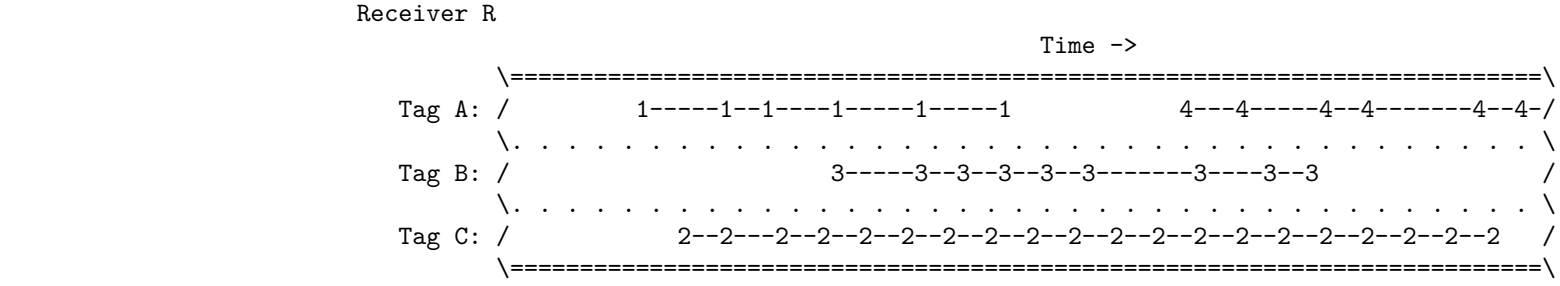
Bird's Eye View of Motus Data

What data look like

Here's a segment of data from a receiver (with a single antenna):



- time increases from left (earlier) to right (later)
- horizontal *lanes*, separated by . . . correspond to individual tags, labelled at left
- **hits** (detections) of a tag are plotted as single digits within its lane
- for Lotek coded ID tags, individual hits are not sufficient to determine which tag is being detected, because multiple tags transmit the same ID. However, the period (spacing between consecutive transmissions) is precise, and differs among tags sharing the same ID.
- so the fundamental unit of detection for Lotek coded ID tags is the **run**, or sequence of hits of a given ID, with spacing **compatible** with the value for that tag. i.e. two hits might differ by the tag's period, or by twice its period, or three times its period, ..., up to a limit required by drift between clocks on the transmit and receive sides.
- for Cornell ID tags, there are 2^32 unique codes, but:
  - there is a significant false positive rate, meaning some "hits" are not from tags
  - there is a significant bit error rate, meaning some "hits" are from tags whose actual code differs by a few bits from the code detected.
  - there is no fixed period for a tag, as transmission depends on accumulation of sufficient energy from the photovoltaic cell.
- so for Cornell ID tags, the fundamental unit of detection is again the run, but assembled from hits based on low inter-code Manhattan distance (i.e. # of bits difference), rather than time gaps.

- in the diagram above, runs are labelled by digits. So all hits of tag A above are either in run 1 or in run 4.
- for Lotek ID tags, a run can have gaps (missing hits) up to a certain size. Beyond that size, measurement error and clock drift are large enough that we can't be sure that the next detection of the same ID code is after a compatible time interval. So for tag A above, we're not sure the gap between the last detection in run 1 and the first detection in run 4 is really compatible with tag A, so run 1 is ended and a new run is begun. We could link runs 1 and 4 post-hoc, once we saw that run 4 was being built with gaps compatible with tag A, but at present, the run-building algorithm doesn't backtrack.)

Complication: data are processed in batches

The picture above is complicated by several facts:

- receivers are often deployed to isolated areas so that we can only obtain raw data from them occasionally
- receivers are not aware of the full set of currently-active tags
- sensorgnome receivers do not "know" the full Lotek code set; they record pulses thought to be from Lotek coded ID tags, but are only able to assemble them into tag hits for a small set of tags for which they have an on-board database, built from the user's own recordings of their tags. This limitation is due to restrictions in the agreement between Lotek and Acadia University for our use of their codeset.
- Lotek receivers report each detected tagID independently, and do not assemble them into runs. This means a raw Lotek .DTA file does not distinguish between tag 123:4.5 and tag 123:9.1 (i.e. between tags with ID 123 and burst intervals 4.5 seconds and 9.1 seconds).

