completion}$
- The set $successes'$ contains elements s of type $STATEMENT$ where $success(s)$ is not an empty set
- The updated set $S'_{success}$ is the union of the previous state of $S_{success}$ and $successes'$

13.6.6 Processes Results

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Scale \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
scaled! : \mathbb{N} \\
raw?, min?, max? : \mathbb{Z} \\
scale : \mathbb{Z} \rightarrow \mathbb{N} \\
\hline
scaled! = scale(raw?, min?, max?) \\
scale(raw?, min?, max?) = \\
\qquad\qquad\qquad\qquad (raw? * ((0.0 - 100.0)\, div\, (min? - max?))) + \\
\qquad\qquad\qquad\qquad (0.0 - (min? * ((0.0 - 100.0) div (min? - max?)))) \\
\hline
\end{array}
$$

- The schema $Scale$ introduces the function $scale$ which takes 3 arguments, $raw?$, $min?$ and $max?$. The function converts $raw?$ from the range $min?..max?$ to 0.0..100.0

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ProcessStatements \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta TimelineLearnerSuccess \\
Scale \\
FilterStatements \\
processed : \mathbb{F} \\
\hline
processed \subseteq S_{success} \\
processed' = \{p : (\mathbb{F}_1\, \#1, \mathbb{N}) \mid \\
\qquad\qquad\qquad \mathbf{let}\ \{processed_i..processed_j\} == \{s_i..s_j\} \bullet \\
\qquad\qquad\qquad i \leq n \leq j \bullet \forall s_n : s_i..s_j \bullet \exists p_n : p_i..p_j \bullet \\
\qquad\qquad\qquad first\, p_n = s_n.timestamp \wedge \\
\qquad\qquad\qquad second\, p_n = scale(s_n.result.score.raw, \\
\qquad\qquad\qquad\qquad\qquad\qquad s_n.result.score.min, \\
\qquad\qquad\qquad\qquad\qquad\qquad s_n.result.score.max)\} \\
S'_{processed} = S_{processed} \cup processed' \\
\hline
\end{array}
$$

- The operation $ProcessStatements$ introduces the variable $processed$ which is a subset of $S_{success}$ which may contain every value within $S_{success}$
- $S_{success}$ is the result of the operation $FilterStatements$
- The operation defines the variable $processsed'$ which is a set of objects p which are ordered pairs of (1) a finite set containing one value and (2) a single positive number.
- The first component of every object p, is the timestamp from the associated *statement* within *processed* ie. $s.timestamp$
- The second component of every object p is the result of the function $scale$. The score values contained within the associated *statement* s are the arguments passed to $scale$. ie $scale(s.result.score.raw, s.result.score.min, s.result.score.max)$
- The result of the operation $ProcessStatements$ is to updated the set $S_{processed}$ with the values contained within $processed'$

13.6.7 Sequence of Operations

$FilterStatements \mathrel{\widehat=} FilterForCompletion \mathbin{;} FilterForSuccess$

- The schema $FilterStatements$ is the sequential composition of operation schemas $FilterForCompletion$ and $FilterForSuccess$
- $FilterForCompletion$ happens before $FilterForSuccess$

$ProcessedStatements \mathrel{\widehat=} FilterStatements \mathbin{;} ProcessStatements$

- The schema $ProcessedStatements$ is the sequential composition of operation schemas $FilterStatements$ and $ProcessStatements$
- $FilterStatements$ happens before $ProcessStatements$

13.6.8 Return

$$
\begin{array}{|l}
\hline
\textit{Return} \\
\Xi TimelineLearnerSuccess \\
ProcessedStatements \\
S_{processed}! : \mathbb{F} \\
\hline
S_{processed}! = S_{processed} \\
\hline
\end{array}
$$

- The returned variable $S_{processed}!$ is equal to the current state of variable $S_{processed}$ after the operations $FilterForCompletion$, $FilterForSuccess$ and $ProcessStatements$

13.7 Pseudocode

Algorithm 2: Timeline of Learner Success

```
Input: S_all
Result: coll'
coll = [];
while S_all ≠ ∅ do
    foreach s ∈ S_all do
        if s.verb.id = COMPLETION then
            do
            S'_completion ← s ∪ S_completion;
            S'_all ← S_all \ s ;
            recur S'_completion, S'_all;
        else
            do
            S'_all ← S_all \ s;
            recur S'_all;
        end
    end
end
while S'_completion ≠ ∅ do
    foreach sc ∈ S'_completion do
        if sc.result.success = SUCCESS then
            do
            S'_success ← sc ∪ S_success;
            S'_completion ← S_completion \ sc;
            recur S'_success, S'_completion;
        else
            do
            S'_completion ← S_completion \ sc;
            recur S'_completion;
        end
    end
end
foreach ss ∈ S'_success do
    raw? ← ss.result.score.raw;
    max? ← ss.result.score.max;
    min? ← ss.result.score.min;
    scaled ← scale(raw?, min?, max?);
    subVec ← [ss.timestamp, scaled];
    coll' ← coll ⌢ subVec;
    recur coll'
end
return coll'
```

- The Z schemas are used within this pseudocode
- The return value coll is an array of arrays, each containing a *statement.timestamp* and a scaled score.

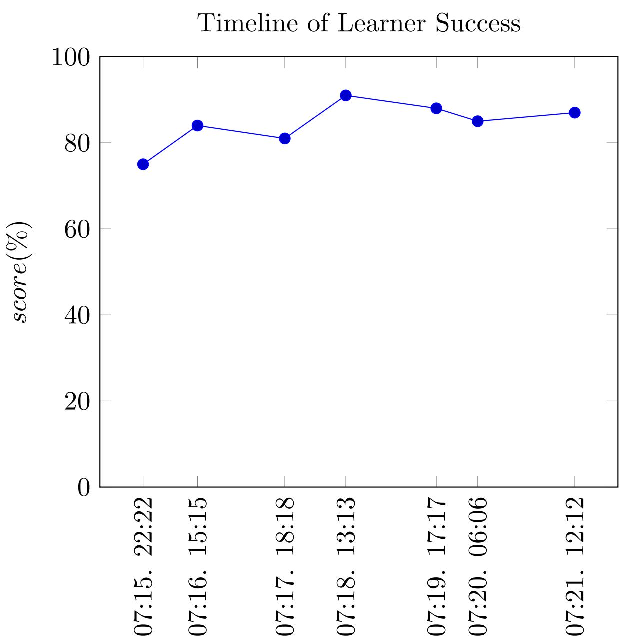
13.8 JSON Schema

```
{"type":"array",
  "items":{"type":"array",
    "items":[{"type":"string"}, {"type":"number"}]}}
```

13.9 Visualization Description

The **Timeline of Learner Success** visualization will be a line chart where the domain is time and the range is score on a scale of 0.0 to 100.0. Every subarray will be a point on the chart. The domain of the graph should be in chronological order.

13.10 Visualization prototype



13.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but they would require adding aditional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement paramters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

- A tooltip containing the name of an activity when hovering over a specific point on the chart
  - this would require utilizing $\$.object.definition.name$
- A tooltip containing the device on which the activity was experienced
  - this would require utilizing $\$.context.platform$
- A tooltip containing the instructor associated with a particular data point
  - this would require utilizing $\$.context.instructor$

14 Which Assessment Questions are the Most Difficult

As learners engage in activities supported by a learning ecosystem, they will experience learning content as well as assessment content. Assessments are designed to measure the effectiveness of learning content and help assess knowledge gained. It is possible that certain assessment questions do not accurately represent the concepts contained within learning content and this may be indicated by a majority of learners getting the question wrong. It is also possible that the question accurately represents the learning content but is very difficult. The following algorithm will identify these types of questions but will not be able to deduce why learners answer them incorrectly.

