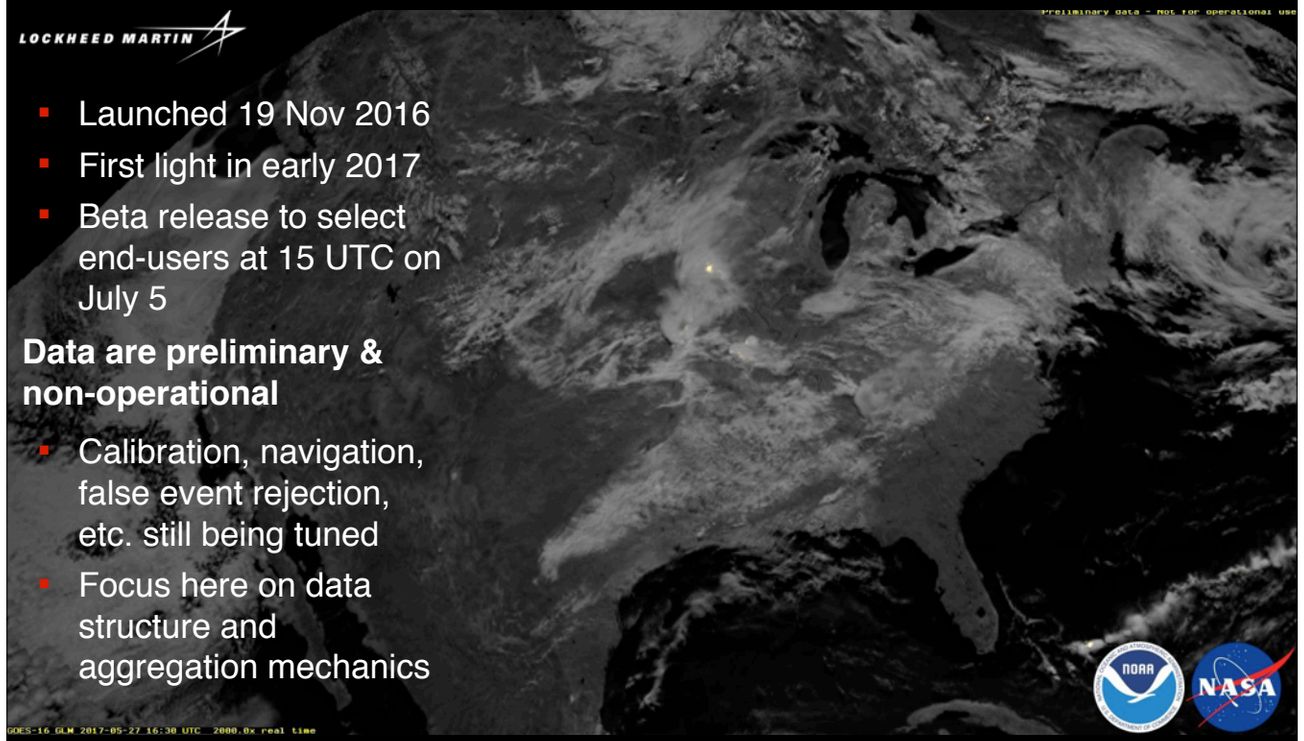
Good morning

I'm an observational atmospheric scientist specializing in lightning, and one of my current projects is to use ground-based data from a lightning mapping array we operate in West Texas, as seen with my research group in the lower right corner, to validate the first-ever lightning mapping instrument in geostationary orbit.

I'm going to focus on GLM data as an example of the kinds of data that lightning instruments produce.