It follows that:

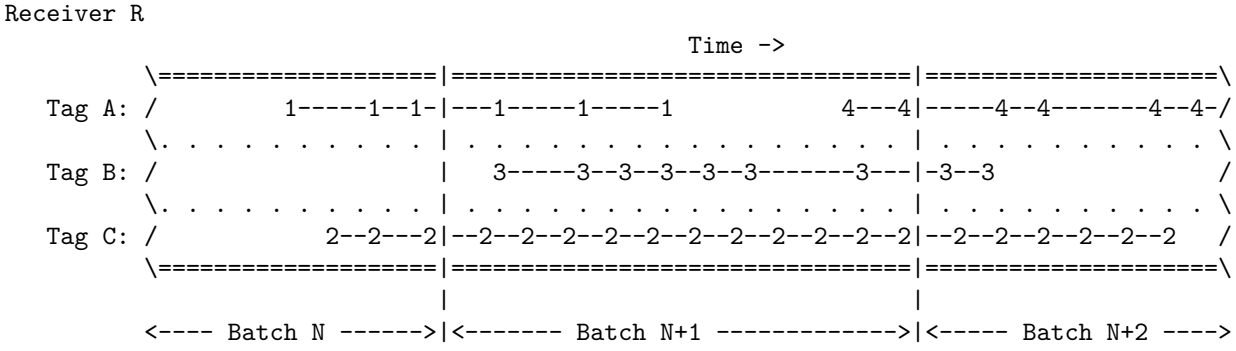
- raw receiver data must be processed on a server with full knowledge of:
  - which tags are deployed and likely active
  - the Lotek codeset(s)
- raw data should be processed in incremental batches
- processed data should be distributed to users in incremental batches, especially if they wish to obtain results "as they arrive", rather than in one lump after all their tags have expired.

Batches

A **batch** is the result of processing one collection of raw data files from a receiver. Batches are not inherent in the data, but instead reflect how data were processed. Batches arise in these ways:

- a user visits an isolated receiver, downloads raw data files from it, then uploads the files to motus.org once they are back from the field
- a receiver connected via WiFi, ethernet, or cell modem is polled for new data files; this typically happens every hour, with random jitter to smooth the processing load on the motus server
- an archive of data files from a receiver is re-processed on the motus server, because important metadata have changed (e.g. new or changed tag deployment records), or because a significant change has been made to processing algorithms.

Batches are artificial divisions in the data stream, so runs of hits will often cross batch boundaries. Adding this complication to the picture above gives this:



Receiver Reboots

A receiver **reboots** when it is powered down and then (possibly much later) powered back up. Reboots often correspond to a receiver:

- being redeployed
- having its software updated
- or having a change made to its attached radios,

so motus treats receiver reboots in a special way:

- a reboot always begins a new batch; i.e. batches never extend across reboots. This simplifies determination of data ownership. For example, all data in a boot session (time period between consecutive reboots) are deemed to belong to the same motus project. This reflects the fact that a receiver is (almost?) always turned o between the time it is deployed by one project, and the time it is redeployed by another project.
- any active tag runs are ended when a receiver reboots. Even if the same tag is present and broadcasting, and even if the reboot takes only a few minutes, hits of a tag before and after the reboot will belong to separate runs. This is partly for convenience in determining data ownership, as mentioned above. It is also necessary because sometimes receiver clocks are not properly set by the GPS after a reboot, and so the timestamps for that boot session will revert to a machine default, e.g. 1 Jan 2000. Although runs from these boot sessions could in principle be re-assembled post hoc if the system clock can be pinned from information in field notes, this is not done automatially at present.
- parameters to the tag-finding algorithm are set on a per-batch basis. At some field sites, we want to allow more lenient filtering because there is very little radio noise. At other sites, filtering should be more strict, because there is considerable noise and high false-positive rates for tags. motus allows projects to set parameter overrides for individual receivers, and these overrides are applied by boot session, because redeployments (always?) cause a reboot.
- when reprocessing data (see below) from an archive of data files, each boot session is processed as a batch.

Incremental Distribution of Data

The motusClient R package allows users to build a local copy of the database of all their tags' (or receivers') hits incrementally. A user can regularly call the `tagme()` function to obtain any new hits of their tags. Because data are processed in batches, `tagme()` either does nothing, or downloads one or more new batches of data into the user's local DB.

Each new batch corresponds to a set of files processed from a single receiver. A batch record includes these items: - receiver device ID - how many of hits of their tags occurred in the batch - first and last timestamp of the raw data processed in this batch

Each new batch downloaded will include hits of one or more of the users's tags (or someone's tags, if the batch is for a "receiver" database).

A new batch might also include some GPS fixes, so that the user knows where the receiver was when the tags were detected.

A new batch will include information about runs. This information comes in three versions:

- information about a new run; i.e. one that begins in this batch
- information about a continuing run; i.e. a run that began in a previous batch, has some hits in this batch, and is not known to have ended
- information about an ending run; i.e. a run that began in a previous batch, might have some hits in this batch, but which is also known to end in this batch (because a sufficiently long time has elapsed since the last detection of its tag)

Although the unique `runID` identifier for a run doesn't change when the user calls `tagme()`, the number of hits in that run and its status (`done` or not), might change.

