treediffer

Release 0.0.1

Dec 09, 2020

Contents

CHAPTER 1

Diff all the things

Diffs are very useful when working with content, especially at scale.

Studio librarians, admins, and LE staff need diffs to keep track of content changes. Curators need diffs to see what new materials are added when content channels are updated. Teachers and learners operating in the "we're running out of materials" mode can watch the diffs to see a list of only the new things added in the current update.

1.1 JSON trees

Tree structures are ubiquitous in the data model throughout the Kolibri platform, and each of these tree structures can be serialized in the form of a JSON tree:

- T0: Ricecooker web archives format? (to know if source website has changed and we need to re-run the chef)
- T1: Ricecooker json input format (debug cheffing code, record input paths before processing)
- T2: Ricecooker->Studio wire format (archive chef runs, see detailed channel diff before uploading to studio)
- T3: Studio json tree format (compute diff(main tree,staged tree) before DEPLOY click)
- T4: Kolibri DB wire format sqlite3 files (show detailed diff info on UPDATE CHANNEL screen)
- T5: Kolibri tree format (show detailed diff info on IMPORT MORE screen)

1.1.1 JSON trees for diffs

Main focus of this project is T2 diffs in ricecooker, and T3 diffs for reviewing changes between staging tree and main tree in Studio.

1.1.2 JSON trees for PUBLISH (a.k.a. EXPORT)

The Studio channel export task needs the information about the studio main tree that is being published which can be "serialized" as a tree T3, with no further need for the export code to "touch" the Studio DB (but may need interaction

with GCP content bucket)

The T3 tree format has several uses:

- A1 = representation format for doing diffs [see diff(main tree,staged tree) above]
- A2 = archival (save studio json tree as "snaphots" or "staged versions", archive full studio tree data on PUBLISH version, could put in git)
- A3 = share export code between ricecooker and studio (single path for all the Kolibri backward compatibility code)
- A4 = publish staging tree (a Studio->Kolibri side-channel for importing a special "draft" channels exported from the staging tree, e.g. In Kolibri import from https://drafts.studio.leq.org/content/databases/{ch_id}.sqlite3 == PUBLISH({ch_id}.staging_tree)

CHAPTER 2

Specs

2.1 Business Requirements Specification

- Know what has changed between two channel "states" (e.g. diff(staged,main) studio trees)
- Desirable changes to content channels propagate quickly through the content pipeline
- Unnecessary and undesirable changes are stopped and corrected at an early state of the pipeline
- Is able to process large trees like Khan Academy (think 500MB of uncompressed JSON)

2.2 Software Requirements Specification

Can be subdivided into essential functionality, optional, and stretch goals.

The essential objectives are:

- Produce a "summary diff" that contains only the counts of added/removed/moved/modified
- Given any two trees (ricecooker, studio, kolibri) produce the detailed diff information about the differences (added/removed/moved/modified) between the two trees

Optional (depending on frontend needs):

- Remove redundancy in diff format (e.g. show only node move instead of node add, node delete, and node move)
- Post-process the diff format to make it most convenient for presenting to users

Stretch goals:

- Post-process the diff to make a minimum description length and avoid information overload (e.g. instead of showing 30 content nodes added, display the diff as the action of adding a single topic node)
- Support kinds of trees:
  - JSON: ricecooker, studio wire format, studio trees
  - Django ORM: Studio, Kolibri

- Basic ORM: standalone script processing of kolibri sqlite3 DB files

CHAPTER 3

Design

Given any two tree-like data structures, let's call them `oldtree` and `newtree`, that consist of dict-like nodes with `node_id`-like and `content_id`-like attributes.

3.1 Tree diff format

A tree diff is a dictionary that describes four types of edits:

- `nodes_deleted`: list of nodes that were present (by `node_id`) in `oldtree` and absent in the `newtree` (by `node_id`)
- `nodes_added`: list of nodes added to the tree (includes both nodes ADD and COPY actions)
- `nodes_moved`: nodes in the `newtree` that have the same `content_id` as a node in the `oldtree` by a new `node_id` has changed. If multiple nodes satisfy this criterion, the only the first node is retuned (in tree-order).
- `nodes_modified`: to be included in this list, a node in the `newtree` must have the same `content_id` as a node in the `oldtree` and modified attributes.

Note a node may appear in more than one of the above lists. For example a node that was moved from location X in the tree to location Y in the tree, and whose title was edited will appear in all four lists:

- in the `nodes_added` list because it now appears in location Y
- in the `nodes_deleted` list because it was removed from location X
- in the `nodes_moved` list since we can detect the node has the same `content_id`
- in the `nodes_modified` list because of the title edit

For more info, see the detailed diff examples in Tree diffs.