14.1 Ideal Statements

In order to accurately determine which assessment questions are the most dificult, there are a few requirements of the data produced by a LRP. They are as follows:

- statements describing a learner answering a question must report if the learner got the question correct or incorrect via $\$.result.success$
- if it is possible to get partial credit on a question, the amount of credit should be reported within the statement
  - the credit earned by the learner should be reported within $\$.result.score.raw$
  - the minimum and maximum possible credit amount should be reported within $\$.result.score.min$ and $\$.result.score.max$ respectively
- If it is possible to get partial credit on a question, it must still be reported if the learner reached the threshold of success via $\$.result.success$
- Statements describing a learner answering a question should contain activities of the type *cmi.interaction*
- activities must be uniquely and consistently identified across all statements
- Statements describing a learner answering a question should[10] use the verb $http://adlnet.gov/expapi/verbs/answered$

[10] it is possible to use another verb iri but if another is used, that will need to be accounted for in data retrieval

14.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource via curl. The following section contains the appropriate parameters with example values as well as the curl command necessary for making the request.[11][12][13]

```
Verb = "verb=http://adlnet.gov/expapi/verbs/answered"

Since = "since=2018-07-20T12:08:47Z"

Until = "until=2018-07-21T12:08:47Z"

Base = "https://example.endpoint/statements?"

endpoint = Base + Verb + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

S = curl -X GET -H "Authorization: Auth"
-H "Content-Type: application/json"
-H "X-Experience-API-Version: 1.0.3"
Endpoint
```

14.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the $

- $\$.result.success$
- $\$.object.id$

14.4 2018 Pilot TLA Statement Problems

The initial pilot test data supports this algorithm. Given that the offical 2018 pilot test is scheduled to take place on July 27th, 2018, this section may require updates pending future data review.

14.5 Summary

1. Query an LRS via a GET request to the statements endpoint using the parameters verb, since and until

[11] See footnote 1.
[12] See footnote 2.
[13] See footnote 3.

2. Filter the results to the set of statements where:
   - $\$.result.success$ is false
3. process the filtered data
   - group by $\$.object.id$
   - determine the count of each group
   - create a collection of pairs = [$\$.object.id$, #]

14.6 Formal Specification

14.6.1 Basic Types

$INCORRECT ::= \{false\}$

14.6.2 System State

$$
\begin{array}{|l}
\hline
MostDifficultAssessmentQuestions \\
Statements \\
S_{all} : \mathbb{F}_1 \\
S_{incorrect}, S_{grouped}, S_{processed} : \mathbb{F} \\
\hline
S_{all} = statements \\
S_{incorrect} \subseteq S_{all} \\
S_{grouped} = \{groups : \mathrm{seq}_1\ statement\} \\
S_{processed} = \{pair : (id, \mathbb{N})\} \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty, finite set and is the component *statements*
- The sets $S_{incorrect}$, $S_{grouped}$ and $S_{processed}$ are all finite sets
- the set $S_{incorrect}$ is a subset of S_{all} which may contain every value within S_{all}
- the set $S_{grouped}$ is a finite set of objects *groups* which are non-empty, finite sequences of the component *statement*
- the set $S_{processed}$ is a finite set of pairs where each contains the component *id* and a natural number

14.6.3 Initial System State

$$
\begin{array}{|l}
\hline
InitMostDifficultAssessmentQuestions \\
MostDifficultAssessmentQuestions \\
\hline
S_{all} \neq \emptyset \\
S_{incorrect} = \emptyset \\
S_{grouped} = \emptyset \\
S_{processed} = \emptyset \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty set
- The sets $S_{incorrect}$, $S_{grouped}$ and $S_{processed}$ are all initially empty

14.6.4 Filter for Incorrect

$$
\begin{array}{|l}
\hline
\textit{Incorrect} \\
\textit{Statement} \\
incorrect : STATEMENT \rightarrow\!\!\!\!\!+ \mathbb{F} \\
s? : STATEMENT \\
s! : \mathbb{F} \\
\hline
s? = statement \\
s! = incorrect(s?) \\
incorrect(s?) = \textbf{if } s?.result.success : INCORRECT \\
\qquad\qquad\qquad\quad \textbf{then } s! = s? \\
\qquad\qquad\qquad\quad \textbf{else } s! = \emptyset \\
\hline
\end{array}
$$

- the schema *Incorrect* introduces the function *incorrect* which takes in the variable $s?$ and returns the variable $s!$
- the variable $s?$ is the component *statement*
- $s!$ is equal to $s?$ if $\$.result.success$ is of the type $INCORRECT$ otherwise $s!$ is an empty set

$$
\begin{array}{|l}
\hline
\textit{FilterForIncorrect} \\
\Delta MostDifficultAssessmentQuestions \\
\textit{Incorrect} \\
incorrects : \mathbb{F} \\
\hline
incorrects \subseteq S_{all} \\
incorrects' = \{s : STATEMENT \mid incorrect(s) \neq \emptyset\} \\
S'_{incorrect} = S_{incorrect} \cup incorrects' \\
\hline
\end{array}
$$

- the set *incorrects* is a subset of S_{all} which may contain every value within S_{all}
- The set $incorrects'$ contains elements s of type $STATEMENT$ where $incorrect(s)$ is not an empty set
- The updated set $S'_{incorrect}$ is the union of the previous state of $S_{incorrect}$ and $incorrects'$

14.6.5 Processes Results

$$
\begin{array}{|l}
\textit{GroupByActivityId} \\
\hline
\textit{Statements} \\
g? : \mathbb{F} \\
g! : \mathbb{F} \\
\textit{group} : \mathbb{F} \rightarrow \mathbb{F} \\
\hline
g? = \textit{statements} \Rightarrow \{g : \textit{statement}\} \\
g! = \textit{group}(g?) \\
g! = \{\textit{groups} : \mathrm{seq}_1\, \textit{statement} \mid \\
\quad \textbf{let } \mathrm{seq}_1\, \textit{statement}_i..\textit{statement}_j == \mathrm{seq}_1\, s_i..s_j \bullet \\
\quad \forall s_n : s_i..s_j \bullet i \leq n \leq j \bullet s_i.\textit{object.id} = s_j.\textit{object.id} = s_n.\textit{object.id}\} \\
\hline
\end{array}
$$

- The schema *GroupByActivityId* introduces the function *group* which has the input of $g?$ and the output of $g!$
- The input variable $g?$ is the component *statements* which implies its a set of objects g which are each a *statement*
- the output variable $g!$ is a set of objects *groups* which are each a non-empty, finite sequence of *statement* where each member of the sequence $s_i..s_j$ has the same $\$.object.id$

$$
\begin{array}{|l}
\textit{CountPerGroup} \\
\hline
\textit{Statement} \\
c? : \mathrm{seq}_1\, \textit{statement} \\
c! : \mathbb{N} \\
\textit{count} : \mathrm{seq}_1\, \textit{statement} \rightarrow \mathbb{N} \\
\hline
c! = \textit{count}(c?) \\
c! \geq 1 \\
\textit{count}(c?) = \forall c_n? : \langle c?_i..c?_j \rangle \bullet i \leq n \leq j \wedge i = 0 \bullet \\
\qquad \exists_1\, c! : \mathbb{N} \bullet \textbf{if } n = i \textbf{ then } c! = n + 1 \textbf{ else } c! = j + 1 \\
\hline
\end{array}
$$

- The schema *CountPerGroup* introduces the function *count* which has the input of $c?$ and the output of $c!$
- The input variable $c?$ is a non-empty, finite sequence in which each element is a *statement*
- The function *count* reads: for all elements $c?_n$ within the sequence $\langle c?_i..c?_j \rangle$, such that n is greater than or equal to i and less than or equal to j, i is equal to zero and there exits a number $c!$ which is equal to $n + 1$ (when $n = i \Rightarrow n = 0$) or equal to n

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AggregateQuestionStatements}} \\
\Delta MostDifficultAssessmentQuestions \\
FilterForIncorrect \\
GroupByActivityId \\
CountPerGroup \\
grouped, processed : \mathbb{F} \\
\hline
grouped = \emptyset \\
grouped' = group(S_{incorrect}) \\
S'_{grouped} = S_{grouped} \cup grouped' \\
processed \subseteq S'_{grouped} \\
processed' = \{p : (id, \mathbb{N}) \mid \\
\qquad\qquad \mathbf{let}\ \{\langle processed_i \rangle .. \langle processed_j \rangle\} == \{g_i .. g_j\} \bullet \\
\qquad\qquad i \leq n \leq j \bullet \forall g_n : g_i .. g_j \bullet \exists\, p_n : p_i .. p_j \bullet \\
\qquad\qquad first\, p_n = head\, g_n.object.id \wedge second\, p_n = count(g_n) \\
S'_{processed} = S_{processed} \cup processed' \\
\hline
\end{array}
$$