GEOSTATIONARY LIGHTNING MAPPER (GLM) DATA: A DISCLAIMER



- Launched 19 Nov 2016
- First light in early 2017
- Beta release to select end-users at 15 UTC on July 5

Data are preliminary & non-operational

- Calibration, navigation, false event rejection, etc. still being tuned
- Focus here on data structure and aggregation mechanics

27 May 2017; source: goes-r.gov

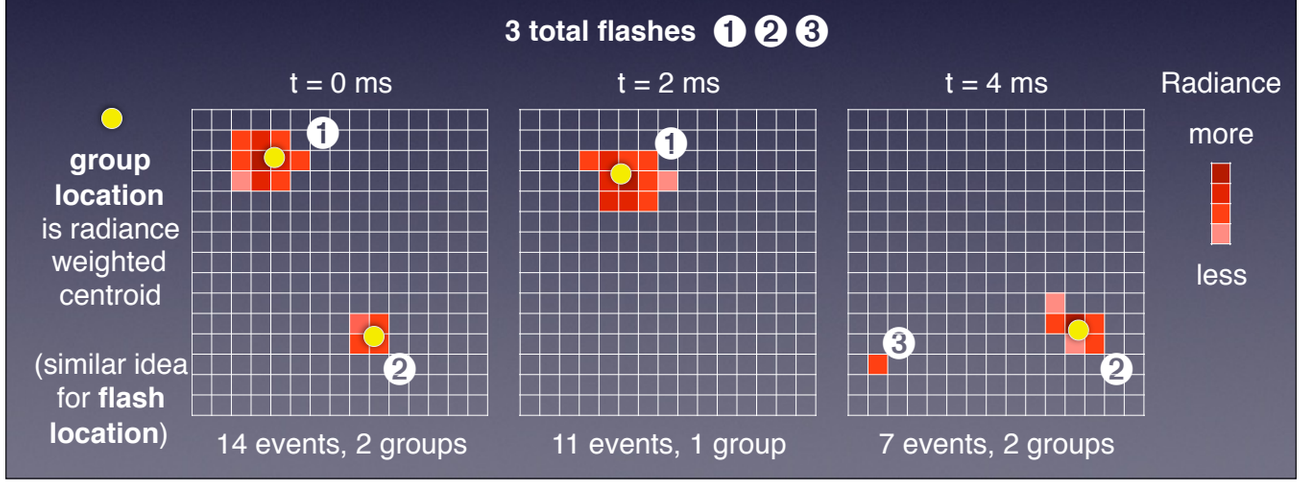
Before I get started, a disclaimer: the GLM data I'm about to show are preliminary and non-operational, and probably some of the first low-level data to be publicly shown. Tuning will continue through this year.

The loop in the background is some of the earliest public imagery, here showing thunderstorms over the midwestern United States. The flickering bright spots on the cloud background are optical pules made by lightning, sped up 2000x real time.

This talk is focused on the structure of the GLM data, and how we can aggregate them to imagery like you see here.

Space-based lightning sensing:
Detects optical pulses from lightning at 500 fps
(e.g., GOES-R GLM, 8 km nominal pixel)

- **Events**: triggered pixels above background threshold
- **Groups**: adjacent pixels in a single frame – ‘strokes’
- **Flashes**: collection of groups close in space/time — all strokes along connected channels



This slide describes the detection methodology and the three levels of hierarchical data produced by GLM and similar space-based sensors that have flown in low earth orbit.

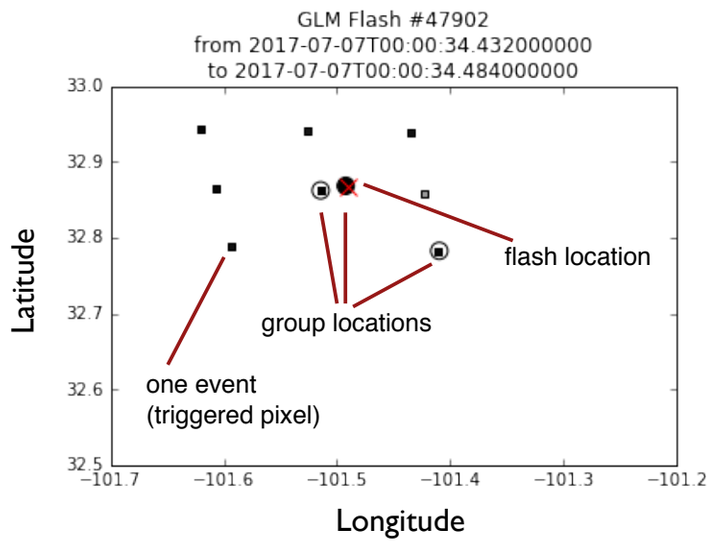
There is a pixel grid which is read out at 500 fps and the locations of indvidual pixels above a tracked background scene are reported. These pixel triggers are called events. In each frame, clusters of adjacent pixels are known as groups, and their radiance-weighted centroid is reported.