3.2 Technical notes

Invariant: the information in a tree diff `treediff(oldtree, newtree)`, when applied as a "patch" to the `oldtree` should produce the `newtree`.

Data format: each of the "diff items" in the four lists includes additional structural annotations like `parent_id` and all node attributes like title, description, files, assessment items, tags, etc., even if they haven't changed to allow for easy display of diffs and post-processing tasks.

3.3 Diff limitations

- Will not recognize assessment items that are moved between exercises
- Nodes from the `oldtree` can be duplicated in several places in the `newtree`. The first of these uses will be recognized as a move, while all others will be recognized as if the nodes were added. A more appropriate way to do this would be to recognize them as `nodes_copied` (a special type of added, where a node with the same content_id in the `oldtree` exists).
- The logic modified and moved logic assumes `content_id` is available for all nodes (topic nodes and content nodes). Need to watch out to make sure this assumption is valid in all cases where we want to use this (ricecooker, studio and Koilbri, and if necessary relax this in future).

3.4 External API (High Level)

The API for this library is to call the `treediff` function which has signature:

```
treediff(oldtree, newtree, preset=None, format="simplified",                    # HL
         attrs=None, exclude_attrs=[], mapA={}, mapB={},                        # LL API
         assessment_items_key='assessment_items', setlike_attrs=['tags'])       # LL API
```

The line tagged with `HL` is the "high level" API for the library, where users just need to specify the two trees they need diffed (e.g. `ricecooker`, `studio`, or `kolibri`), which will set the appropriate values for the low-level API kwargs.

The `raw` format returns the diff in the "internal representation" described above that includes duplication (primarily used for debugging), the `simplified` format (default) removed moved nodes from the added and deleted lists but keeps all results as flat lists. The `restructured` format tries to return the diff as "chunks" of tree (i.e. indicate an addition of topic + 3 children as one addition, instead of four separate additions).

3.5 Internal API (Low Level)

The functions the `treediff` module are named `diff_` and accept the following standard set of keyword arguments, which we'll call the "low level" API:

```
LL = dict(
    attrs=None, exclude_attrs=[], mapA={}, mapB={},                         # LL API
    assessment_items_key='assessment_items', setlike_attrs=['tags'])        # LL API
)
```

- `attrs` (list(str)): if specified, only these attributes will be check for differences the default value is set to `None` which means all values will be checked.

- `exclude_attrs` (list(str)): do not check these attributes, e.g. for cases where we expect them to be different and we want to avoid false positives.
- `mapA` (dict): map from the common diff attributes to attributes nodes in `treeA`.
- `mapB` (dict): map from the common diff attributes to attributes nodes in `treeB`.
- `assessment_items_key` (str): specify where to look for assessment items, in Studio and Kolibri leave the default value. In ricecooker, set to `questions`.
- `setlike_attrs` (list(str)): treat these attributes as sets (i.e. order doesn't matter)

Note: in general users should not need to set the low level API params and can instead rely on one of the presets, which will set the appropriate keyword args, e.g. using `preset="studio"` is equivalent to call with `**studio_preset_kwargs`.

3.5.1 Attribute maps

The arguments `mapA` and `mapB` are used to make the same diff logic work for trees that have different attributes names. The internal diff logic always works will look for the following standard attributes (derived from Studio data model):

- Nodes:
  - `node_id`: an unique identifier that represents the node's position within the tree
  - `parent_id`: the node ID of the parent node
  - `content_id`: a persistent identifier for the content of the node (allows us to detect moves)
  - `sort_order`: position within parent
- Assessment items:
  - `assessment_id`: like content_id for assessment items (allows us to detect moves)
  - `order`: sort order of the question within the exercise

Example attribute map used to map the "standard attributes" to the attributes used by ricecooker JSON trees:

```
ricecooker_map = {
    # channel attrs
    "root.node_id": "id",              # a.k.a. channel_id
    "root.content_id": "source_id",    # unique identifier within source_domain
    #
    # node attrs
    "license_name": "license.license_id",
    "license_description": "license.description",
    "copyright_holder": "license.copyright_holder",
    "role_visibility": "role",
}
```

Note the special `root.*` attributes which are used when processing the root node. Note also the values in the `ricecooker_map` use `.`-accessor patterns, which tell to look use the value of the nested dict object.

