Overview

Single source of record. Our vision is for Rally to be the single source of record for your software and systems project data. To see this vision forward, we are constantly striving to improve your ability to make better decisions by gleaming insight from your data. Up until now, it's been difficult to get time-series data out in a raw form without resorting to taking your own nightly data dumps or parsing the revision history of each individual work item.

First installment of Analytics 2.0. This API (code named Lookback API), is the first installment on Rally's Analytics 2.0 initiative. This A2.0 effort will include interactive charts and visualizations along with user interface support to easily configure them and place them wherever you want, both inside and outside of Rally. However, in the tradition of Agile, we are doing the smallest increment that can deliver value (and provide feedback) soonest,

Go back in time. With this API, you will be able to roll back the clock and see what any work item or collection of work items looked like at any moment in time. This is the data that you need to calculate your own time-series charts (burn, cumulative flow, defect trend, etc.). Have you ever wanted your burn chart to be based upon counts of work items? Have you ever wondered how much the burn-up projection would change if you filtered out all of the "nice-to-have" stories? Want your defect trend chart filtered by a custom field? All of these things are now possible.

Advanced temporal visualizations and queries. It also opens up a whole new world of advanced visualizations. Imagine showing an animation of your kanban board or showing the iteration status as it looked at 5pm on the last day of the iteration. We can't wait to see what you dream up. Furthermore, even if you are looking for a particular type of event but don't know when it occurred, this API can help. You can ask it for all of the instances where a story moved from In-Progress backward to Ranked or any time when a P1 defect was downgraded.

Total dogfooding. No longer will we maintain a private data warehouse that we use to generate charts in Rally but give you much more limited access to your own data. This API and the underlying data model are exactly what we are consuming in our own A2.0 development. Also, the charts that we develop will be available in source code form (like Apps are now) so you can tweak them in ways that we can't even imagine.

Release status

Currently: early-access preview, running in prodution for a sub-set of all subscriptions, connected to live Rally for changes... usually within a few seconds.

Anticipated phase of availability is indicated by coloring as follows:

Alpha - No coloring
Delayed in Preview - There were a few things that are not present 03/2012 (mostly due to Rally WSAPI bug(s)). We are fixing these as they come up.
Expected by General Availability - Yellow
Later (if ever) - Pink

TODOS FOR THIS DOCUMENT

- Fix all JSON so that the keys are in quotes.
- Specify what we do when the mime type doesn't match the extension. Should we allow requests against the endpoint without the .js or .csv or .json or .xml to work if the mime type specifies it.

Table of Contents

Data model

The data model for the repository that sits under this API has been carefully crafted for efficient analytics. It is particularly well suited to seeing how your data changes over time which is the focus of most reports (burn charts, defect trend, cumulative flow, etc.).

Snapshot schema

The data is stored in a snapshot schema which means that every time there is a change, an entirely new snapshot of the effected entity is saved with the new values (as well as the old ones). The older snapshot is not removed. It is only updated to adjust its _ValidTo timestamp.

Let's say you have this:

```
{
  _id: B2E,       // GUID just for analytics engine
  ObjectID: 777,  // objectID (OID) from Rally
  _UnformattedID: 2345,
  _Type: ["PersistableObject", "DomainObject",
          "WorkspaceDomainObject", "Artifact", "Defect"]
  Name: "Footer disappears when using new menu",
  State: "Submitted",
  _ValidFrom: "2011-01-01T12:34:56Z",
  _ValidTo: "9999-01-01T00:00:00Z",    // "current" snapshot
  ... // Other fields not shown
}
```

Then on January 2, 2011, at noon GMT, the analytics engine receives a notice that Rally object 777 had its "State" field changed from "Submitted" to "Open". The latest record for rally object 777 is read. Its _validTo is updated but nothing else is changed in that record. Rather, a new record is created showing the new value as well as the previous values for the field(s) that changed. So, the repository would now contain the updated original plus the new snapshot like so:

```
{
  _id: B2E,       // GUID just for analytics engine
  ObjectID: 777,  // objectID (OID) from Rally
  _UnformattedID: 2345,
  _Type: ["PersistableObject", "DomainObject",
          "WorkspaceDomainObject", "Artifact", "Defect"]
  Name: "Footer disappears when using new menu",
  State: "Submitted",
  _ValidFrom: "2011-01-01T12:34:56Z",
  _ValidTo: "2011-01-02T12:00:00Z", // updated
  ... // Other fields not shown
}

{
  _id: A37,       // a new analytics "document" so it gets a new _id
  ObjectID: 777,  // same old Rally OID
  _UnformattedID: 2345,
  _Type: ["PersistableObject", "DomainObject",
          "WorkspaceDomainObject", "Artifact", "Defect"]
  Name: "Footer disappears when using new menu",
  State: "Open",
  _ValidFrom: "2011-01-02:12:00:00Z",  // equals B2E's _ValidTo
  _ValidTo: "9999-01-01T00:00:00Z",
  _PreviousValues: {
    State: "Submitted"
  },
  ... // Other fields not shown
}
```

