A&R HISTORICAL DATA MIGRATION: MIGRATION ITINERARY – 2017-2019

OVERALL POINTS

AWS TRANSACTWRITE

Due to the quantity of records that must be brought into the system, AWS DynamoDB TransactWrite is leveraged to reduce processing time. TransactWrite lumps up to 25 database operations into a single all-or-nothing transaction. All operations must succeed for the transaction to pass. If at least one operation fails, the whole transaction fails and none of the operations are executed.

This logic greatly improves migration time, but the tradeoff is failing to migrate valid records that were part of a failed TransactWrite. All failed records will be captured.

Additionally, only a single database operation can be carried out on any given database item per transaction, which may necessitate additional logic to break up transactions and avoid unnecessary failures.

IDEMPOTENCY

The migration has been set up to be idempotent. It can be run many times without changing the outcome. This is useful in ensuring records are not duplicated or overwritten but it has drawbacks when coupled to AWS TransactWrite. TransactWrite has a ConditionExpression set up to ensure existing database items are not overwritten, but if this ConditionExpression fails (ie the database item already exists), the whole transaction will fail. Therefore, running a migration back-to-back with the same source data will result in ballooning failures.

DATA SETUP

SCHEMA VALIDATION:

The migration schema that maps new and old fields is too large to be visually checked. There is a safety check built in that will throw if the schema has been invalidated in any way.

- The schema length must equal the number of columns in the spreadsheet (83).
- Each entry in the schema must have the correct properties.

GET DATABASE SNAPSHOT:

The migration frequently requires existence checks of items in the current database. To reduce the number of DB hits, a DB snapshot of the park/subarea structure is taken early on and is used for existence checking later on.

The snapshot only captures information that will help construct AWS DynamoDB primary keys for legacy records.

```
'<orcs>': {
  '<subAreaId>': { subAreaName: '<subAreaName>', activities: [Array]}
}
'1003': {
  '0028': { subAreaName: 'Bishop Bay Hot Springs', activities: [Array] }
},
'1030': { '0055': { subAreaName: 'Cascade', activities: [Array] } },
'3883': {
  '0351': { subAreaName: 'Main Parking Area', activities: [Array] },
  '0352': { subAreaName: 'River Mouth', activities: [Array] }
},
```

LOAD HISTORICAL DATA:

The historical data is parsed using the schema and brought into memory.

HISTORICAL DATA VALIDATION

The historical data is parsed for potential issues that might prevent migration, and a list of migration operations is created (create new parks, create new subareas, etc.).

Rows of data that fail to migrate will be captured.

DETERMINE NEXT SUBAREA ID

Since imported legacy parks must be given a new subarea Id on import, the next available subarea Id is determined based on the largest subarea Id currently in the system.

VALIDATE SPREADSHEET ROW

The data row is validated. A row is considered invalid under any of the following conditions:

- Missing orcs, park name, subarea name, year or month.
- Orcs is not a number.

If a row is invalid, it is captured as a failure and the migration skips to the next row.

DETERMINE ROW ACTIVITIES

The activities within the row are extracted based on the schema. Each column in the row belongs to an activity. If a column in the row is ‘populated’, then that row is considered to have the activity associated with that column, and that activity is pushed to a list of activities for that row. The activities can any combination of the following (note: the ‘Meta’ activity exists to capture metadata and is not included).

- Frontcountry Camping
- Frontcountry Cabins
- Backcountry Camping

- Backcountry Cabins
- Group Camping
- Day Use
- Boating
- Legacy Data

The number of activities within a row correlates to the number of legacy records that will be added to the system for that row, so the running tally of activity counts is recorded.

If the row has no activities, or if an error is encountered while parsing for activities, the row is captured as a failure and the migration skips to the next row.

DETERMINE LEGACY PARKS

If the park in the row does not exist in the current system, we must create a legacy park to house the legacy records. We can easily check if a park does not exist in the current system by searching for the ORCS provided in every row of the historical data.

If there is no matching park in the current system (and a legacy park has not already been created), then a legacy park will be created.

DETERMINE LEGACY SUBAREAS

If the subarea in the row does not exist in the current system, we must create a legacy subarea to house the new legacy records. Unfortunately, the key subarea identifiers differ between the new system and the historical data - the new system has Subarea IDs, while the old system just relies on the subarea name/park combination to be unique.

The name of the historical subarea can be extracted from the historical data by slicing the string in the column labelled 'Park Sub Area', where the format of the string is '<Park Name> - <Subarea Name>'. After slicing the string at the first hyphen, the remainder of the string will be used for the subarea name comparison.

The name of the subarea in the historical data will be compared to the name of the park in the new system's db. If there are no matches (and a legacy subarea has not already been created), then a legacy subarea will be created.

Every time a legacy subarea is created, the next subarea id is incremented.

VALID ROW

If a row is deemed valid, it is saved to the change log along with a list of activities that were detected in the row, and the subarea id that will be assigned to the row. The change log is the object that will be traversed when creating new database items when the migration is executed.