3.5.2 List of functions that use the low level API

If we call the keyword args of the low level API `**LL` we can write the entire functions of the `treediffer` library as follows:

```
# PHASE 1: diffing
def diff_subtree(parent_idA, nodeA, parent_idB, nodeB, **LL):
    "Compute the changes between the node `nodeA` in the old tree and the..."

    def diff_attributes(nodeA, nodeB, **LL):
        "Compute the diff between the attributes of `nodeA` and `nodeB`.
        def diff_files(listA, listB):
             "Compute the diff of two lists for files, treating them as set-like.""
        def diff_assessment_items(listA, listB, **LL):
            "Compute the diff between the lists of assessment items `listA` and␣
↪`listB`,

    def diff_children(parent_idA, childrenA, parent_idB, childrenB, **LL):
        "Compute the diff between the nodes in `childrenA` and the nodes in␣
↪`childrenB`."

# PHASE 2: move detection
def detect_moves(nodes_deleted, nodes_added):
    "Look for nodes with the same `content_id` that appear in both lists, ..."

# PHASE 3: simplification
def simplify_diff(raw_diff):

# PHASE 4: restructuring
def restructure_diff(simplified_diff):
```

The "main" work happens in `diff_subtree` which computes the diff of node attributes and recursively calls diff_subtree on all children down the tree (PHASE 1). Node moves are detected are detected as a post-processing step (PHASE 2), and so are the format conversions for presentation needs (PHASE 3 and PHASE 4).

CHAPTER 4

Tree diff formats

The functions in `treediffer` are used to find differences between two trees. The resulting diff contains both structural information about the nodes' position in the tree as well as the nodes' attributes.

4.1 Example of node deleted

A node deletion appears under the key `nodes_deleted.{node_id}` and contains the structural info where the deletion occurred, and the attributes of the node that was deleted:

```
nodes_deleted = [
    {
        "old_node_id": (str),
        "old_parent_id": (str),
        "old_sort_order": (float),
        "content_id": (content_id),
        "attributes": {
            "title": {"value": "The title of the deleted node"},
            "description": {"value": "The description of the node that was deleted"},
            "content_id": {"value": (str)},
            "sort_order": {"value": (float)},
            ... }
    },
    {},
    ...,
    {},
]
```

To apply this deletion patch, find content node `c = ContenNode.filter(tree_id=...).get(node_id=deleted['node_id'])`, and its parent `parent_node = ContenNode.filter(tree_id=...).get(node_id=deleted['old_parent_id'])`, then call `parent_node.children.remove(c)`. The `attributes` dict is provided for information purposes only (e.g. to display info about the node being deleted in the UI).

4.2 Example of node added

A node added to the tree is appears under the key `nodes_added.{node_id}` and is represented by the following dict:

```
nodes_added = [
    {
        "parent_id": (node_id),
        "node_id": (node_id),
        "sort_order": (float),
        "content_id": (content_id),
        "attributes": {
            "title": {"value": "The title of the new node"},
            "description": {"value": "The description of the new node"},
            "content_id": {"value": (str)},
            "sort_order": {"value": (float)},
            ... }
    },
    {},
    ...,
    {},
]
```

To apply this "patch," create a new `c = ContentNode(**map(..., attributes))`, find the parent node where the new node is to be added, `parent_node = ContenNode.filter(tree_id=...).get(node_id=added['parent_id'])`, then call `parent_node.children.add(c)`.

Note: when applying a patch with multiple node additions, watch out for sort order in systems that don't know how to handle `sort_order` (ricecooker).

4.3 Example of node moved

Nodes in the `newtree` that have the same `content_id` as a node in the `oldtree` but whose `node_id` has changed can be interpreted as moves. If multiple nodes satisfy this criterion, the only the first node is treated as a move (in tree-order) while others "node clones" of the same `content_id` are recorded as additions.

```
nodes_moved = [
    {
        "content_id": (content_id),
        "node_id": (node_id),
        "old_node_id": (node_id),
        "parent_id": (node_id),
        "old_parent_id": (node_id),
        "sort_order": (float),
        "old_sort_order": (float),
        "attributes": {
            "title": {"value": "The title of the new node"},
            "description": {"value": "The description of the new node"},
            "content_id": {"value": (str)},
            "sort_order": {"value": (float), "old_value": (== old_sort_order)},
            ...},
    },
    {},
    ...,
```

(continues on next page)

(continued from previous page)

```
    {},
]
```

4.4 Example of node modified

When a node appears in both the "before tree" and the "after tree" in the same `node_id` (position within the tree) but who have different attributes. These "attribute diffs" are stored under the keys `old_value`, `value` = new value, with special handling for set-like attributes (files and tags), and list-like attributes (assessment_items).