- The schema *AggregateQuestionStatements* introduces the variables *grouped* and *processed*
- *grouped* starts as an empty set but then becomes $grouped'$ which is the output of applying the function *group* to the set of statements $S_{incorrect}$ created by the opperation $FilterForIncorrect$
- $grouped'$ is a set of sequences. The elements of those sequences are statements which all have the same *statement.object.id*
- The set $S_{grouped}$ is updated to the set $S'_{grouped}$ which is the union of $S_{grouped}$ and $grouped'$
- the variable *processed* is a subset of $S'_{grouped}$ which can contain every value within $S'_{grouped}$
- the variable *processed* is updated to be the variable $processed'$ which is a set of objects p which are ordered pairs of the component *id* and a natural number. p is defined as:
  - for all sequences $g_i..g_j$ within the set *processed*, there exists an ordered pair p_n such that:
    - the first element of p_n is equal to the *object.id* of the first statement within the sequence g_n.
    - The second element of p_n is equal to the value returned when g_n is passed to the function *count*.
- The set $S'_{processed}$ is the union of the sets $S_{processed}$ and $processed'$

14.6.6 Sequence of Operations

$ProcessedQuestions \mathrel{\widehat=} FilterForIncorrect \mathbin{⨾} AggregateQuestionStatements$

- The schema $ProcessedQuestions$ is the sequential composition of operation schemas $FilterForIncorrect$ and $AggregateQuestionStatements$
- $FilterForIncorrect$ happens before $AggregateQuestionStatements$

14.6.7 Return

$$
\begin{array}{|l}
\hline
\textit{ReturnAggregate} \\
\Xi MostDifficultAssessmentQuestions \\
ProcessedQuestions \\
S_{processed}! : \mathbb{F} \\
\hline
S_{processed}! = S_{processed} \\
\hline
\end{array}
$$

- The returned variable $S_{processed}!$ is equal to the current state of variable $S_{processed}$ after the operations $FilterForIncorrect$ and $AggregateQuestionStatements$

14.7 Pseudocode

Algorithm 3: Most Difficult Assessment Questions

Input: S_{all}, $displayN$
Result: $display''$
$context = \{\}$;
$display = []$;
while $S_{all} \neq \emptyset$ **do**
  foreach $s \in S_{all}$ **do**
    if $s.result.success = INCORRECT$ **then**
      do
      $S'_{incorrect} \leftarrow s \cup S_{incorrect}$;
      $S'_{all} \leftarrow S_{all} \setminus s$;
      recur $S'_{all}, S'_{incorrect}$;
    else
      do
      $S'_{all} \leftarrow S_{all} \setminus s$;
      recur S'_{all}
    end
  end
end
while $S'_{incorrect} \neq \emptyset$ **do**
  foreach $si \in S'_{incorrect}$ **do**
    $id \leftarrow si.object.id$;
    if $id \notin context$ **then**
      do
      $count = 1$;
      $context' \leftarrow \{id : count\}$;
      $S'_{incorrect} \leftarrow S_{incorrect} \setminus si$;
      recur $context', S'_{incorrect}$;
    else
      do
      $count' \leftarrow inc(context.id)$;
      $context' \leftarrow \{id : count'\}$;
      $S'_{incorrect} \leftarrow S_{incorrect} \setminus si$;
      recur $context', S'_{incorrect}$;
    end
  end
end
foreach $id \in context'$ **do**
  $IdToCount \leftarrow [id, context.id]$;
  $display' \leftarrow display \frown IdToCount$;
  recur $display'$
end
return $display'' \leftarrow$ **take(sortBySubArray(**$display'$**),** $displayN$**)**

- The Z schemas are used within this pseudocode
- The return value display is an array of length display-n, where each element of display is an array of [*statement.object.id*, #] where # representing the number of times *statement.object.id* appeared within $S'_{incorrect}$

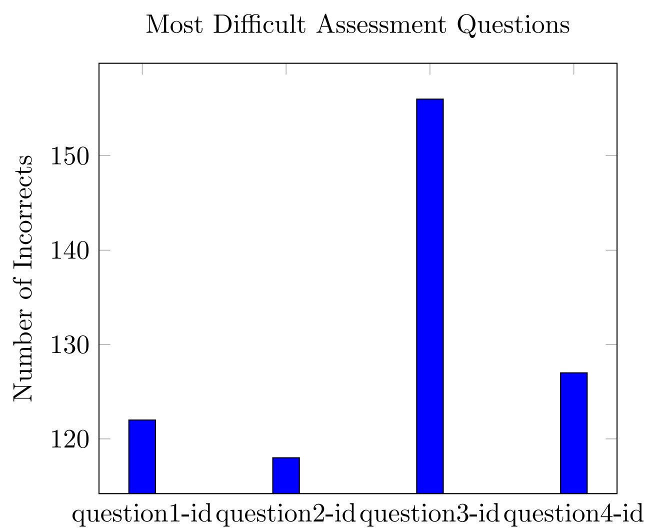
14.8 JSON Schema

```
{"type":"array",
  "items":{"type":"array",
    "items":[{"type":"string"}, {"type":"number"}]}}
```

14.9 Visualization Description

The **Most Difficult Assessment Questions** visualization will be a bar chart where the domain consists of *statement.object.id* and the range is a number greater than or equal to 1. Every subarray within the array display will be a grouping within the bar chart. The pseudocode specifies an input paramter display-n which controls the length of the array display and therefor the number of groups contained within the visualization.

14.10 Visualization prototype



14.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but they would require adding aditional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement paramters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

- Use the name of the activity for the x-axis label instead of its id.
  - $\$.object.definition.name$
  - grouping of statements should still happen by $\$.object.id$ to ensure an accurate count
- a tooltip containing contextual information about the question such as:
  - The question text
    * $\$.object.definition.description$
  - Interaction Type
    * $\$.object.definition$ which contains interaction properties
  - Answer choices
    * $\$.object.definition$ which contains interaction properties
  - Correct answer
    * $\$.object.definition$ which contains interaction properties
  - Most popular incorrect answer
    * This would require an extra step of processing and all statements would need to utilize interaction properties within $\$.object.definition$
  - average partial credit earned (if applicable)
    * $\$.result.score.scaled$
    * The one potential issue with using scaled score is the calculation of scaled is not stricly defined by the xAPI specification but is instead up to the authors of the LRP. This results in the inability to reliably compare scaled scores across LRPs.
    * if $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$ are reported for all questions, it becomes possible to reliably compare scores across questions and LRPs.
  - average number of re-attempts
    * this would require additional steps of processing so that $\$.actor$ is considered as well
    * due to the problem of actor unification, ie the same person being identified differently across statements, this metric may not be accurate.
  - average time spent on the question
    * $\$.result.duration$
    * this would require additional steps of processing to extract the duration and average it.

- a tooltip containing contextual information about the course and/or assessment the question was within
  - the instructor for the course
    - $\$.context.instructor$
  - competency associated with the question and/or course
    - $\$.context.contextActivities$
  - metadata about the learning content associated with the question such as average time spent engaging with associated content before attempting the question.
  - this would require additional steps of processing to retrieve metadata about the content and its usage.
    - $\$.context.contextActivities$

15 How Often are Recommendations Followed

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), it becomes possible to retell their learning story through data and data visualization. One important aspect of that story is the recommendations provided to the learner and whether or not the learner follows those recommendations.

15.1 Ideal Statements

In order to accurately determine if a learner is following recommendations, there are a few requirements of the data produced by a LRP and the recommender itself. They are as follows:

- Every time the recommender makes a recommendation, a statement should be produced which uses the verb $https://w3id.org/xapi/dod{-}isd/verbs/recommended$[14] and has the recommended piece of content as the object.
  - the content should be uniquely and consistently identified across all statements.
- When a learner launches recommended content, the resulting launched statement should use the verb $http://adlnet.gov/expapi/verbs/launched$[15] and contain a refrence to the recommened content statement within $\$.context.statement$
  - Launching of content should use the above IRI regardless of why the content was launched
  - If it not possible to refrence the exact recommended content statement, the launch statement should have some indication that it was the result of a recommendation.[16]

[14] See footnote 4

[15] See footnote 4

[16] It is possible to determine if recommendations are followed (with some level of error) without this explicit linking of launched to recommended but this severly complicates the algorithm. In this case, in order to optmize for accuracy, the algorithm would need to consider the actor and their general activity within a session, the object of both launched and recommended statements generated within the session, the time lapse between recommendations and launches with a predefined lapse value which determines if a launch was close enough in time to a recommendation to be considered a result of the recommendation. An additonal constraint on the above case is the recommendation statements should contain a reference to to the person recieving the recommendation, otherwise determining the 1:1 relationships between recommendations and launches requires aditional complexity and will still not be 100% accurate due to the reliance on the time lapse value.

