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

Yet Analytics

September 9, 2019

Introduction

This report introduces the initial learning analytics algorithms, **timeline of learner success**,**which assessment questions are the most difficult** and **rate of completions**. In doing so, it establishes a set of style guidelines for the reporting of algorithms and associated visualization templates. This document will be updated to include additional learning analytics questions presented elsewhere within this github repo in addition to other learning analytics algorithms which have yet to be defined. Updates may also address refinement of these algorithms 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 and referenced within the formal specifications of learning analytics algorithms
2. An algorithm definition which will consist 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

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

- 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

Agent

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

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

Member

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

- 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*

Group

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

- 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*

Actor

$$
\begin{array}{l}
Agent \\
Group \\
actor : AGENT \lor GROUP \\
\hline
actor = agent \lor group
\end{array}
$$

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

1.4 Verb Schema

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

- 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

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

- 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

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

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

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

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

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

- 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

Score

$$score : \mathbb{F}_1$$
$$scaled, min, max, raw : \mathbb{Z}$$

$$scaled = \{n : \mathbb{Z} \mid -1.0 \leq n \leq 1.0\}$$
$$min = n < max$$
$$max = n > min$$
$$raw = \{n : \mathbb{Z} \mid min \leq n \leq max\}$$
$$score = \mathbb{P}_1\{scaled, raw, min, max\}$$

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

Result

$$Score$$
$$Extensions$$
$$success, completion, response, duration : \mathbb{F}_1 \#1$$
$$result : \mathbb{F}_1$$

$$success = \{true\} \lor \{false\}$$
$$completion = \{true\} \lor \{false\}$$
$$result = \mathbb{P}_1\{score, success, completion, response, duration, extensions\}$$

- 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

Instructor

$$Agent$$
$$Group$$
$$instructor : AGENT \lor GROUP$$

$$instructor = agent \lor group$$

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

Team

$$Group$$
$$team : GROUP$$

$$team = group$$

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

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

- 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

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

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

- 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

Attachments

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

- 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

Statement

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

- 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

Statements

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

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

2 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.

2.1 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

2.2 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.[1][2][3]

[1] 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

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

[3] 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
```

2.3 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*

2.4 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.

2.5 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*, #]

2.6 Formal Specification

2.6.1 Basic Types

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

$SUCCESS ::= \{true\}$

2.6.2 System State

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

- 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

2.6.3 Initial System State

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

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

2.6.4 Filter for Completion

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

- 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 $\$.\textit{verb.id}$ is of the type $COMPLETION$ otherwise $s!$ is an empty set

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

- 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 $completions'$

2.6.5 Filter for Success

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

- 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

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

- 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 $successes'$

2.6.6 Processes Results

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

- 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

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

- 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 arugments 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′*

2.6.7 Sequence of Operations

$FilterStatements \mathrel{\widehat=} FilterForCompletion \fatsemi FilterForSuccess$

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

$ProcessedStatements \mathrel{\widehat=} FilterStatements \fatsemi ProcessStatements$

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

2.6.8 Return

Return

$$\Xi TimelineLearnerSuccess$$
$$ProcessedStatements$$
$$S_{processed}! : \mathbb{F}$$

$$S_{processed}! = S_{processed}$$

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

2.7 Pseudocode

Algorithm 1: Timeline of Learner Success

Input: S_{all}
Result: $coll'$
$coll = []$;
while $S_{all} \neq \emptyset$ **do**
  foreach $s \in S_{all}$ **do**
    if $s.verb.id = COMPLETION$ **then**
      do
      $S'_{completion} \leftarrow s \cup S_{completion}$;
      $S'_{all} \leftarrow S_{all} \setminus s$;
      recur $S'_{completion}, S'_{all}$;
    else
      do
      $S'_{all} \leftarrow S_{all} \setminus s$;
      recur S'_{all};
    end
  end
end
while $S'_{completion} \neq \emptyset$ **do**
  foreach $sc \in S'_{completion}$ **do**
    if $sc.result.success = SUCCESS$ **then**
      do
      $S'_{success} \leftarrow sc \cup S_{success}$;
      $S'_{completion} \leftarrow S_{completion} \setminus sc$;
      recur $S'_{success}, S'_{completion}$;
    else
      do
      $S'_{completion} \leftarrow S_{completion} \setminus sc$;
      recur $S'_{completion}$;
    end
  end
end
foreach $ss \in S'_{success}$ **do**
  $raw? \leftarrow ss.result.score.raw$;
  $max? \leftarrow ss.result.score.max$;
  $min? \leftarrow ss.result.score.min$;
  $scaled \leftarrow scale(raw?, min?, max?)$;
  $subVec \leftarrow [ss.timestamp, scaled]$;
  $coll' \leftarrow coll \frown subVec$;
  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.

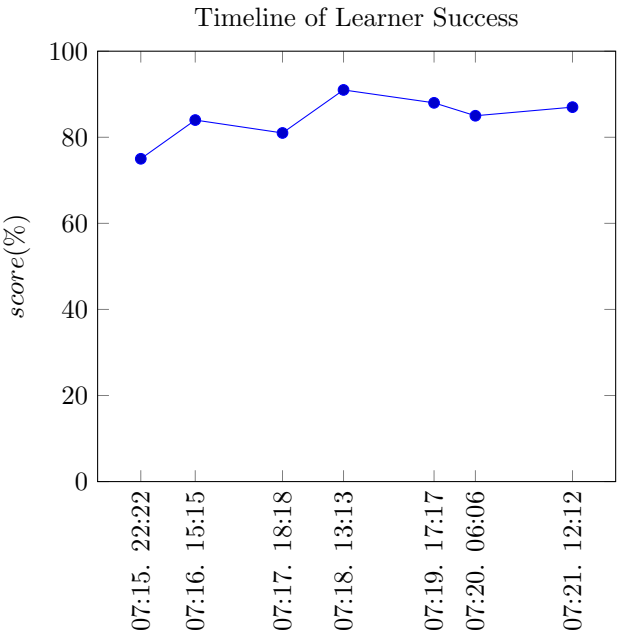
2.8 JSON Schema

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

2.9 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.

2.10 Visualization prototype



2.11 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 $\$.context.instructor$

3 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.

3.1 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[4] use the verb $http://adlnet.gov/expapi/verbs/answered$

3.2 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.[5][6][7]