Things to note:

- Every time there is a change, an entirely new snapshot of the effected entity is saved with the new values (as well as the unchanged ones). Each snapshot is numbered sequentially and identified by the _SnapshotNumber field.
- The _PreviousValues field stores the values that were replaced when this particular snapshot was added.
- The entity type and fields are named according to how they are named in Rally's WSAPI.
- Fields that start with an underscore ("_") are fields that do not exist in Rally's WSAPI and are provided for the convenience of analytics queries.
- All timestamps are stored in GMT.
- The way _ValidFrom and _ValidTo are manipulated, you can rely upon the property that for a given Rally ObjectID, only one version of the object will be active for any moment in time.
- The examples above are only a subset of all the fields in the document.
- Big text fields and attachments are excluded, but Name is included along with most other fields, including custom fields.
- Null (<No Entry>) values are not stored except...
- There is a special case where a value is changed from null to a non-null value. In this case, the _PreviousValues field will explicitly say that it was null before the change. This null pattern is also followed for all fields that are set when the item is created.

Queries

Queries are submitted to the Rally analytics engine, either (1) via POST where the contents have a single well-formed JSON object; or (2) via GET with the same parameters.

A query follows this general format:

```
{
  find: {  // Required
    // Query clauses
  },
  fields: ["State", "PlanEstimate"], // Field list
  pagesize: 1000, // default (if omitted) DEFAULT_PAGESIZE
                  // changed to min([MAX_PAGESIZE, <your value>)
  start: 0, // Specifies at which object it should begin returning
  sort: {_id: 1},
  hydrate: {
    // Hydrate specification
  }
}
```

or

```
https://rally.../analytics/1.27/1234/artifact/snapshot/query.js?find=
{...}
```

where 1.27 is the version of the API and 1234 is the ObjectID of the Workspace.

Find (required)

The operators in the query syntax for the analytics APIs are a subset of that offered by MongoDB. We find this to be an elegant syntax that provides a lot of flexibility and low friction with our current implementation, which happens to use MongoDB. You can read about the MongoDB query operators here.

Supported Operators

We support the following operator usage:

- `{a: 10}` - docs where a is 10 or an array containing the value 10
- `{a: 10, b: "hello"}` - docs where a is 10 and b is "hello"
- `{a: {$gt: 10}}` - docs where a > 10, also $lt, $gte, and $lte
- `{a: {$ne: 10}}` - docs where a != 10
- `{a: {$in: [10, "hello"]}}` - docs where a is either 10 or "hello"
- `{a: {$exists: true}}` - docs containing an "a" field
- `{a: {$exists: false}}` - docs not containing an "a" field
- `{a: {$type: 2}}` - docs where a is a string (see bsonspec.org for more types)
- `{a: /foo.*bar/}` - docs where a matches the regular expression "foo.*bar"
- `{"a.b": 10}` - docs where a is an embedded document where b is 10
- `{$or: [{a: 1}, {b: 2}]}` - docs where a is 1 or b is 2
- `{$and: [{a: 1}, {b: 2}]}` - docs where a is 1 and b is 2
- `{a: {$all: [10, "hello"]}} - NOT SUPPORTED NOW`
- `{a: {$mod: [10, 1]}} - NOT SUPPORTED NOW`
- `{a: {$size: 3}} - NOT SUPPORTED NOW`
- `{a: {$elemMatch: {b: 1, c: 2}} NOT SUPPORTED NOW`

The following operator usage supported by MongoDB is currently supported only as an experiment at this time but may not be supported long-term by this API. Consider them undocumented features. Use them at your own risk. The main problem with $nin is performance. If we can figure out a way to allow its safe usage (throttling?), then we may "document" it. We left it in for now to enable experimentation that will help us discern how valuable it might be.

- `{a: {$not: {$type: 2}}} - $not is NOT SUPPORTED NOW`
- `{a: {$nin: [10, "hello"]}} - $nin is NOT SUPPORTED NOW`

- `{$where: "this.a == this.b"} - $where is NOT SUPPORTED NOW`

Operators for date fields

_ValidFrom and _ValidTo supports the following operators:

- equals (for example {_ValidFrom: "2011-01-01TZ"})
- $gt
- $gte
- $lt
- $lte
- $ne

__At only supports equals.

In exchange for limited operators on _ValidFrom, _ValidTo and __At, you get full support for ISO-8601 date string formats including week numbers, day of the year, etc.

For other date fields, the API will attempt to parse the canonical forms of ISO-8601 (yyyy-mm-ddTZ, yyyy-mm-ddThh:mm:ssZ, and yyyy-mm-ddThh:mm:ss.lllZ as well as the timeshift equivalents). The API will attempt to apply whatever operators you give it on date fields... whether they make sense or not.