In the first frame, we have 14 events, and two groups, then 11 events and 1 group, followed by 7 events and two groups.

The final entity reported is a “flash”, which is a collection of groups close enough in space and time. The flash is meant to encompass all strokes that take place along a single connected lightning channel. Here, there are three flashes, as you see labeled.

One GLM flash
3 groups, 9 events, total duration 52 ms

Data from a relatively low flash rate storm: O(10) flashes per minute

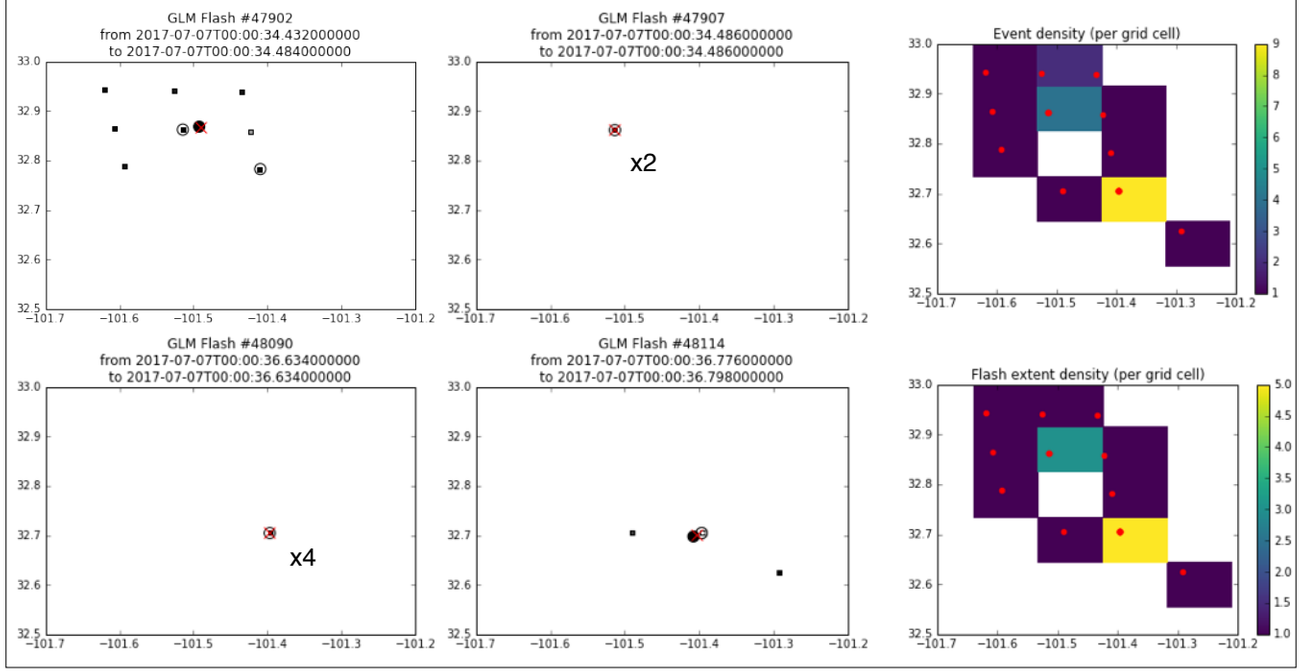


Thanks to Ryan May for access to test data from the Unidata GRB/S3 feed

Here is one GLM flash, which contains 3 groups and 9 events. For GLM, the events, groups, and flashes, each have an associated latitude and longitude. You can see that these eight pixels lit up at least once in three separate frames, with the group location moving around, but with the overall radiance weighted centroid of the flash somewhere near the center of the pixels that lit up.