15.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource via curl.[17][18][19]

```
R = "verb=https://w3id.org/xapi/dod-isd/verbs/recommended"
L = "verb=http://adlnet.gov/expapi/verbs/launched"

Since = "since=2018-07-20T12:08:47Z"
Until = "until=2018-07-21T12:08:47Z"
Base = "https://example.endpoint/statements?"

endpoint1 = Base + R + "&" + Since + "&" + Until
endpoint2 = Base + L + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

SR = curl -X GET -H "Authorization: Auth"
          -H "Content-Type: application/json"
          -H "X-Experience-API-Version: 1.0.3"
          endpoint1
SL = curl -X GET -H "Authorization: Auth"
          -H "Content-Type: application/json"
          -H "X-Experience-API-Version: 1.0.3"
          endpoint2

S = SR + SL
```

15.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the $

- $.*verb.id*
- $.*context.statement*

15.4 2018 Pilot TLA Statement Problems

At the time of writing this document, launched statements do not include a statement reference or any indication of a connection between recommendations and launches. The authors of this document do not have access to the LRS

[17] footnote 1 applies to both S1 and S2.
[18] See footnote 2.
[19] See footnote 3.

containing the recommended statements and thus can not draw any conclusions about any issues which may be present within those statements or any aspects of those statements which may correlate them to launch statements. The following algorithm is going to assume that the input set of statements follow the guidlines outlined in section 5.1 as the additional algorithmic considerations brought on by non ideal statements, as specified within footnote 16, result in an algoirthm which is not optimal for near real time visualizations.

15.5 Summary

1. Query an LRS via a GET request to the statements endpoint using the paramters verb, since and until to gather all statements with the verb $http://adlnet.gov/expapi/verbs/launched$.

2. Query an LRS via a GET request to the statements endpoint using the paramters verb, since and until to gather all statements with the verb $https://w3id.org/xapi/dod-isd/verbs/recommended$.[20]

3. Group all collections of statements by a $TIMEUNIT$

4. seperate the collection of grouped launched statements into a collection of those which were the result of a recommendation and those which were not.

5. Take the count of all groups of statements
   - Recommended statements per $TIMEUNIT$
   - Launches due to recommendations per $TIMEUNIT$
   - Launches not due to recommendations per $TIMEUNIT$

6. Calculate summary statistics for the overall time range and per $TIMEUNIT$
   - Divide launches due to recommendations by the total number of launches to determine the percentage of launches due to recommendations
   - Divide launches due to recommendations by the total number of recommendations to determine the percentage of recommendations which are followed.

[20] If since and until are specified, they should be the same in both requests.

15.6 Formal Specification

15.6.1 System State

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FollowedRecommendations}} \\
\hline
Statements \\
CountPerGroup \\
S_{recommended}, S_{launched} : \mathbb{F}_1 \\
ordered_L, ordered_R, grouped_{launched}, grouped_{recommended}, \\
onlyRecommended, cPerGroup_{launched}, cPerGroup_{recommended}, \\
cPerGroup_{followed}, combined : \mathrm{seq} \\
t_{start}, N_{launched}, N_{recommended}, N_{followed}, P_{followed}, P_{dueto} : \mathbb{N} \\
tr_{start}, tr_{end} : \mathbb{F} \\
unit? : TIMEUNIT \\
\hline
S_{recommended} = statements \\
S_{launched} = statements \\
combined = \langle (tr_{start}, tr_{end}, N_{launched}, N_{recommended}, N_{followed}, P_{followed}, P_{dueto}) \rangle \\
count(grouped_{launched}) = count(grouped_{recommended}) \\
count(onlyRecommended) = count(grouped_{launched}) \Rightarrow \\
count(onlyRecommended) = count(grouped_{recommended}) \\
count(cPerGroup_{launched}) = count(cPerGroup_{followed}) = count(cPerGroup_{recommended}) \\
\hline
\end{array}
$$

- $S_{recommended}, S_{launched}$ are both non-empty, finite sets.
  - $S_{recommended}$ and $S_{launched}$ contain the results of querying an LRS for recommended and launched statements respectively.
- $ordered_L, ordered_R, grouped_{launched}, grouped_{recommended}, onlyRecommended,$ $cPerGroup_{launched}, cPerGroup_{recommended}, cPerGroup_{followed}$ and $combined$ are all finite sequences.
  - $ordered_L$ and $ordered_R$ are the sequences of statements within $S_{launched}$ and $S_{recommended}$ respectively and sorted by timestamp.
  - $grouped_{launched}$ is the result of grouping the statements within $ordered_L$ by $unit?$.
  - $grouped_{recommended}$ is the result of grouping the statements within $ordered_R$ by $unit?$.
  - $onlyRecommended$ is the result of filtering the statements within the sequence $grouped_{launched}$ to only include statements where $statement.context.statement$ is present
  - $cPerGroup_{launched}, cPerGroup_{recommended}, cPerGroup_{followed}$ are all sequences of numbers which represent the count within each subsequence of $grouped_{launched}$, $grouped_{recommended}$ and $onlyRecommended$ respectively.

- $combined$ is a sequence of ordered pairs where each pair consists of tr_{start}, tr_{end}, $N_{launched}$, $N_{recommended}$, $N_{followed}$, $P_{followed}$ and P_{dueto}

- $t_{start}, N_{launched}, N_{recommended}, N_{followed}, P_{followed}, P_{dueto}$ are all natural numbers
- tr_{start}, tr_{end} are both timestamps which represent the the start and end of the time range for each a group of statements.
- $unit?$ is an input representing a time interval, ie day vs month vs hour.
- all sequences are the same length so that each subsequence represents the same time grouping. In other words, indexes are comparable across sequences.

15.6.2 Initial System State

$$
\begin{array}{|l}
\hline
\textit{InitFollowedRecommendations} \\
\textit{FollowedRecommendations} \\
\hline
S_{recommended} \neq \emptyset \\
S_{launched} \neq \emptyset \\
unit? = \{day\} \\
ordered_L = \langle\rangle \\
ordered_R = \langle\rangle \\
grouped_{launched} = \langle\rangle \\
grouped_{recommended} = \langle\rangle \\
onlyRecommended = \langle\rangle \\
cPerGroup_{launched} = \langle\rangle \\
cPerGropu_{recommended} = \langle\rangle \\
cPerGroup_{followed} = \langle\rangle \\
combined = \langle\rangle \\
t_{start} = 0 \\
N_{launched} = 0 \\
N_{recommended} = 0 \\
N_{followed} = 0 \\
P_{followed} = 0 \\
P_{dueto} = 0 \\
\hline
\end{array}
$$

- $S_{recommended}$ and $S_{launched}$ are initially not empty sets
- all sequences are initially empty
- all numbers are initially zero
- the default $TIMEUNIT$ is set to day

15.6.3 Group by Timestamp

$SortByTimestamp$

$Statement$
$IsoToUnix$
$orderByTimestamp : \mathbb{F}_1 \rightarrow seq_1$
$o? : \mathbb{F}_1$
$o! : seq_1 statement$

$o? = \{o : statement\}$
$o! = orderByTimestamp(o?)$
$o! = \langle o_i..o_j \rangle \bullet \forall o_n : o_i..o_j \bullet o_n : STATEMENT \wedge i \leq n \leq j \bullet$
$\quad convert(o_i.timestamp) \leq convert(o_n.timestamp) \leq convert(o_j.timestamp)$

- The schema $SortByTimestamp$ introduces the function $orderByTimestamp$ which takes in a non-empty, finite set and returns a non-empty, finite sequence.
- $orderByTimestamp$ is a sequence of statements ordered from earliest to latest.