FormattedID

Rally's standard WSAPI allows a query clause like

```
(FormattedID = 2345) // Note the missing "DE"
```

We don't actually have a FormattedID field in the analytics database. Rather, we keep a field named _UnformattedID. It is the FormattedID without the prefix. You can submit queries like

```
{_UnformattedID: 2345}
```

However, you can also send queries with a clause on FormattedID. They will simply be converted to two clauses; one on _UnformattedID and the other on _Type. So, it becomes

```
{_UnformattedID: 2345, _Type: "Defect"}
```

Because of complications associated with the conversion around FormattedID, not all combinations of operators are supported. There is support for: $and, $or, $ne, $in, and $exists with FormattedID, including all recursive combinations of $and and $or. If a complicated query is not behaving as expected try recomposing it with ObjectID instead of FormattedID. Operators we will likely never support on FormattedID include $lt, $lte, $gt, $gte, $all, $mod, $nin, $nor, $type... but they don't really make sense for this field.

String matching against drop-down fields

The allowed values for drop-down fields are specific to the Rally work item type. So the allowed values for the State field for Tasks are different than the State field for Defects. When you submit a query against a drop-down field, it will try to match the provided drop-down value(s) against any type that might have that value. In most cases, this is what you want. So:

```
{KanbanState: "Accepted"}
```

will match any work item that happens to have a KanbanState field set to "Accepted".

However, if you want to restrict your query to a particular work item type, you can do so by including an _Type clause in your query like:

```
{_Type: "Defect", Priority: "High"}
```

Inequality comparisons against drop down fields

You can also do inequality comparisons against drop-down fields, like this:

```
    {KanbanState: {$lt:"In Progress"}}
```

In the above example, the Lookback API will first find all types that have a KanbanState field and find the ordered list of values for that type. Let's say you have this:

HierarchicalRequirement.KanbanState $\in$ ['Ready', 'In Progress', 'Done']
Defect.KanbanState $\in$ ['Triage', 'Ready', 'In Progress', 'Done']

The Lookback API will expand the above query into the following:

```
    {KanbanState: {$in:[null, "Triage", "Ready"]}
```

Notice how "null" is an automatically-added lowest order value. This means that any work item where the the KanbanState field is "<No Entry>" will match the above clause.

Highly selective criteria

In order to limit the load that any one query will place on the service and to ensure that every query uses a fairly selective index, we are going to place restrictions on queries. For instance, we will allow you to get all versions of a single work item, but we may not allow you to get all versions of every work item in a project. On the other hand, we will allow you to get a single moment-in-time view of all work items in a single project but may restrict deep project hierarchies. Project hierarchy queries may be limited by how many projects they encompass and/or may require that you specify some other criteria like State, Priority, etc. The definition of "highly selective" will be a moving target during the alpha period. [REMOVE ALPHA REFERENCE LATER] Our current plan is to start out with no such restrictions and add the restrictions as use cases are better defined. Also, these restrictions may become more liberal when throttling is implemented.

Fields

If no fields parameter is provided, then only the minimum fields are returned in the response (see Response section below for the list of minimum fields). Note, this behavior is the opposite of MongoDB which will return the full documents by default.

If you want all the fields returned send in "true":

```
      fields: true  // Throttled for non-production use only?
```

Most commonly, you will provide a list of fields:

```
      fields: ["Name", "PlanEstimate"]
```

The above will return only Name and PlanEstimate.

We also support MongoDB's object format for field specifications. This gains us the use of the slice operator to return a particular element in an array value. The most likely usage for this is to get the last element in the _Type array.

```
      fields:{ObjectID: 1, _Type: {$slice: -1}}
```

For an explanation why we didn't use "fetch" like Rally's standard web services, see the FAQ.

Hydrate

By default, this API does not attempt to disambiguate drop-down field values from their native ObjectID integer form into strings. However, you can do this disambiguation on the client by calling our allowed values endpoint. For example, sending a GET to:

https://rally1.../webservice/1.26/allowedattributevalue/1234.js

would give you a response that would include the string for the allowed value whose OID is 1234.

Also, by default, the _UnformattedID field does not contain the prefix that's on the FormattedID.

You can specify some fields to be hydrated and others to not be hydrated using the same format at the "fields" parameter. For instance:

```
  hydrate: {
    State: 1,
    FormattedID: 1
  }
```

will disambiguate the State field into a string instead of an ObjectID. It will also put the prefix in front of the FormattedID so you'll get back "DE2345" instead of just "2345".

Start and pagesize

These behave as you would expect from Rally's standard WSAPI parameters of the same name which is also how SQL's SKIP and LIMIT behave. If you supply no pagesize, it will default to DEFAULT_PAGESIZE, which is a pre-determined, low value. If you supply a value higher than the service's current MAX_PAGESIZE value, the query will be executed as if you sent MAX_PAGESIZE.

Note: the settings for DEFAULT_PAGESIZE and MAX_PAGESIZE are subject to change over time.

Sort

You specify sort using an object whose key is the field you want it sorted by and whose value is either 1 for ascending or -1 for descending. This sorts by the _ValidFrom descending:

```
  sort: {_ValidFrom: -1}
```

Sort should use the same index as the query. We may reply with an error if the sort you have chosen is not supported by a valid index.