```
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
```

3.3 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 $

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

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

[5] See footnote 1.

[6] See footnote 2.

[7] See footnote 3.

3.4 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.

3.5 Summary

1. Query an LRS via a GET request to the statements endpoint using the parameters verb, since and until
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 = [$\$.object.id$, #]

3.6 Formal Specification

3.6.1 Basic Types

$INCORRECT :== \{false\}$

3.6.2 System State

$MostDifficultAssessmentQuestions$

$Statements$
$S_{all} : \mathbb{F}_1$
$S_{incorrect}, S_{grouped}, S_{processed} : \mathbb{F}$

$S_{all} = statements$
$S_{incorrect} \subseteq S_{all}$
$S_{grouped} = \{groups : \text{seq}_1\ statement\}$
$S_{processed} = \{pair : (id, \mathbb{N})\}$

- 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

3.6.3 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

3.6.4 Filter for Incorrect

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

- 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

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

- 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 $incorrects'$

3.6.5 Processes Results

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

- 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 $\$.object.id$

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

- 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 $\langle c?_i \mathinner{..} c?_j \rangle$, 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

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

- 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 $g_i \mathinner{..} g_j$ 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 $processed'$

3.6.6 Sequence of Operations

$ProcessedQuestions \mathrel{\widehat=} FilterForIncorrect \fatsemi AggregateQuestionStatements$

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

3.6.7 Return

$ReturnAggregate$

$\Xi MostDifficultAssessmentQuestions$
$ProcessedQuestions$
$S_{processed}! : \mathbb{F}$

$S_{processed}! = S_{processed}$

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

3.7 Pseudocode

Algorithm 2: 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 $S'_{incorrect}$

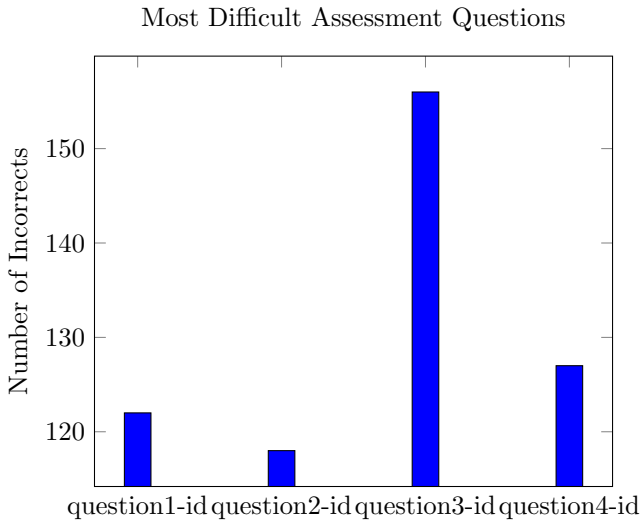
3.8 JSON Schema

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

3.9 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.

3.10 Visualization prototype



3.11 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*

4 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[8] of the various digital resources within the learning ecosystem.

4.1 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[9] 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*

[8] 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

[9] See footnote 4

- 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 $\$.result.duration$

4.2 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.[10][11][12]