$WithinRange$

$withinRange : (\mathbb{N}, \mathbb{N}, \mathbb{N}, TIMEUNIT) \rightarrow \mathbb{F}_1 \#1$
$in?, start?, state? : \mathbb{N}$
$unit? : TIMEUNIT$
$out! : \{TRUE\} \vee \{FALSE\}$

$unit? = \{second\} \Rightarrow 1 \vee \{minute\} \Rightarrow 60 \vee \{hour\} \Rightarrow 3600 \vee$
$\quad \{day\} \Rightarrow 86400 \vee \{week\} \Rightarrow 604800 \vee$
$\quad \{month\} \Rightarrow 2629743 \vee \{year\} \Rightarrow 31556926$
$out! = withinRange(in?, start?, state?, unit?)$
$withinRange(in?, start?, state?, unit?) =$
$\quad \textbf{if } in? \leq start? + ((state? + 1) * unit?)$
$\quad\quad \textbf{then } out! = \{TRUE\}$
$\quad\quad \textbf{else } out! = \{FALSE\}$

- The schema $WithinRange$ introduces the function $withinRange$ which takes in three numbers and a $TIMEUNIT$ and returns either $\{TRUE\}$ or $\{FALSE\}$
- $withinRange$ checks to see if $in?$ is less than or equal to a start time $start?$ plus the result of multiplying the numeric conversion for $unit?$ by the $state?$.
- $state?$ represents the current group, ie. day 1 vs day 2 vs day 3. The +1 is to account for array indexes starting at 0.

$$
\begin{array}{|l}
\hline
\textit{GroupByTimeUnit} \\
\textit{Statement} \\
\textit{IsoToUnix} \\
\textit{WithinRange} \\
groupByTimeUnit : (\mathrm{seq}_1, \mathbb{N}, TIMEUNIT) \rightarrow \mathrm{seq}_1 \\
g?\,, g! : \mathrm{seq}_1 \\
t_{start}? : \mathbb{N} \\
\hline
g? = \langle g?_i \,..g?_j \,\rangle \bullet \forall g?_n : g?_i \,..g?_j \bullet i \leq n \leq j \bullet g?_n = statement \wedge \\
\quad convert(g?_i\,.timestamp) \leq convert(g?_n\,.timestamp) \leq convert(g?_j\,.timestamp) \\
g! = groupByTimeUnit(g?\,, t_{start}?\, state?\,, unit?\,) \\
g! = \langle g : seq \mid \forall g?_n : g?_i \,..g?_j \bullet \exists_1 \langle g_r \rangle : \langle g_q \rangle .. \langle g_s \rangle \bullet q \leq r \leq s \wedge r = state? \bullet \\
\qquad\qquad \textbf{if } withinRange(convert(g?_n\,.timestamp), t_{start}, r, unit?\,) = \{TRUE\} \\
\qquad\qquad\quad \textbf{then } g? \upharpoonright g?_n \wedge g?_n \mathsf{\ in\ } \langle g_r \rangle \Rightarrow \langle g_r \rangle = \langle g_{ri} .. g_{rn} .. g_{rj} \rangle \bullet ri \leq rn \leq rj \bullet g_{rn} = g?_n \\
\qquad\qquad\quad \textbf{else if } \forall g_n? : g_i? \,..g_j? \bullet withinRange(g?_n\,, t_{start}, r, unit?\,) = \{FALSE\} \\
\qquad\qquad\qquad \textbf{then } groupByTimeUnit((g? \upharpoonright \langle g_r \rangle), t_{start}?\,(r+1), unit?\,) \bullet \langle g_r \rangle = \langle\rangle \vee \neq \langle\rangle\rangle \\
\hline
\end{array}
$$

- The schema $GroupByTimeUnit$ intorudces the function $groupByTimeUnit$ which takes as arguments a non-empty, finite sequence, a natural number and a $TIMEUNIT$ and outputs a non-empty, finite sequence of sequences.

- For every statement within the input sequence, $groupByTimeUnit$ checks to see if the timestamp of that statement is within the range of t_{start} and $unit?$. If it is, that statement is removed from the input sequence $g?$ and added to the current subsequence $\langle g_r \rangle$. If none of the remaining statements within the input sequence are within the range of t_{start} and $unit?$, then the variable $state?$ is incremented, the current subsequence $\langle g_r \rangle$ is either a collection of matched statements or is an empty sequence and the search for remaining subsequences $\langle g_{r+state?} \rangle$ continues.

- because the input sequence $g?$ is orderd chronologically, this implies that once a statement does not fit into a range, the rest of the statements remaining in the input sequence will not fit into that range and $state?$ must be incremented to generate a new subsequence $\langle g_{r+state?} \rangle$ so that the remaining statements can be grouped.

15.6.4 Processes Results

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ OrderStatements\ }}\\
\Delta FollowedRecommendations \\
SortByTimestamp \\
\hline
ordered'_L = orderByTimestamp(S_{launched}) \\
ordered'_R = orderByTimestamp(S_{recommended}) \\
t'_{start} = convert((head\, ordered'_L).timestamp) \\
\hline
\end{array}$$

- The schema *OrderStatements* updates the system state defined by the schema *FollowedRecommendations*.
- $ordered'_L$ is the result of ordering the statements contained within the set $S_{launched}$ chronologically.
- $ordered'_R$ is the result of ordering the statements contained within the set $S_{recommended}$ chronologically.
- t'_{start} is the timestamp from the first statement within $ordered'_L$ converted to unix time.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ GroupByTime\ }}\\
\Delta FollowedRecommendations \\
GroupByTimeUnit \\
\hline
grouped'_{launched} = groupByTimeUnit(ordered'_L, t'_{start}, 0, unit?) \\
grouped'_{recommended} = groupByTimeUnit(ordered'_R, t'_{start}, 0, unit?) \\
\hline
\end{array}$$

- The schema *GroupByTime* updates the state defined by the schema *FollowedRecommendations*.
- $grouped'_{launched}$ is the result of passing $ordered'_L$, t'_{start}, 0 and $unit?$ to the function *groupByTimeUnit*.
- $grouped'_{recommended}$ is the result of passing $ordered'_R$, t'_{start}, 0 and $unit?$ to the function *groupByTimeUnit*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ OnlyRecommendedLaunches\ }}\\
\Delta FollowedRecommendations \\
\hline
onlyRecommended' = \langle o : seq \mid \mathbf{let}\ grouped'_{launched} == gl \Rightarrow \\
\qquad \langle\langle gl_i\rangle..\langle gl_j\rangle\rangle \Rightarrow \langle\langle gl_{ii}..gl_{ij}\rangle..\langle gl_{ji}..gl_{jj}\rangle\rangle \bullet \\
\qquad \forall\langle gl_n\rangle : \langle gl_i\rangle..\langle gl_j\rangle \bullet \exists_1\langle o_n\rangle : \langle o_i\rangle..\langle o_j\rangle \Rightarrow \langle\langle o_{ii}..o_{ij}\rangle..\langle o_{ji}..o_{jj}\rangle\rangle \bullet \\
\qquad ((\forall o_{in} : o_{ii}..o_{ij} \bullet o_{in}.context.statement \neq \emptyset \wedge o_{in}\ \mathsf{in}\ gl_i)\wedge \\
\qquad (\forall o_{jn} : o_{ji}..o_{jj} \bullet o_{jn}.context.statement \neq \emptyset \wedge o_{jn}\ \mathsf{in}\ gl_j))\vee \\
\qquad \langle o_n\rangle = \langle\rangle \\
\hline
\end{array}$$

- The schema *OnlyRecommendedLaunches* updates the state defined by the schema $FollowedRecommendations$.
- $onlyRecommended'$ is the sequence of objects o where o is a sequence consisting of statements (or no statements) from the corresponding sequences within $grouped'_{launched}$ where $statement.context.statement$ exists.
- $onlyRecommended'$ maintains the same number and ordering of time groups (subsequences) as $grouped'_{launched}$ and $grouped'_{recommended}$.