Imagery (a heatmap) is easier to digest than points, so count up events and flashes on a grid. Example: 8 flashes in 20 seconds (10,000 frames)

- *Event density: how many pixels detected light in each grid cell?*
- *Flash extent density: how many flashes produced light in each grid cell?*



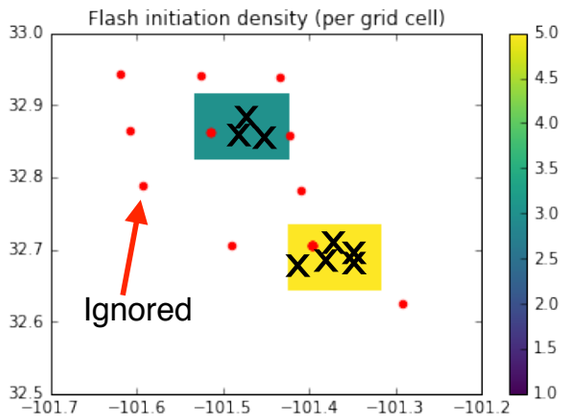
Over the course of 20 seconds, this storm produced eight flashes: the one you've already seen, plus six single-pixel flashes, and another one where three pixels lit up.

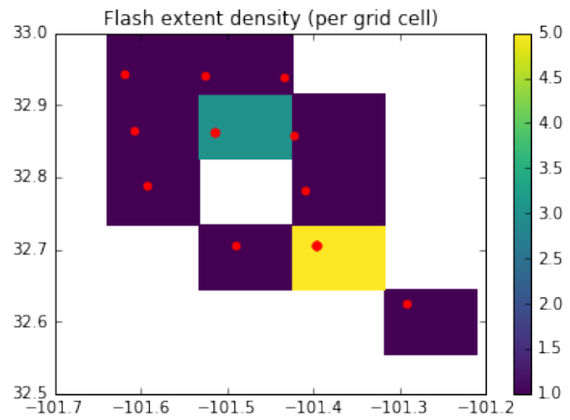
It's easier to see what's going on with the flashes over 20 seconds if we define a new grid, independent of the pixels, and count up the events and flashes. In the upper right is event density, which says how many GLM pixels were triggred. Most grid cells have only one trigger, except in the top center, where two pixels fall in the same grid cell, and at the center of the two largest flashes, where 4 and 9 events are counted.

We use the event and flash information to count how many flashes produced light in each grid cell. This is called flash extent density, and is a kind of local flash rate. It avoids some of the double-counting issues we see in the event density grid.

IMAGERY USING THE EVENT-FLASH LINKAGE IS EASIER TO DIGEST THAN IMAGERY MADE FROM FLASH POINTS ALONE

- If we ignore the spatial extent of each flash and use only the flash center locations (individual points), imagery becomes very sparse (left).
- Flash extent density (right) captures much more information, such as the width of the storm.





What happens if we ignore the events, and only use the flash centroid locations? The images we get are much sparser, and lose some of the spatial extent information that's helpful in understanding the data as storm cells. So flash extent density is probably the most useful one-image summary of all of the data that GLM collects.

Flash extent density calculation in practice

- Convert arbitrary event location (`x, y`) to grid locations (`x_i, y_i`)
- Replicate the `flash_id` for each event
- Find the unique `(x_i, y_i, flash_id)` vectors
  - *This row is eliminated*
- Find flash counts with `histogramdd`
- This logic extends to event coordinates in N dimensions

event		flash
x_i	y_i	id
1	1	1
6	8	2
1	1	1
3	4	1

```
def extent_density(x, y, ids, x0, y0, dx, dy, xedge, yedge):
    x_i = np.floor( (x-x0)/dx ).astype('int32')
    y_i = np.floor( (y-y0)/dy ).astype('int32')
    unq_idx = unique_vectors(x_i, y_i, ids)
    count, edges = np.histogramdd((x[unq_idx],y[unq_idx]), bins=(xedge,yedge))
    return count, edges
```

In practice, how can we calculate flash extent density? It's a relatlivey short function. We first convert the arbitrary positions of events to grid coordinates, and we also need an ID vector that give the flash ID for each event. We then use a unique-vector function to eliminate duplicate (x,y,id) vectors, and get the flash counts using histogramdd.