Response

The following is an example of a response to a query where no "fields" parameter is specified:

```
{
  _rallyAPIMajor: "1",
  _rallyAPIMinor: "26",
  Errors: [],
  Warnings: [],
  TotalResultCount: 54,
  StartIndex: 1,
  PageSize: 5,
  ETLDate: "2011-10-10T12:34:56.789Z",
  Results: [
    {
      _id: B2E,
      ObjectID: 777,
      _ValidFrom: "2011-01-01T12:34:56Z",
      _ValidTo: "9999-01-01T00:00:00Z",
      Project: 3456,
    },
    ... // Another 4 records
  ]
}
```

Notes:

- The fields shown above are the only fields that are returned when a request is sent

without a "fields" parameter.
- The following fields behave identically as they would in the standard Rally WSAPI: _rallyAPIMajor, _rallyAPIMinor, Errors, Warning, TotalResultCount, StartIndex, PageSize, and Results
- The following fields have no equivalent from Rally's standard WSAPI:
    - ETLDate. Indicates how up to date the data in the analytics engine is compared to "live" Rally.
    - ObjectID. This is not provided by default in the standard WSAPI but it is from this Analytics API
    - _ValidFrom
    - _ValidTo

ETLDate

Comparing the ETLDate timestamp to now will tell you how far behind the analytics engine is from the current time. We have borrowed the "point in time" consistent views concept from Multiversion Concurrency Control (MVCC). It's an ideal fit for us because of our NoSQL horizontal scaling (meaning no cross-document transactions) architecture... but even more so for this application than others that use NoSQL because we needed to keep the old versions anyway.

Paging with ETLDate. To ensure that the data that comes back on pages number 2 thru n, on larger queries, it's best to include a clause in the queries for the 2nd thru nth pages that limits the response to items that where _ValidFrom is before or equal to the ETLDate that was returned in the response for the 1st page:

```
{_ValidFrom: {$lte: <ETLDate from 1st page>}}
```

Additional fields available upon request

The following fields are also available if specified with the fields parameter:
- _ref: "https://rally1.../webservice/1.26/defect/777.js"
- _SnapshotRef: "https://.../query.js?find={_id: B2G}
- FormattedID. Not available on outgoing until/if we do hydrate, incoming it's converted to a query on _UnformattedID and _Type
- _UnformattedID (If FormattedID="DE2345", then _UnformattedID=2345)
- Revision information
- _Revision. OID of revision record
- _RevisionNumber
- _User. User who made the edit
- _SnapshotNumber
- Name
- Custom string fields
- All foreign key ObjectIDs (Workspace, Iteration, Release, Parent, Requirement, etc.)
- All numeric fields (PlanEstimate, TaskActualTotal, TaskEstimateTotal, etc.)
- All booleans
- All date fields
- Child Collections as lists of foreign key ObjectIDs: Tags, Tasks, Changesets, Defects, TestCases, Children, Duplicates, Predecessors, Successors

Missing:

- Big/rich text fields (Description, Notes, etc.)
- Attachments
- Weblink fields

Endpoints

Primary:

- `.../analytics-api/1.27/1234/artifact/snapshot/query.js?find:{"KanbanState":"Accepted"}`
  where 1.27 is the API version and 1234 is the ObjectID of the workspace

Other endpoints

- `.../analytics-api/x/....`
  to use the latest version of the API (like ALM)
- `.../analytics-api/1.27/1234/artifact/123456.js`
  return latest version of artifact with ObjectID 123456 in workspace 1234
- `.../analytics-api/1.27/1234/artifact/123456/snapshot.js`
  return all the snapshots with (_ValidFrom < ETLDate) for ObjectID 123456
- `.../analytics-api/1.27/1234/artifact/snapshot/987654.js`
  return specific snapshot 987654
- `.../analytics-api/1.27/1234/artifact/123456/snapshot/query.js?find:{"KanbanState":"Accepted"}`
  do the given query, adding in the criteria ObjectID = 123456
- Also, all of the above support .json extension.

Status endpoint:

If you hit the service endpoint with a GET and no parameters or with a POST and an empty body, the response will include the current status, base urls, and constants for the service. It will look something like this:

```
{
  _rallyAPIMajor: "1",
  _rallyAPIMinor: "26",
  BaseURI: "https://.../analytics/snapshot",
  InteractiveHelpURL: "https://.../analytics_help.html",
  MAX_PAGESIZE: 1000,
  DEFAULT_PAGESIZE: 5,
  ETLDate: "2011-10-10T12:34:56.789Z"
}
```

Since they are subject to change, your code should get constants like MAX_PAGESIZE from the service rather than assume some previously used value.

JSONP, CORS and cross-domain queries

Note: JSONP is not supported at this time. There were issues with using it in conjunction with basic authentication. We are exploring possibilities for cross-browser access including CORS and additional forms of authentication.

Authentication and authorization

Basic authentication is supported. Supply the properly encoded username and password in the header of each request.

Authorization is a bit different from Rally's standard web services. If you have read privileges for ALL work items in the response, the query will come back as expected. If you are missing read permission for ANY of of the work items that match your query, NONE will be returned. Rather, you will receive an "unauthorized" response.