Reprocessing Data

motus will occasionally need to reprocess raw files from receivers. There are several reasons:

- new or modified tag deployment records. The tag detection code relies on knowing the active life of each tag it looks for, to control rates of false positive and false negative hits. If the deployment record for a tag only reaches the server after it has already processed raw files overlapping the tag's deployment, then those files will need to be reprocessed in order to (potentially) find the tag therein. Similary, if a tag was mistakenly sought during a period when it was not deployed, it will have "used up" signals that could instead have come from other tags, thereby causing both its own false positives, and false negatives on other tags. (This is only true for Lotek ID tags; Cornell should be unaffected, provided deployed tags are well dispersed in the ID codespace.)
- bug fixes or improvements in the tag finding algorithm
- corrections of mis-filed data from receivers. Sometimes, duplication among receiver serial numbers (a rare event) is only noticed $after$ data from them has already been processed. Those data will likely have to be reprocessed so that hits are assigned to the correct station. Interleaved data from two receivers having the same serial number will typically prevent hits from at least one of them, as the tag finder ignores data where the clock seems to have jumped backwards significantly.

The (eventual) Reprocessing Contract

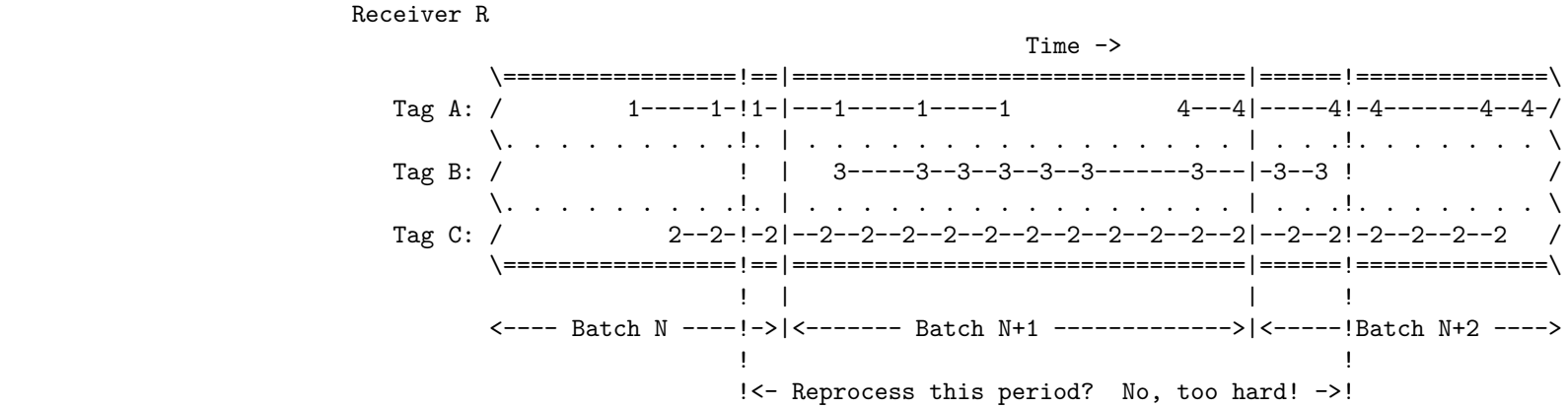
Reprocessing can be very disruptive from the user's point of view ("What happened to my hits?"), so motus reprocessing will be:

1. transparent: motus will maintain a record of what was reprocessed, why, when, what was done differently, and what changed
2. optional: users should be able to choose whether they wish to update their copy of the database
3. all-or-nothing: users should be guaranteed that they either have the reprocessed data, or the original data, but not an undefined mix of the two.
4. reversable: users should be able to revert to any previous version of their database.

Initially, motus data processing might not adhere to this contract, but it is an eventual goal.

Reprocessing simplified: only by boot session

The most general reprocessing scenario could look like this:



if raw data records from an arbitrary stretch of time could be reprocessed. However, this is complicated because runs like `1 2`, and `4` above might lose or gain hits within the reprocessing period, but not outside of it. This might even break an existing run into distinct new runs.

This situation is challenging (NB: not impossible; might be a TODO) to represent in the database if we want to maintain a full history of processing. For example,

if reprocessing deletes some hits from run 2, how do we represent both the old and the new versions of that run?

The complications arise due to runs crossing the reprocessing period boundaries, so for simplicity we should *choose a reprocessing period that* ***no*** *runs cross*. Currently, that means a boot session, as discussed above.

Distributing reprocessed data

The previous section shows why we only reprocess data by boot session. Given that, how do we fulfill the reprocessing contract?

Note that a reprocessed boot session will fully replace one or more existing batches and one or more runs, because batches and runs both nest within boot sessions.

So a reprocessing event can be represented as a `changeset` with these items:

- `replacedBatchIDs`: IDs of batches to be replaced. Records for these batches, as well as for runs/hits overlapping/contained by them will also be replaced. GPS records will also be retired.
- `newBatches`: records for new batches; these will have new batchIDs
- `newRuns`: records for new runs; these will have new runIDs
- `newHits`: records for new hits; these will have new hitIDs
- `newGPS`: records for new GPS fixes

In keeping with the limited size of queries to the motus data server, the `newXXX` items will be obtained incrementally by the motusClient package.