$$
\begin{array}{|l}
\hline
GetCounts \\
\Delta FollowedRecommendations \\
CountPerGroup \\
\hline
cPerGroup'_{launched} = \langle c : \mathbb{N} \mid \textbf{let } grouped'_{launched} == gl \Rightarrow \langle \langle gl_i \rangle .. \langle gl_j \rangle \rangle \bullet \\
\qquad \forall \langle gl_n \rangle : \langle gl_i \rangle .. \langle gl_j \rangle \bullet \exists_1 c_n : \mathbb{N} \bullet \\
\qquad \textbf{if } gl_n = \langle \rangle \\
\qquad\qquad \textbf{then } c_n = 0 \\
\qquad\qquad \textbf{else } c_n = count(\langle gl_n \rangle) \rangle \\
cPerGroup'_{recommended} = \langle c : \mathbb{N} \mid \textbf{let } grouped'_{recommended} == gr \Rightarrow \langle \langle gr_i \rangle .. \langle gr_j \rangle \rangle \bullet \\
\qquad \forall \langle gr_n \rangle : \langle gr_i \rangle .. \langle gr_j \rangle \bullet \exists_1 c_n : \mathbb{N} \bullet \\
\qquad \textbf{if } gr_n = \langle \rangle \\
\qquad\qquad \textbf{then } c_n = 0 \\
\qquad\qquad \textbf{else } c_n = count(\langle gr_n \rangle) \rangle \\
cPerGroup'_{followed} = \langle c : \mathbb{N} \mid \textbf{let } onlyRecommended' == or \Rightarrow \langle \langle or_i \rangle .. \langle or_j \rangle \rangle \bullet \\
\qquad \forall \langle or_n \rangle : \langle or_i \rangle .. \langle or_j \rangle \bullet \exists_1 c_n : \mathbb{N} \bullet \\
\qquad \textbf{if } or_n = \langle \rangle \\
\qquad\qquad \textbf{then } c_n = 0 \\
\qquad\qquad \textbf{else } c_n = count(\langle or_n \rangle) \rangle \\
\hline
\end{array}
$$

- The schema $GetCounts$ updates the state defined by the schema $FollowedRecommednations$.
- $cPerGroup'_{launched}$ is a sequence of numbers c where each c is either 0 or the result of passing the current subsequence of $grouped'_{launched}$ (gl_n) to the function *count*.
- $cPerGroup'_{recommended}$ is a sequence of numbers c where each c is either 0 or the result of passing the current subsequence of $grouped'_{recommended}$ (gr_n) to the function *count*.
- $cPerGroup'_{followed}$ is a sequence of numbers c where each c is either 0 or the result of passing the current subsequence of $onlyRecommended'$ (or_n) to the function *count*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CombineSequences \qquad\qquad\qquad\qquad\qquad\qquad}}\\
\Delta FollowedRecommendations \\
\hline
combined' = \langle c : (tr'_{start}, tr'_{end}, N'_{launched}, N'_{recommended}, N'_{followed}, P'_{followed}, P'_{dueto}) \mid \\
\qquad\qquad\qquad \textbf{let } grouped'_{launched} == gl \Rightarrow \langle \langle gl_i \rangle .. \langle gl_n \rangle .. \langle gl_j \rangle \rangle \\
\qquad\qquad\qquad\quad cPerGroup'_{launched} == cl \Rightarrow \langle cl_i .. cl_n .. cl_j \rangle \\
\qquad\qquad\qquad\quad cPerGroup'_{recommended} == cr \Rightarrow \langle cr_i .. cr_n .. cr_j \rangle \\
\qquad\qquad\qquad\quad cPerGroup'_{followed} == cf \Rightarrow \langle cf_i .. cf_n .. cf_j \rangle \\
\qquad\qquad\qquad \bullet \forall \langle gl_n \rangle : \langle gl_i \rangle .. \langle gl_j \rangle \bullet i \leq n \leq j \bullet \\
\qquad\qquad\qquad \exists_1 c_n : (tr_{startn}, tr_{endn}, N_{launchedn}, N_{recommendedn}, N_{followedn}, P_{followedn}, P_{dueton}) \bullet \\
\qquad\qquad\qquad tr_{startn} = (head\, gl_n).timestamp \\
\qquad\qquad\qquad tr_{endn} = (last\, gl_n).timestamp \\
\qquad\qquad\qquad N_{launchedn} = cl_n \\
\qquad\qquad\qquad N_{recommendedn} = cr_n \\
\qquad\qquad\qquad N_{followedn} = cf_n \\
\qquad\qquad\qquad P_{followedn} = cf_n \div cr_n \\
\qquad\qquad\qquad P_{dueton} = cf_n \div cl_n \rangle \\
\hline
\end{array}
$$

- The schema *CombineSequences* changes the state defined by the schema *FollowedRecommendations*.
- $combined'$ is a sequence of objects c where each c is an ordered pair of $tr'_{start}, tr'_{end}, N'_{launched}, N'_{recommended}, N'_{followed}, P'_{followed}, P'_{dueto}$.
- for each c_n:
  - $tr'_{start} \leadsto tr_{startn}$ which is equal to the timestamp for the first statement within gl_n
  - $tr'_{end} \leadsto tr_{endn}$ which is equal to the timestamp for the last statement within gl_n.
  - $N'_{launched} \leadsto N_{launchedn}$ which is equal to the current count of launched statements within the nth time grouping aka cl_n.
  - $N'_{recommended} \leadsto N_{recommendedn}$ which is equal to the current count of recommended statements within the nth time grouping aka cr_n.
  - $N'_{followed} \leadsto N_{followedn}$ which is equal to the current count of recommended statements within the nth time grouping aka cf_n.
  - $P'_{followed} \leadsto P_{followedn}$ which is equal to the result of dividing cf_n by cr_n.
  - $P'_{dueto} \leadsto P_{dueton}$ which is equal to the result of dividing cf_n by cl_n.

15.6.5 Sequence of Operations

$ProcessFollowedRecommendations \mathrel{\widehat=}$
$OrderStatements \fatsemi GroupByTime \fatsemi OnlyRecommendedLaunches \fatsemi$
$GetCounts \fatsemi CombineSequences$

- The schema $ProcessFollowedRecommendations$ defines the order of operations for the steps within the $FollowedRecommendations$ algorithm.

15.6.6 Return

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ReturnFollowedRecommendations \quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Xi FollowedRecommendations \\
ProcessFollowedRecommendations \\
combined! : \mathrm{seq} \\
\hline
combined! = combined' \\
\hline
\end{array}
$$

- The schema $ReturnFollowedRecommendations$ describes the return value of the system defined by the schema $FollowedRecommendations$
- The return value $combined!$ is the variable $combined'$ defined wtihin the schema $CombineSequences$

15.7 Pseudocode

Algorithm 4: Followed Recommendations

Input: $S_{recommended}$, $S_{launched}$ $timeUnit$
Result: $combined'$
$ordered'_L \leftarrow orderByTimestamp(S_{launched})$;
$ordered'_R \leftarrow orderByTimestamp(S_{recommended})$;
$t'_{start} \leftarrow convert((head\, ordered'_L).timestamp)$;
$grouped'_{launched} \leftarrow groupByTimeUnit(ordered'_L, t'_{start}, 0, timeUnit)$;
$grouped'_{recommended} \leftarrow$
  $groupByTimeUnit(ordered'_R, t'_{start}, 0, timeUnit)$;
$grouped_{followed} \leftarrow []$;
foreach G in $grouped'_{launched}$ **do**
  $curGrouping \leftarrow []$;
  foreach G_n in G **do**
    if $G_n.context.statement \neq nil$ **then**
      do
      $curGrouping' \leftarrow curGrouping \frown G_n$;
      recur curGrouping'
    else
      recur curGrouping'
    end
  end
  $grouped'_{followed} \leftarrow grouped_{followed} \frown curGrouping'$;
  recur $grouped'_{followed}$
end
$C_{launched} \leftarrow$ **map count() grouped**$'_{\textbf{launched}}$;
$C_{recommended} \leftarrow$ **map count() grouped**$'_{\textbf{recommended}}$;
$C_{followed} \leftarrow$ **map count() grouped**$'_{\textbf{followed}}$;
$combined \leftarrow []$;
for $i \leftarrow 0$ **to** $count(c_{launched})$ **by** 1 **do**
  $tr_{starti} \leftarrow (first(nth(grouped'_{launched}, i))).timestamp$;
  $tr_{endi} \leftarrow (last(nth(grouped'_{launched}, i))).timestamp$;
  $N_{Li} \leftarrow nth(C_{launched}, i)$;
  $N_{Ri} \leftarrow nth(C_{recommended}, i)$;
  $N_{Fi} \leftarrow nth(C_{followed}, i)$;
  $P_{Fi} \leftarrow N_{fi} \div N_{Ri}$;
  $P_{duetoi} \leftarrow N_{fi} \div N_{Li}$;
  $subVec_i \leftarrow [tr_{starti}, tr_{endi}, N_{Li}, N_{Ri}, N_{Fi}, P_{Fi}, P_{duetoi}]$;
  $combined' \leftarrow combined \frown subVec_i$
end
return $combined'$

- **map count()** $grouped'_{...}$ means apply the function **count()** to every sequence within the sequence $grouped_{...}$ and put all results into a single array.