POST VALIDATION

After the above validation steps are complete, the user is prompted with a quantity of successes/failures.

The user can choose to view a list of the legacy parks and subareas that are slated for creation. They can also choose to see a high-level list of the encountered failures.

After reviewing the above lists, the user must enter “Y” to approve of the change log before the actual migration can commence.

MIGRATION EXECUTION

CREATION OF LEGACY PARKS

The first step of the migration is to create legacy parks, so that legacy subareas have a parent park to be tied to in a future step. Parks are created via AWS DynamoDB TransactWrite (up to 25 at a time). Conditional checking of parks should not be necessary at this point since there should be no parks in the creation list that are already in the system, but conditional checking is active for now anyways.

CREATION OF LEGACY SUBAREAS

Following parks, legacy subareas are then created. Since we need a two-way link between park and subarea, there are two steps that are executed for each legacy subarea:

1. Legacy subarea is created with a field linking it to its parent park’s ORCS
2. The parent park’s list of subareas is updated with the addition of the new subarea. The new addition to the list will have the object structure:

```
{
  "name": "<subarea name>",
  "id": "<subarea id?>",
  "isLegacy": "T/F <legacy subarea flag>"
}
```

Because both steps must succeed for the two-way link to complete, we must ensure that both the ‘PUT’ and ‘UPDATE’ operations in steps 1 & 2 occur as part of the same transaction. This way, either they both succeed, or they both fail, and we will know the difference.

Additionally, since we cannot perform more than 1 action on the same database item per transaction, we must keep track of which parks have been touched per transaction, and break up the transactions when we encounter a duplicate.

Because of the above rules we will end up with transactions of varying sizes. Conditional checking, like parks, is active but probably is not necessary.

Subareas objects in the new system contain a list of activities so that an end user may create a new record of that activity type for that subarea without there being a previous record in the system. Since legacy data is not editable, activities of legacy records will NOT be appended to their parent subareas.

CREATION OF LEGACY RECORDS

When creating legacy records, we will be creating one legacy record for every activity we detect within a given row of historical data (1-8 new records/row). To do this, the change log is traversed. In an earlier step, we linked a list of activities in the row to the row itself, so we can pre-allocate a set number of transaction slots for each row.

Each legacy record will be a ‘PUT’ and we must ensure that all legacy records in a row are included in the same transaction. This way, either they all succeed, or they all fail, and we will know the difference.

When a legacy record transaction is successful, all included records are saved as ‘successes’.

Legacy records will be flagged with an ‘isLegacy’ Boolean for quick identification. They will also be time-stamped with the datetime they were migrated in the ‘lastUpdated’ field. Finally, any legacy fields that do not map over from the historical data will be captured as key:value pairs within a new ‘legacyData’ subobject.

POST EXECUTION

When the migration is complete, the user will be presented with a list of successes and failures. They will have the option to generate 4 post-execution reports:

- List of legacy parks created
- List of legacy subareas created
- List of legacy records created
- List of historical data rows that failed to be transferred into the system

The last report of failures is a .csv that is structured the same way the historical data is, so that the errors can be addressed in the failure report, and the failure report can be minimally edited before being fed back into the migration script as new source data.

It is important to note: because an entire transaction will fail if just one of the items within the transaction fails, the number of failures can balloon and not be representative of the number of errors present. The number of errors can be up to 25x less than the number of failures.

An example running of 21956 rows resulted in a failure report of 316 rows. Of these 316 rows, 70 failures were owing to missing park ORCS, leaving 246 failed rows due to transaction failures. After reviewing the failed rows, 11 duplicate rows were found, or roughly 246/25.

REMOVING LEGACY DATA

If the migration runs into unexpected complications, the script `purgeLegacy.js` is included to remove all legacy-tagged items in the database.

OPTIONAL TEST DATA:

A spreadsheet containing a test row is included. Swapping out the filename to this spreadsheet will add in a sample legacy park, legacy subarea, and a fully populated legacy record for each activity when the migration is run. Test Legacy items are identified with ‘HIST’ (HISTorical).

Legacy park: {pk: ‘park’, sk: ‘HIST’}

Legacy subarea: {pk: ‘park::HIST’, sk: ‘HIST’}

Legacy records: {pk: ‘HIST::Frontcountry Camping’, sk: ‘202303’},

{pk: ‘HIST::Frontcountry Cabins’, sk: ‘202303’}

{pk: ‘HIST::Backcountry Camping’, sk: ‘202303’}

{pk: ‘HIST::Backcountry Cabins’, sk: ‘202303’}

{pk: ‘HIST::Group Camping’, sk: ‘202303’}

{pk: ‘HIST::Day Use’, sk: ‘202303’}

{pk: ‘HIST::Boating’, sk: ‘202303’}

{pk: ‘HIST::Legacy Data’, sk: ‘202303’}