Hierarchy

There are several different hierarchies that are represented in the documents.

Project hierarchy

The Project hierarchy is also represented as an array starting at a root Project for this Workspace. So if work item 777 is at the bottom of this Project hierarchy:

- Project 7890
    - Project 6543
        - Project 3456
            - Work item 777

The document for work item 777 would look like this:

```
{
  ObjectID: 777,
  Project: 3456,
  _ProjectHierarchy: [7890, 6543, 3456],
  ...
}
```

To retrieve all work items that are in Project 7890 or any of its child projects, you would simply include this clause in your query:

```
   _ProjectHierarchy: 7890
```

Project scope up

If you want to accomplish the equivalent of projectScopeUp, you'll need to submit two queries:. There are two conditions: (1) projectScopeUp in combination with projectScopeDown; and (2) projectScopeUp WITHOUT projectScopeDown.

For the former, you would (a) First, submit the query like above and keep the results as part of your result set. Let's say we're going to scopeUp on 3456, you would say:

```
      _ProjectHiearchy: 3456
```

(b) When you get that response, inspect the _ProjectHierarchy field of any of the responses that come back. For all of the values before the one you are interested in (3456 in this example), build another array to pass back into a $in clause ("[7890, 6543]" in this example). Then submit a query like this and add it to the earlier results:

```
      Project: {$in: [7890, 6543]}
```

For condition (2), you would (a) first submit the same query as 1a above but set the pagesize to 1 (This is favored over simply querying "Project: 3456" because it has a lower chance of hitting the problem in the note below.).

(b) You would inspect the results and submit the following:

```
      Project: {$in: [7890, 6543, 3456]}
```

what comes back is projectScopeUp without projectScopeDown.

Note, there is a risk that there are no results from the initial query of "_ProjectHiearchy: 3456". In that case, this approach will not work.

Work item hierarchy

The work item hierarchy works the same way using the _ItemHierarchy field. So if you have this hierarchy:

- Story 333
  - Story 444
    - Story 555
      - Story 666
        - Defect 777
          - Task 12
        - Task 13
    - Story 888
    - Story 999

The document for Story 666 would look like this:

```
{
  ObjectID: 666,
  _Type: "HierarchicalRequirement",
  Parent: 555,
  _ItemHierarchy: [333, 444, 555, 666],
  ...
}
```

To retrieve all Stories that descend from Story 333 (includes 333, 444, 555, 666, 888, and 999 but not Defect 777), you would include this clause in your query:

```
{
  _ItemHierarchy: 333,
  _Type: "HierarchicalRequirement"
}
```

Note: Until the _IsLeaf functionality described below is implemented (if ever), you can simulate it with either of these two query clauses: (1) for just Stories:

```
Children: null
```

(2) for Stories or Portfolio Items:

```
$or: [
    {_Type: "HierarchicalRequirement", Children: null},
    {_Type:"PortfolioItem", Children: null, UserStories: null}
]
```

Additionally, there is a virtual field named _IsLeaf that is set for all leaf Portfolio Items and Stories (but not Defect, Tasks, etc.). So, this query clause:

```
{
  _ItemHierarchy: 333,
  _IsLeaf: true
}
```

would match Story 666, Story 888, and Story 999 in the example tree above. Note, that we could exclude the _Type clause because _IsLeaf is only set to true for Stories (and leaf Portfolio Items that have no children Stories, once Portfolio Manager is released).

Note that _ItemHierachy will cross work item type boundaries. So the document for Task 12 would look like this:

```
{
  ObjectID: 12,
  _Type: "Task",
  Parent: 777,
  _ItemHierarchy: [333, 444, 555, 666, 777, 12],
  ...
}
```

So, if you wanted all of the Tasks that were descendant from some high level story like Story 333, you could get that with this query clause:

```
{
  _ItemHierarchy: 333,
  _Type: "Task"
}
```

Work Item type "hierarchy"

While this may not be as obvious at first, there is also a _Type hierarchy in Rally. A Defect is also an Artifact, is also a WorkspaceDomainObject, etc. We represent this in the analytics database like this.

```
{
  _Type: [
    "PersistableObject",
    "DomainObject",
    "WorkspaceDomainObject",
    "Artifact",
    "Defect"
  ],
  ...
}
```

The way queries against the analytics engine work, if you include:

```
 _Type: "Artifact"
```

as a clause in your query, it will match the _Type field in the example above. There is no need to say "contains" or provide any additional operator for this to match. It knows that if the target value is an array, any element of that array is sufficient for a match.

To query for Defects and HierarchicalRequirements:

```
  _Type: {$in: ["Defect", "HierarchicalRequirement"]}
```

Scenarios

The examples below show you how to retrieve data in a variety of scenarios.

1. Retrieve the entire history of a particular Rally work item

```
{objectID: 777}
```

You could just as easily have queried on FormattedID.