```
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
```

4.3 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 $

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

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

4.4 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 $\$.result.score.max$.[13] 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.

4.5 Summary

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

4.6 Formal Specification

4.6.1 Basic Types

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

4.6.2 System State

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

[13] 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.

- 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 $TIMEUNIT$

4.6.3 Initial System State

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

- 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 $\{day\}$

4.6.4 Calculate Rate

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

- 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.

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

- 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

4.6.5 Processes Results

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

- 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 $S_{completions}$

4.6.6 Return

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

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

4.7 Pseudocode

Algorithm 3: Rate of Completions

Input: $S_{completed}$, $timeUnit$
Result: $ratePerObjTu'$
$context = \{\}$;
$ratePerObjTu = []$;
while $S_{completion} \neq \emptyset$ **do**
  foreach $s \in S_{completion}$ **do**
    $id \leftarrow s.object.id$;
    $ts \leftarrow convert(s.timestamp)$;
    if $id \notin context$ **then**
      do
      $times = [ts]$;
      $context' \leftarrow \{id : times\}$;
      $S'_{completion} \leftarrow S_{completion} \setminus s$;
      recur $context', S'_{completion}$;
    else
      do
      $times' \leftarrow context.id \frown ts$;
      $context' \leftarrow \{id : times'\}$;
      $S'_{completion} \leftarrow S_{completion} \setminus s$;
      recur $context', S'_{completion}$;
    end
  end
end
foreach $k \in context'$ **do**
  $allTs \leftarrow context'.k$;
  $totalDuration \leftarrow max(allTs) - min(allTs)$;
  $totalCount \leftarrow count(allTs)$;
  $rate \leftarrow totalCount \div (totalDuration \div timeUnit)$;
  $subVec = [k, rate, timeUnit]$;
  $ratePerObjTu' \leftarrow ratePerObjTu \frown subVec$;
  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*.

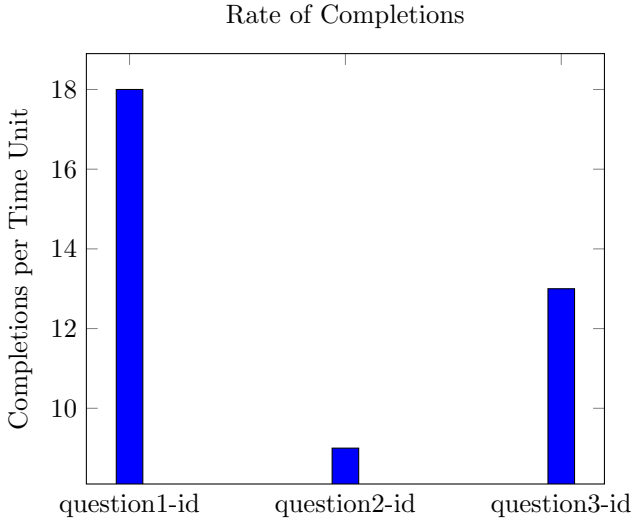
4.8 JSON Schema

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

4.9 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.

4.10 Visualization prototype



4.11 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.

5 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.

5.1 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.

[14] See footnote 4

  - 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]

5.2 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
```

[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.

[17] footnote 1 applies to both S1 and S2.

[18] See footnote 2.

[19] See footnote 3.