In this small example, we see that the third row is eliminated, because (1,1,1) appears twice, and we only want to light up a grid cell once for each flash.

LIGHTNING DATA ARE ODD

- Two kinds of atmospheric science datasets:
  - *Georeferenced, regularly gridded data - “imagery”*
  - *Measurements at arbitrary point locations / times, or time series*
- These models are accompanied by mature data formats, visualization approaches, and data services.
  - *e.g., NetCDF/CDM files following the Climate and Forecast metadata standards*
- Lightning data are best represented by a data model that is neither as structureless as points nor as rigid as grids
  - *This data model does not exist in the CF standards.*
  - *It is a foreign key relationship that is more frequently encountered in database schemas.*
  - *A sort of virtual coordinate that links hierarchical set membership?*
- Let’s try to use `xarray` to traverse the hierarchical GLM data

The way we have to deal with lightning data as described on the last slide is a bit odd in the atmospheric sciences. It’s common to deal with regularly gridded images, as well as scattered measurements at arbitrary places/times. Each of those are common enough that they are encoded in the NetCDF / Common Data Model and Climate and Forecast (CF) metadata standards.

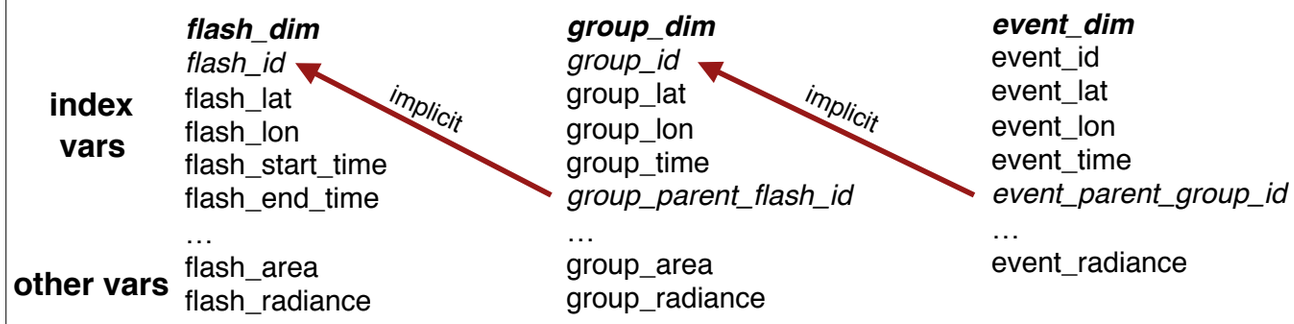
However, lightning data are somewhere in between - they are neither as structureless as points nor as rigid as grids. A model for that isn’t really embodided in metadata standars, but is familiar from database schemas - it’s a foreign key relationship.

Maybe the standards bodies can consider adding some convention for encoding this information - something like a virtual dimension that encodes hierarchical set membership.

Anyway, what we’ll do next is try to use xarray, a dimension- and standards-aware library, to traverse the hierarchical GLM data.

GLM DATA STORAGE MODEL
GOES-R.GOV, PRODUCT DEFINITION AND USER'S GUIDE
VOL. 5: LEVEL 2 PRODUCTS

- NetCDF files with good metadata
  - *Climate and Forecast (CF) conventions where possible*
- *Stored as a CF `point featureType`*
  - *The only possibility for arbitrary locations and times*
  - *The CF spec includes an "indexed ragged array," but it's only a partial solution because it describes connections only to one other dimension, and only implicitly*
- Common keys specify linkages across the hierarchy, but tools don't automate their use
  - *How do I get the `flash_id` for each event?*
  - *How many events were in this flash?*