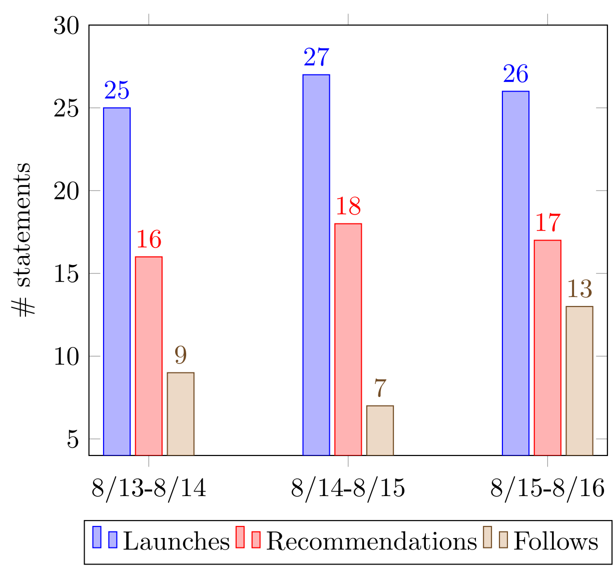
15.8 JSON Schema

```
{"type":"array",
   "items":{"type":"array",
      "items":[{"type":"string"}, {"type":"string"},
               {"type":"number"}, {"type":"number"},
               {"type":"number"}, {"type":"number"},
               {"type":"number"}]}}
```

15.9 Visualization Description

The **Followed Recommendations** visualization can be a bar chart where the domain is time ranges and the range is a number representing the total count of statements recorded. For each time range, there will be three groups: 1) the number of launched statements 2) the number of recommended statements 3) the number of launches which are due to recommendations. Above each grouping or on hover, summary statistics can be desplayed which describe the percentage of launches due to recommendations and the percentage of recommendations which were followed.

15.10 Visualization prototype



- The percentages described in section 5.9 are not displayed here.

15.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but they would require adding aditional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement paramters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

- populate a tooltip with the most popular launched, recommended and followed activity.
- populate a tooltip with the number of actors associated with the launches and follows.
- populate a tooltip with the actor who most often and the actor who lease often follows recommendations.

Appendex A: Visualization Exemplars

Appendex A includes a typology of data visualizations which may be supported within DAVE workbooks. These visualizations can either be one to one or one to many in regards to the algorithms defined within this document. Future iterations of this document will increasingly include these typologies within the domain-question template exemplars.