<technical_sidebar>When the data is updated from Rally, there are brief moments when the data in this analytics engine may not be self-consistent. For example, when a high level story is re-parented, the analytics engine needs to adjust the _ItemHierarchy field for all of the descendant stories. Since each story is kept in a separate "document" in the analytics database and the NoSQL technology that we are using does not support cross-document transactions (traditional SQL transactions are traded-off for horizontal scalability), we are using this ETLDate timestamp approach to make sure your queries are self-consistent. Until the long running re-parenting operation is complete, the ETLDate timestamp will continue to point to the last time in the analytics engine where all the changes have trickled across all effected documents.</technical_sidebar>

Note that for any query that doesn't already specify a criteria on _ValidFrom, the ETLDate ("current") timestamp is going to be added. The below query will actually be run:

```
{objectID: 777, _ValidFrom: {$lte: "current"}}
```

2. Retrieve the "current" version of a particular Rally work item

```
{objectID:777, __At: "current"} // "__At" has two underscores
```

The "current" will be replaced with the ETLDate timestamp which may be anywhere from a few milliseconds to a few minutes behind transactions in the primary Rally UI.

Note that the "__At" field is not actually a field in the analytics engine. It's syntactic sugar equivalent to:

```
{
  objectID: 777,
  _ValidFrom: {$lte: "current"},
  _ValidTo:{$gt: "current"}
}
```

You should feel free to explicitly use _ValidTo and _ValidFrom depending upon your needs.

3. Retrieve a particular Rally work item as it looked on a particular date

```
{FormattedID: "DE2345", __At:"2011-01-10T00:00:00Z"}
```

The date in the example above is midnight on January 9, 2011 in the Zulu timezone. It is preferable to specify the following day "2011-01-10T00:00:00Z" than to use 24:00:00 on the day in question, but "2011-01-09T24:00:00:00Z" is equivalent.

Further, it is permissible to exclude the lower order elements when they are zero. So, you could have written "2011-01-10T00Z".

Also, note that we used the "FormattedID" field instead of the ObjectID for this example. In the example below, we omit the "DE" prefix. The FormattedID is actually stored without the "DE" in the _UnformattedID field and it is stripped from queries so the below example is equivalent.

```
{
  _UnformattedID:"2345",
  _Type: "Defect",
  __At: "2011-01-10T00:00:00Z"
}
```

Note, the above translation example works well when a single FormattedID is specified. However, it's less obvious how it would work if multiple kinds of artifacts were specified. For instance, how would it behave if it recieved this:

```
  FormattedID: {$in["DE2345", "S1234"]}
```

The above example $in clause is supported by in the API because it automatically converts it to an appropriate $or clause. $exists, equals ":" are supported in this circumstance but other operators may not be supported. If you have need to compose a complicated query, it is best to use ObjectID rather than FormattedID. There is limited support for: $and, $or, $ne, $in, and $exists with FormattedID, although not all recursive combinations of $and and $or will work. Operators not supported include $lt, $lte, $gt, $gte, $all, $mod, $nin, $nor, $type.

4. Retrieve a set of work items as they looked on a particular date

```
{
  _ProjectHierarchy: 7890,
  _Type: "Defect",
  Priority: "High Attention",
  __At:"2011-02-01T00Z"
}
```

The above query will return the list of defects in Project 7890 and its sub-projects that had the their Priority field set to "High Attention" as of the end of January, 2011. Notice how the __At field is set to February 1. Our own charts will follow the convention of using 00:00:00.000 of the first day of the following month when trying to identify the state as of the end of a particular month. We will search for entities whose _ValidFrom and _ValidTo dates are "<" and ">=", respectively to the provided timestamp.

5. Retrieve defects that experienced a particular state transition in July 2011

```
{
  _Type: "Defect",
  Project: 7890,
  "_PreviousValues.KanbanState": {$lt: "Completed"},
  KanbanState: {$gte: "Completed"},
  _ValidFrom: {
    $gte: "2011-07-01TZ",
    $lt: "2011-08-01TZ"
  }
}
```

Because the analytics database doesn't know anything about ordering of the AllowedValues for the State field, it will look up your AllowedValues for the State field and convert the above query to something like:

```
{
  _Type: "Defect",
  Project: 7890,
  "_PreviousValues.KanbanState": {$in: [null, "Idea", "Defined"]},
  KanbanState: {$in: ["Completed", "Accepted", "Released"]},
  _ValidFrom: {
    $gte: "2011-07-01TZ",
    $lt: "2011-08-01TZ"
  }
}
```

The State field is "required" (not-nullable) in Rally but the null value is considered the lowest value in the ordered list of AllowedValues so it is included in every such query regardless of the current "required" setting for this field. The thinking is that you may have recently set the field to "required" meaning that there might be some old snapshots where the value is missing. This approach handles all cases as expected. Note, that MongoDB handles this as expected.