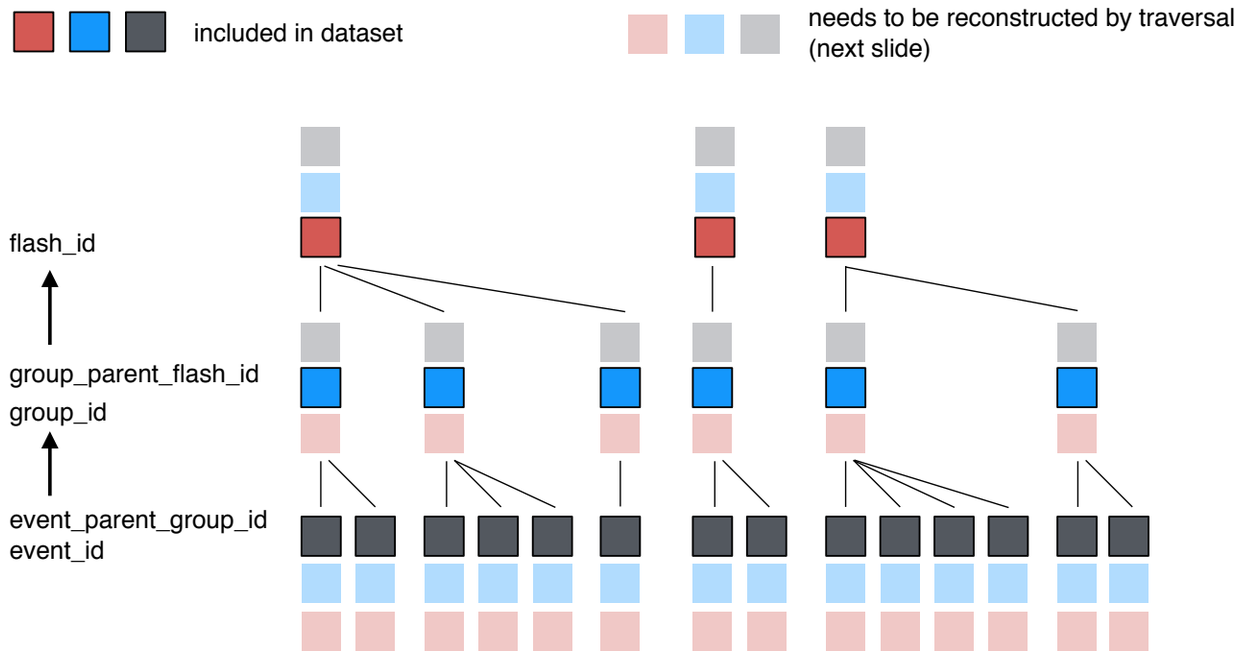
The GLM data storage model is actually pretty good - the metadata is CF compliant.

As it's the only thing available, the data are stored as scattered point data, and use an idea from the CF spec called an indexed ragged array. It's a partial solution.

You can see in the event dimension we have an event_id as well as an event_parent_group_id, which gives the group to which the event belongs. That serves as a link up the hierarchy, but it's not something xarray (or the standards) natively understand.