Example data structure:

```
nodes_modified = [
    {
        "node_id": (node_id),
        "parent_id": (node_id),
        "content_id": (content_id),
        # intentionally not tracking changes in sort_order since those are defined as␣
↪moves
        "changed": ["title", "sort_order", "tags", "assessment_items"],
        "attributes": {
            "title":{
                "old_value": "Old title",
                "value": "The new title for the node"
            },
            "description": {"value": "The description of the new node"},
            "content_id": {"value": (str)},
            "sort_order": {"value": (float), "old_value": (== old_sort_order)},
            "tags": {                                                          # set-
↪like
                "old_value": ['oldtag1', 'oldtag2'],
                "value": ['oldtag1', 'newtag3', 'newtag4'],
                "tags_removed": ['oldtag2'],
                "tags_added": ['newtag3', 'newtag4'],
            },
            "assessment_items": {                                              #␣
↪list-like
                "old_value": [
                    {"id": "aiid1", "assessment_id": 'q1', ... },
                    {"id": "aiid2", "assessment_id": 'q2', ... },
                    {"id": "aiid3", "assessment_id": 'q3', ... }
                ],
                "value": [
                    {"id": "aiid1", "assessment_id": 'q1', ... },
                    {"id": "aiid4", "assessment_id": 'q4', ... },
                    {"id": "aiid3", "assessment_id": 'q3', ... },
                ],
                "deleted": [
                    {"id": "aiid2", "assessment_id": 'q2', ... },
                ]
                "added": [
                    {"id": "aiid4", "assessment_id": 'q4', ... },
                ]
                "moved": [],
                "modified": [],
```

(continues on next page)

(continued from previous page)

```
            },
        },
    },
    {},
    ...,
    {},
]
```

4.5 Node structural annotations

4.6 Flat diffs

Suppose a topic node T1 is added which has two children N1 and N2. Depending on the use case for the diff, we want to represent this change in different ways. The `raw` and `simplified` diff formats correspond to flat lists, so this change will appear as three separate items in the `nodes_added` list:

```
nodes_added = [
    ...,
    {
        "parent_id": (node_id(parentT1)),
        "node_id": (node_id(T1)),
        "sort_order": (float),
        "content_id": (content_id(T1)),
        "attributes": {
            "title": {"value": "T1"},
            "description": {"value": "The description of the new topic node"},
            "content_id": {"value": (str)},
            "sort_order": {"value": (float)},
            ... }
    },
    {
        "parent_id": (node_id(T1)),
        "node_id": (node_id(N1)),
        "sort_order": 1.0,
        "content_id": (content_id(N1)),
        "attributes": {
            "title": {"value": "N1"},
            "description": {"value": "The description of the first new node"},
            "content_id": {"value": (str)},
            "sort_order": {"value": 1.0},
            ... }
    },
    {
        "parent_id": (node_id(T1)),
        "node_id": (node_id(N2)),
        "sort_order": 2.0,
        "content_id": (content_id(N2)),
        "attributes": {
            "title": {"value": "N2"},
            "description": {"value": "The description of the second new node"},
            "content_id": {"value": (str)},
            "sort_order": {"value": 2.0},
            ... }
```

(continues on next page)

(continued from previous page)

```
    },
    ...,
]
```

This is appropriate for counting number of nodes added/deleted/moved/modified.

4.7 Restructured diffs

In the `restructured` diff format we'll combine these additions into a single logical addition of the topic, and indicate the presence of the child notes as children to the top-level topic addition:

```
nodes_added = [
    ...,
    {
        "parent_id": (node_id(parentT1)),
        "node_id": (node_id(T1)),
        "sort_order": (float),
        "content_id": (content_id(T1)),
        "attributes": {
            "title": {"value": "T1"},
            "description": {"value": "The description of the new topic node"},
            "content_id": {"value": (str)},
            "sort_order": {"value": (float)},
            # no children     # <<< Note: children treated outside of attributes
            ... },
        "children": [
              {
                  "parent_id": (node_id(T1)),
                  "node_id": (node_id(N1)),
                  "sort_order": 1.0,
                  "content_id": (content_id(N1)),
                  "attributes": {
                      "title": {"value": "N1"},
                      "description": {"value": "The description of the first new node
↪"},
                      "content_id": {"value": (str)},
                      "sort_order": {"value": 1.0},
                      ... }
              },
              {
                  "parent_id": (node_id(T1)),
                  "node_id": (node_id(N2)),
                  "sort_order": 2.0,
                  "content_id": (content_id(N2)),
                  "attributes": {
                      "title": {"value": "N2"},
                      "description": {"value": "The description of the second new node
↪"},
                      "content_id": {"value": (str)},
                      "sort_order": {"value": 2.0},
                      ... }
              },
        ]
    },
```

(continues on next page)

(continued from previous page)

```
    ...,
]
```

This format would is better suited for displaying the changes in a UI in a compact logical manner to avoid overwhelming users with long lists.