6. Find out how long a work item has been in its current KanbanState

```
{
  find: {
    ObjectID: 777,
    KanbanState: <current KanbanState>,
    "_PreviousValues.KanbanState": {$ne: <current KanbanState>}
  },
  sort: {_ValidFrom: -1},
  pagesize: 1
}
```

This will return the snapshot of the most recent time that ObjectID 777 had transitioned into its current KanbanState. This assumes you have previously retrieved the latest version of ObjectID 777 and know its current KanbanState. The _ValidFrom field in the record that is returned is the timestamp of when it entered the current KanbanState. You simply subtract that from today() to find out how many day's it has been in this KanbanState.

If you batched up a list of work items, you could replace the ObjectID caluse with:

```
   ObjectID: {$in:[<your list of ObjectIDs>]}
```

and get them all back at once. However, you'd have to drop the sort and limit clauses and figure out on the client which one was the last one for each work item. You decide which is better for your situation.

7. Create a Portfolio Item, or Story (epic) burn chart... or cumulative flow chart

If you have this:

- Portfolio Item 333
    - Story 444
        - Story 555
            - Story 666
                - Defect 777 (via the Requirement field)
                    - Task 12
                - Task 13
        - Story 888
        - Story 999

To get all leaf stories that descend from PI 333 (666, 888, and 999) on a particular date and get the PlanEstimate, and ScheduleState fields on January 1:

```
{
  find: {
    _ItemHierarchy: 333,
    $or[
        {_Type: "HierarchicalRequirement", Children: null},
        {_Type: "PortfolioItem", Children: null, UserStories: null}
    ],
    // Later use "_IsLeaf: true" instead of above "$or" clause
    __At: "2011-01-01T00:00:00.000Z"
  },
  {fields: ["PlanEstimate", "ScheduleState"]}
}
```

You would then vary the __At clause value to get the other points for the chart.

8. Provide analytics about Defects below some high level Story

"From the EMC... A strong interest was expressed in rolling up defect information by feature ie. for defects linked to a set of stories which roll up to epics and to features provide some analysis at the epic level about those defects."

So, if you have this:

- Story 555
    - Story 666
        - Defect 777 (via the Requirement field)
            - Task 12
        - Defect 000
        - Task 13
    - Story 888
        - Defect 999

and you have a query clause like this

```
  _ItemHierarchy: 555
```

it will match Story 555, Story 666, Defect 777, Task 12, Defect 000, Task 13, Story 888, and Defect 999. Adding a restriction on _Type will get you just the Defects. So, for EMC's scenario, you want just Defect 777, Defect 000, and Defect 999:

```
{
  _ItemHierarchy: 555,
  _Type: "Defect"
}
```

9. Create your own throughput report except use the state transition to Completed instead of Accepted

You can see from scenario 5 how to sense when a particular state transition occurs. You can expand on this to build your own throughput report and control what boundary is counted. The example below keys off of the "State" field but it could just as easily have keyed off of a custom field like "KanbanState".

```
{
  find: {
    _ProjectHierarchy: 7890,
    State: {$gte: "Completed"},
    _PreviousValues.State: {$lt: "Completed"},
    _ValidFrom: {
      $gte: <first day of some month>,
      $lt: <first day of next month>
    }
  },
  pagesize: 0
}
```

Because we set the pagesize to zero, the response would not contain any of the actual work items but the header information for the response would contain the count needed for this Throughput metric in the TotalResultCount field. We will eventually add metrics like sum and sum of squares to allow you to calculate other metrics. For now, if you want more than the count, you'll need to do that math on the client.

You would then iterate, the <...month> literals until you had the data you needed.

Query and response details

You should keep the following in mind when working with this API:

- The API is JSON-only. No XML although we may do that later and there may be an option to get it in csv format also.
- The API is REST(-ish) only. There is no SOAP interface. We say REST-ish, because you can use POSTs against this API as an alternative to GETs and the reference-following mechanism recommended (HATEOAS) in Roy Fielding foundational dissertation on REST are not completely followed by this API.
- The versioning of this API will be kept in sync with Rally's other web services APIs. If you are using version 1.26 of Rally's other API(s), you can also expect to be able to use version 1.26 of this API.
- However, we may make backward-compatible changes to this this API without incrementing the version number. For example, the **addition** of query functionality to this API would not require a version increment. However, removing or changing the syntax for some functionality would require a revision. Because there is no WSDL to change in a REST API as there would be in a SOAP API, this less strict approach should be acceptable for maintaining backward compatibility.

Warnings

- [REMOVE ONCE WE LEAVE ALPHA] It is an alpha API. We WILL change it on you without incrementing the version. We will tightly control who is using the API during this alpha period and make sure to give users advance notice before any potentially backward breaking changes are made.

- [MODIFY AS WE ADD SUPPORT FOR OTHER ENTITIES AND REMOVE ONCE WE LEAVE ALPHA] This API is focused on Hierarchical requirements (User Stories), Defects, and Tasks for now. The API may actually give you access to other entities like TestCases, but for these other entities, you should have lower expectations of stability (such as it is anyway in an Alpha API), and documentation as well as indexes and denormalization for efficient querying. Portfolio Items (new with the about to be released Rally Portfolio Manager), will be next on the list.