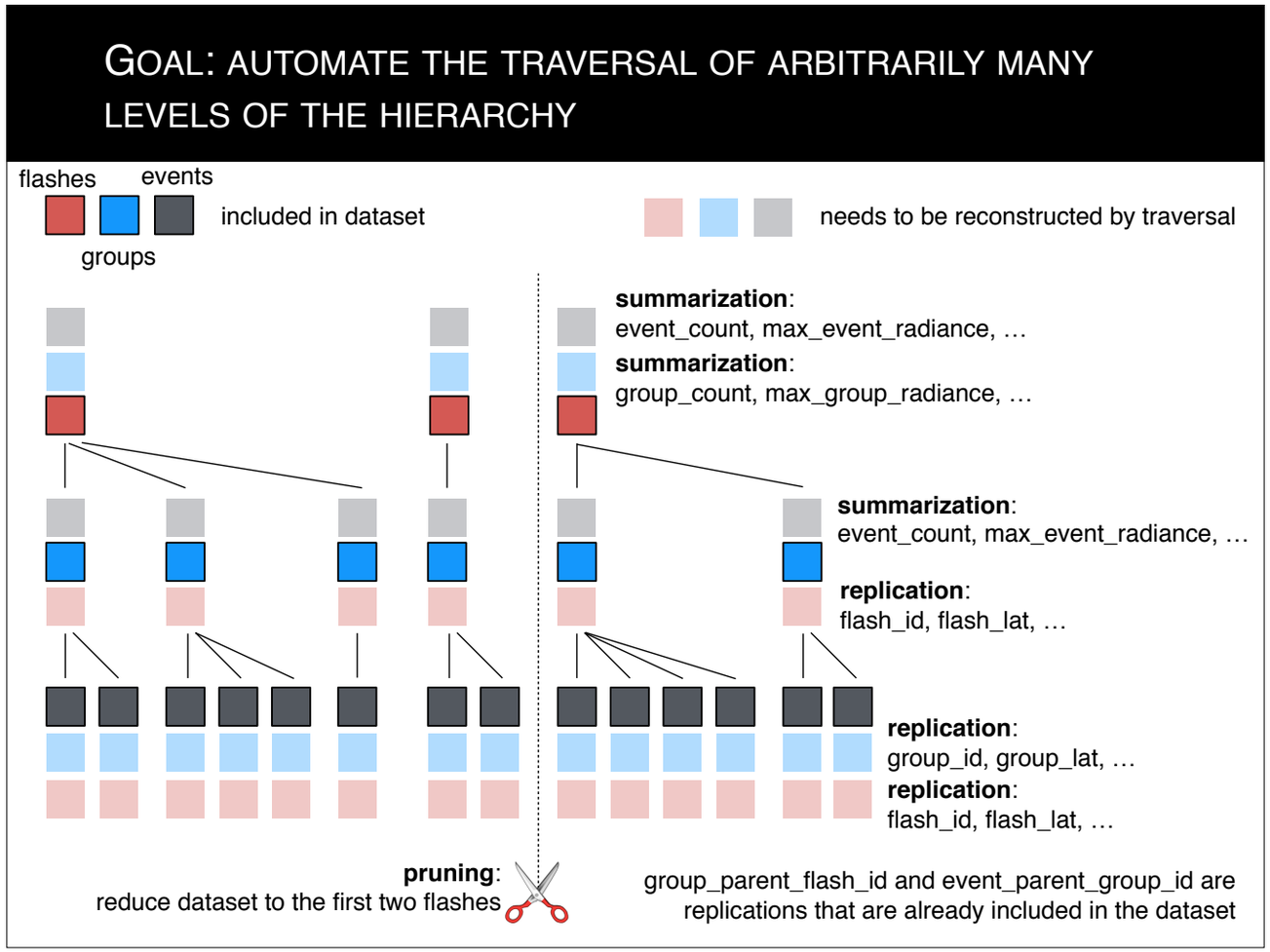
This makes some useful queries difficult: how do I know how many events were in a flash? How do I get the flash ID for each event so I can calculate flash extent density?

HIERARCHICAL LINKAGE AMONG DIMENSIONS



Here's a visual depiction of the dataset and some things we want to do. There are three levels in the hierarchy, each of which has its own dimension in the dataset. It is shown in the bold colors, with flashes in red, groups in blue, and events in black.

At each level, we have IDs and parent ID links to the next level up. What we lack is, for example, access to the replicated flash ids at the event level, as represented by the bottom pink row.



What we want is to be able to automate the replication of information down the tree, and summarization of info up the tree. For instance, we might want to count events, or replicate the flash_id along the event dimension. The idea even extends to other calculations, such as maximum radiance, or the flash latitude and longitude that corresponds to each event.

A final operation that we might want to do is prune out flashes of interest, reducing the size of the dataset.