Line Charts

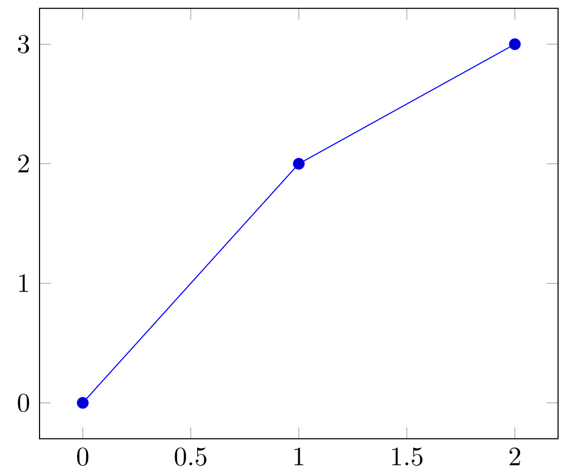


Figure 1: Line Chart

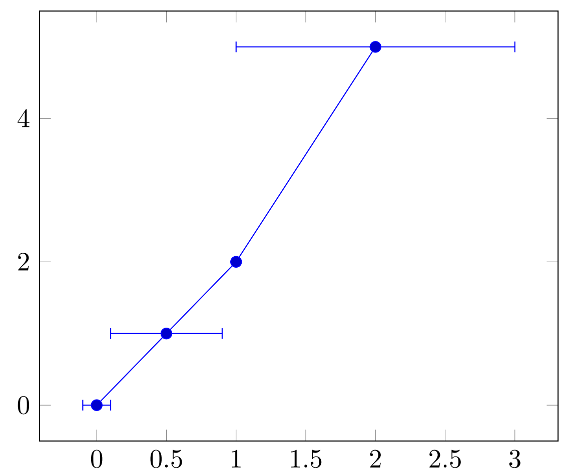


Figure 2: Line Chart with Error

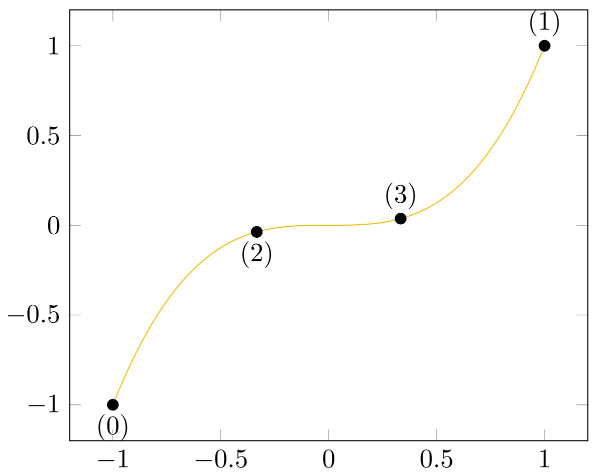


Figure 3: Spline Chart

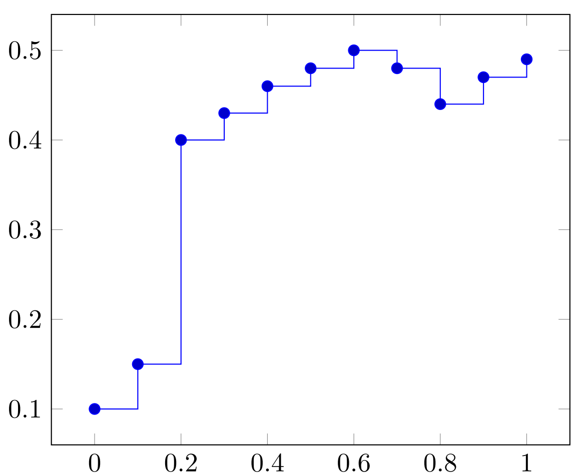


Figure 4: Quiver Chart

Grouping Charts

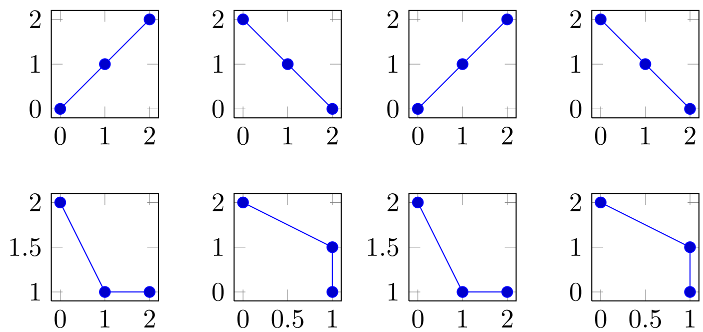


Figure 5: Grouped Line Charts

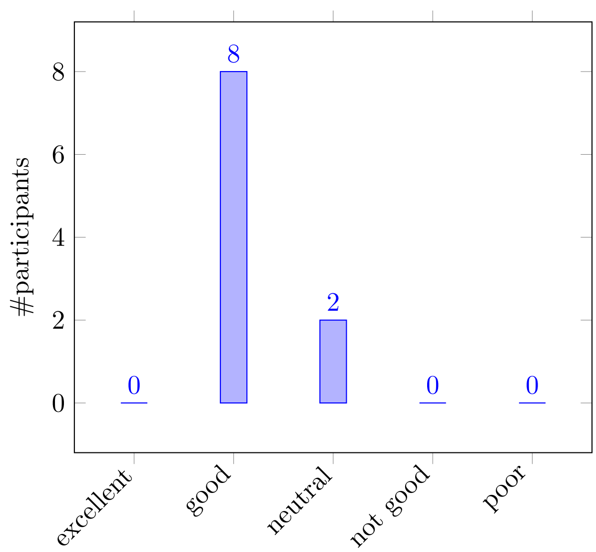


Figure 6: Histogram

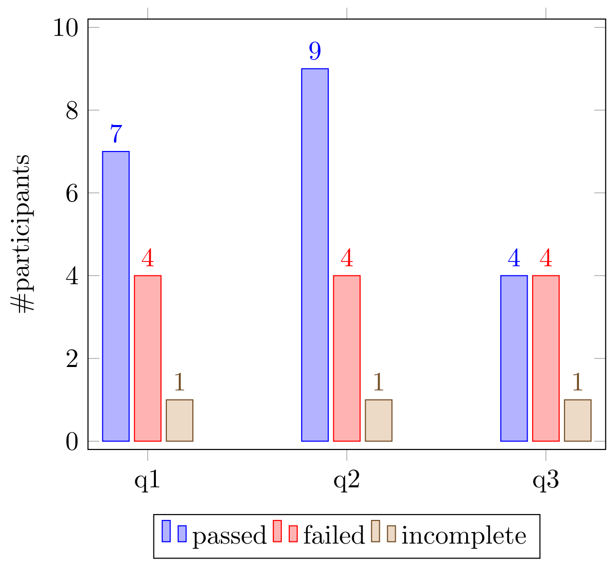


Figure 7: Bar Chart

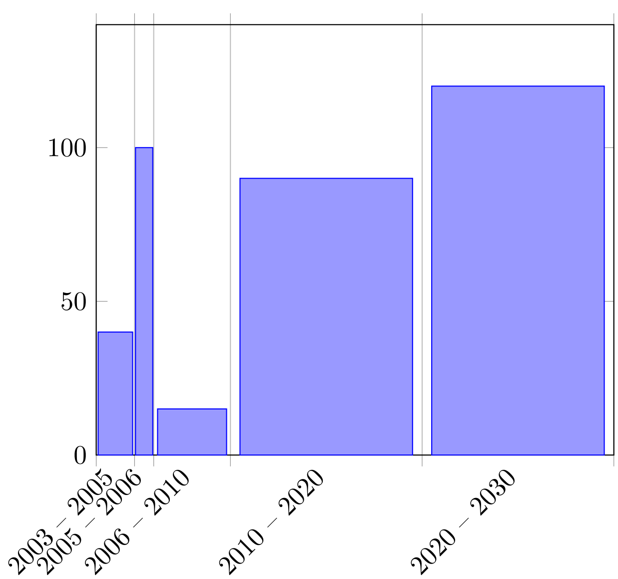


Figure 8: Bar Chart Grouped by Time Range

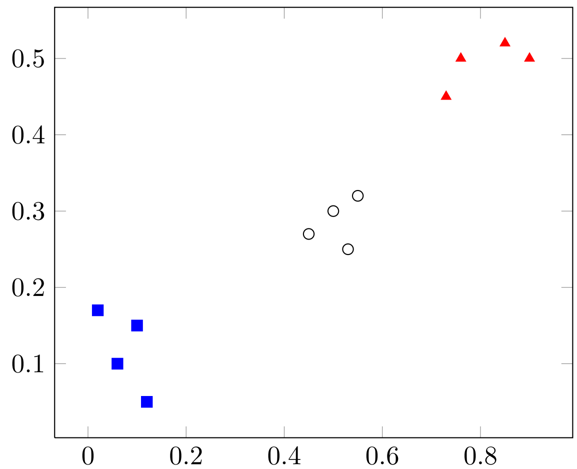


Figure 9: Scatter Plot

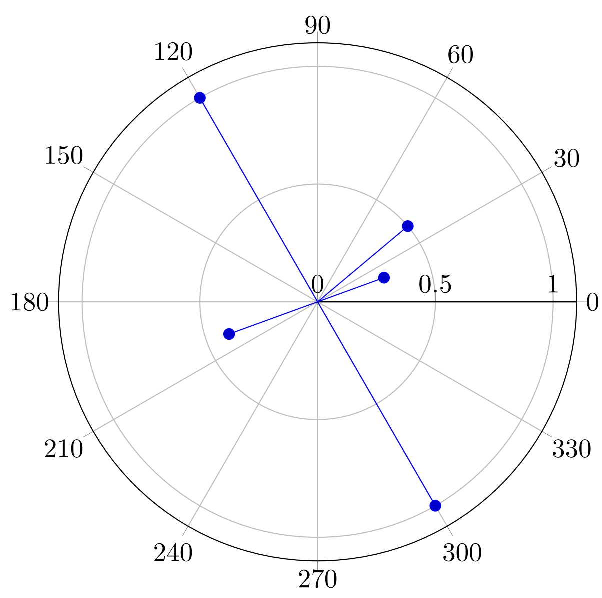


Figure 10: Polar Chart

Specialized Charts

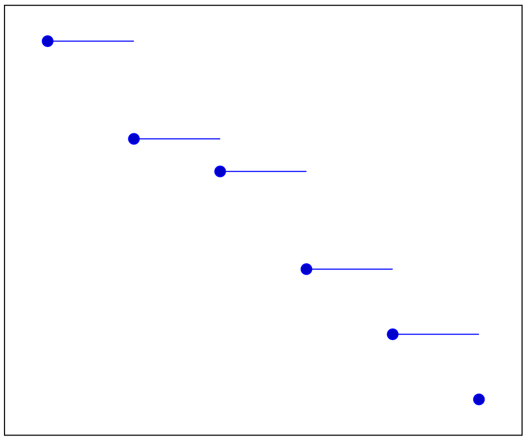


Figure 11: Gantt Chart

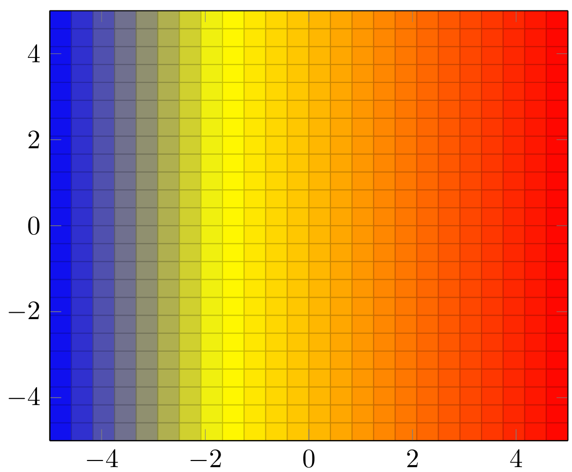


Figure 12: Heat Map

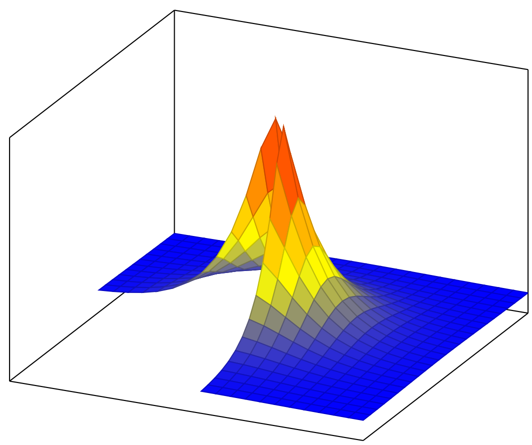


Figure 13: 3D Plot

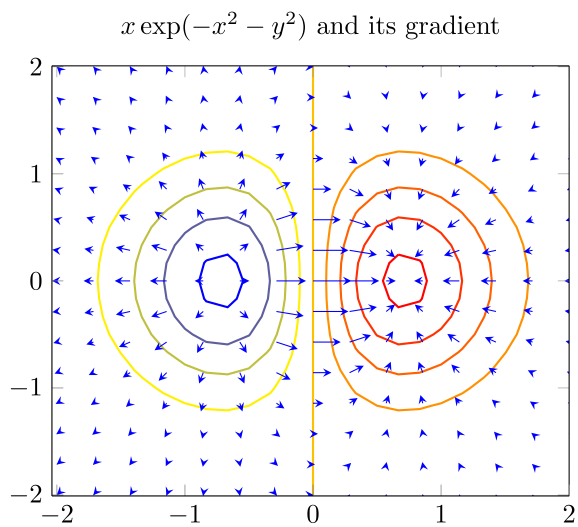


Figure 14: Gradient Plot