Dates, timestamps, and timezones

All date and time are stored in the analytics database in GMT and communicated to/from using ISO-8601 format using the following possible notations:

- "2011-02-01T00:00:00.000Z" - "Z" means "zulu" time or GMT.
- "2011-02-01T00:00:00Z" - You can omit the milliseconds with "Z" form as well as the forms below. In fact, you can omit all of the lower order values that you want to be zero. "2011-02-01T00Z" is equivalent. [Question: Can you leave off all of the zeros like "2011-02-01TZ"? Does this make parsing to figure out when to add the user's time zone harder like "2011-02-01T"?]
- "2011-02-01T00:00:00-05:30" To specify a timezone shift in ISO format, use the shift time (e.g. "-05:30") in place of "Z".

Paradigm shifts

The past is unchangeable

We want to encourage you to think of the past as unchangeable. This is not as obvious as it sounds and it's actually a departure from the way some of our current reports work. This is best explained with an example. Let's say you want to trend the count of all open P1 defects within in a particular project over time. One way to do this would be to first find all P1 defects that are currently in that project and then look at the history of each of those defects to find when they were open and resolved. Using this approach, any defects that had moved into the project during the time of interest would would be included in ALL of the data points on the trend line and any that had moved out of the project at an earlier date would be counted in NONE of the points. Similarly, if they didn't start out as P1s or were downgraded from P1 priority later, that wouldn't matter. By first finding the defects that are NOW in the project and NOW have P1 priority, you have essentially changed the past.

Rather, we want you to think of the generation of this defect trend line as the application of the same query at particular moments in the past. So, if you want to trend by month starting three months ago, you run the query against the data as it looked three months ago, and then you run it again against the data as it looked two months ago, and then finally against the way it looked at the most recent month boundary. If a defect was assigned to your project three months ago but isn't now, it will be included in the count back then but excluded from this month's count.

The reason for this design choice is simple, by taking this approach, we can warehouse the aggregations from any queries that you run for fast retrieval the next time they are run. It also turns out that this is usually the preferred behavior. We decided to trade-off the ability to perform the alternative type queries for those rare cases where it is preferable in exchange for a huge speed and simplicity advantage for the common case.

A different security model

The current Rally security model is based upon pre-identifying the projects for which you currently have permission BEFORE it even knows what data satisfies your request. Once you are authenticated and your current project scope is set, we then know all of the projects within the project scope for which you currently have permission. This list of permissible projects is pre-populated upon each request and it becomes a part of any queries that are submitted to the database ("... WHERE project IN [039284, 039284, 093923]..." for example). This works great for processing the typical read and edit transactions that you do in Rally every day, but it's not ideal for analytics. Under the current model, it's possible for two users to request the exact same report but to get back different results because they have different permissions. Worse, the current analytics engine has no way to warn users of this because, under the current security model, it has no way to know it's missing some data. The "WHERE projects IN ..." clause is automatically added to every query made by the current analytics engine.

We wanted to fix this, so **the new analytics security model doesn't check your permissions until AFTER it has figured out what data matches your query.** Every restricted entity (includes User Stories, Defects, and Tasks) in the new analytics repository has a project field. After a query against these entities is complete, it will extract the set of projects mentioned in the returned entities and compare it to the list of projects for which you have permission. If you have permission for all of them, no worries. If you are missing read permission for one or more, the request will return with an error listing the projects for which you do not have permission. [MAYBE THIS NEEDS TO BE A SETTING. SOME FOLKS MIGHT NOT LIKE US SENDING BACK A LIST OF UNPERMITTED PROJECT NAMES. HOWEVER, I WOULD LIKE TO MAKE IT A WORKSPACE SETTING AND HAVE IT BE ENABLED BY DEFAULT BECAUSE I THINK IT IS MUCH MORE USABLE TO TELL THEM WHICH ONES THEY NEED TO GET ADDED.]

Note, that this new security model interacts with the "past is unchangeable" concept. The list of projects will include projects that were pointed to by past versions of work items even if those past projects have been deprecated and contain no work at this time.

FAQ

Q: Is this how things are actually stored?
A: Pretty much. We're trying to avoid impedance mismatch between tables and objects. We're trying to make it fast, clean, and simple.

Q: Why is the API so close to the native MongoDB interface?
A: We like it. In fact, we had an earlier design that was based upon another analytics oriented database technology (CouchDB) and for that design we were still borrowing the MongoDB syntax.

There are all sorts of derivatives of SQL (JQL, MDX, TSQL, etc.). The next phase of evolution for web applications, will see an explostion of JSON-based query languages. With MongoDB's popularity, we wouldn't be surprised if others built upon it.

Note that the API is not blindly passing along whatever you give it. We convert your find clauses into objects; validate and modify them; and then build a query for MongoDB. If we ever have to go away from MongoDB, we will re-implement the query language and translate it to whatever technology is chosen.

Q: Why does this API use "fields" rather than "fetch" like Rally's standard web services API?
A: We might be convinced otherwise if there is an uproar over this difference, however, we felt that the semantics for "fetch" are overloaded. It both hydrated the sub-objects as well as specified what fields to return. We wanted to make those two things separate.