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

S = SR + SL
```

5.3 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*

5.4 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 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.

5.5 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.

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

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.

5.6 Formal Specification

5.6.1 System State

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

- $S_{recommended}, S_{launched}$ 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.

5.6.2 Initial System State

InitFollowedRecommendations

FollowedRecommendations

$S_{recommended} \neq \emptyset$
$S_{launched} \neq \emptyset$
$unit? = \{day\}$
$ordered_L = \langle\rangle$
$ordered_R = \langle\rangle$
$grouped_{launched} = \langle\rangle$
$grouped_{recommended} = \langle\rangle$
$onlyRecommended = \langle\rangle$
$cPerGroup_{launched} = \langle\rangle$
$cPerGropu_{recommended} = \langle\rangle$
$cPerGroup_{followed} = \langle\rangle$
$combined = \langle\rangle$
$t_{start} = 0$
$N_{launched} = 0$
$N_{recommended} = 0$
$N_{followed} = 0$
$P_{followed} = 0$
$P_{dueto} = 0$

- $S_{recommended}$ 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

5.6.3 Group by Timestamp

SortByTimestamp

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

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

- 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

$$
\begin{array}{l}
withinRange : (\mathbb{N}, \mathbb{N}, \mathbb{N}, TIMEUNIT) \rightarrow \mathbb{F}_1 \#1 \\
in? , start? , state? : \mathbb{N} \\
unit? : TIMEUNIT \\
out! : \{TRUE\} \lor \{FALSE\} \\
\hline
unit? = \{second\} \Rightarrow 1 \lor \{minute\} \Rightarrow 60 \lor \{hour\} \Rightarrow 3600 \lor \\
\quad \{day\} \Rightarrow 86400 \lor \{week\} \Rightarrow 604800 \lor \\
\quad \{month\} \Rightarrow 2629743 \lor \{year\} \Rightarrow 31556926 \\
out! = withinRange(in? , start? , state? , unit?) \\
withinRange(in? , start? , state? , unit?) = \\
\quad \textbf{if } in? \leq start? + ((state? + 1) * unit?) \\
\quad\quad \textbf{then } out! = \{TRUE\} \\
\quad\quad \textbf{else } out! = \{FALSE\}
\end{array}
$$

- 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.

GroupByTimeUnit

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

- 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.

5.6.4 Processes Results

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

- 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.

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

- 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 $groupByTimeUnit$.

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

- 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 $grouped'_{recommended}$.

GetCounts

$\Delta FollowedRecommendations$

$CountPerGroup$

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

- 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*.

CombineSequences

$\Delta FollowedRecommendations$

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

- 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:
  - $tr'_{start} \rightsquigarrow tr_{startn}$ which is equal to the timestamp for the first statement within gl_n
  - $tr'_{end} \rightsquigarrow tr_{endn}$ which is equal to the timestamp for the last statement within gl_n.
  - $N'_{launched} \rightsquigarrow N_{launchedn}$ which is equal to the current count of launched statements within the nth time grouping aka cl_n.
  - $N'_{recommended} \rightsquigarrow N_{recommendedn}$ which is equal to the current count of recommended statements within the nth time grouping aka cr_n.
  - $N'_{followed} \rightsquigarrow N_{followedn}$ which is equal to the current count of recommended statements within the nth time grouping aka cf_n.
  - $P'_{followed} \rightsquigarrow P_{followedn}$ which is equal to the result of dividing cf_n by cr_n.
  - $P'_{dueto} \rightsquigarrow P_{dueton}$ which is equal to the result of dividing cf_n by cl_n.

5.6.5 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.

5.6.6 Return

$$
\begin{array}{|l}
\hline
ReturnFollowedRecommendations \\
\Xi FollowedRecommendations \\
ProcessFollowedRecommendations \\
combined! : \text{seq} \\
\hline
combined! = combined' \\
\hline
\end{array}
$$

- 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 $CombineSequences$

5.7 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**() $\mathbf{grouped'_{launched}}$;
$C_{recommended} \leftarrow$ **map count**() $\mathbf{grouped'_{recommended}}$;
$C_{followed} \leftarrow$ **map count**() $\mathbf{grouped'_{followed}}$;
$combined \leftarrow []$;
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.

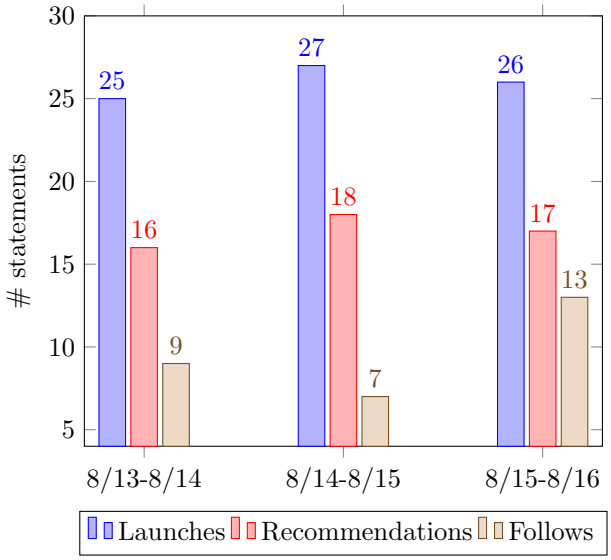
5.8 JSON Schema

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

5.9 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.

5.10 Visualization prototype



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

5.11 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.