SOLUTION: A ONETOMANYTRAVERSAL CLASS

- User only needs to specify the entity IDs and parent-child links (in order, descending from parent to child)

```
entity_id_vars = ['flash_id', 'group_id', 'event_id']
parent_id_vars = [ # flash_id has no parent
                   'group_parent_flash_id',
                   'event_parent_group_id']
traversal = OneToManyTraversal(dataset, entity_id_vars,
                               parent_id_vars)
```

- Traverse and index hierarchical `xarray` datasets
  - *Other variables are along for the ride when indexing along a dimension*
- Automatically sets up two `Dataset.groupby` operations for each dimension
  - *e.g.,* `group_id` *and* `group_parent_flash_id`

It turns out we can solve all of these problems in a standardized way with a One to many traversal class. The user specifies, in order, the hierarchy IDs and the parent relationship foreign keys, and then the queries we need are handled in a standard way – so this works for any dataset.

Because we're working with xarray datasets, when we do the traversal and indexing, other variables come along for the ride.

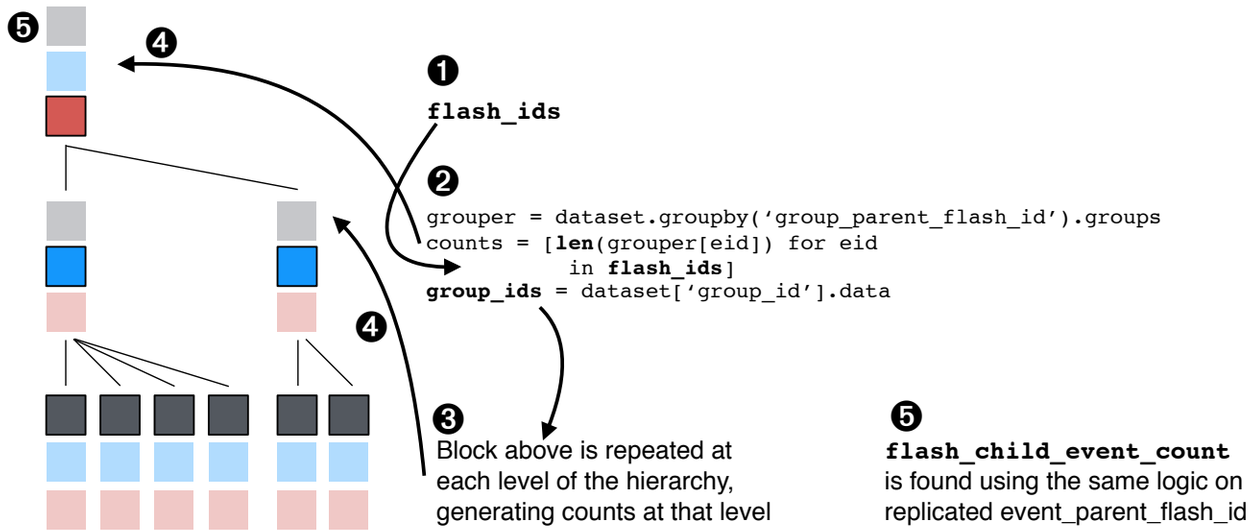
The essential operations are facilitated by groupbys that are set up on the ID and parent ID variables along each dimension.

SUMMARIZATION USING ONETOMANYTRAVERSAL

User code

```
all_counts = traversal.count_children('flash_id', 'event_id')
flash_child_group_count = xr.DataArray(all_counts[0], dims=['flash_dim',])
group_child_event_count = xr.DataArray(all_counts[1], dims=['group_dim',])
dataset['flash_child_group_count'] = flash_child_group_count
dataset['group_child_event_count'] = group_child_event_count
```

Under the hood *Descend the hierarchy*, in this case counting children



Here's an example of descending the hierarchy to count children.

The user code requests a traversal from the flash_id to event_id, and returned are flash_child_group_count and group_child_event_count as the class descends the hierarchy.

In step 1, flash_ids are selected, and then used at the group level with the group_parent_flash_id groupby to get a count for each flash_id.

Then the group_ids are extracted, and the process is repeated at the next level down.

Those counts are then of the correct dimension to be assigned at the flash and group levels.

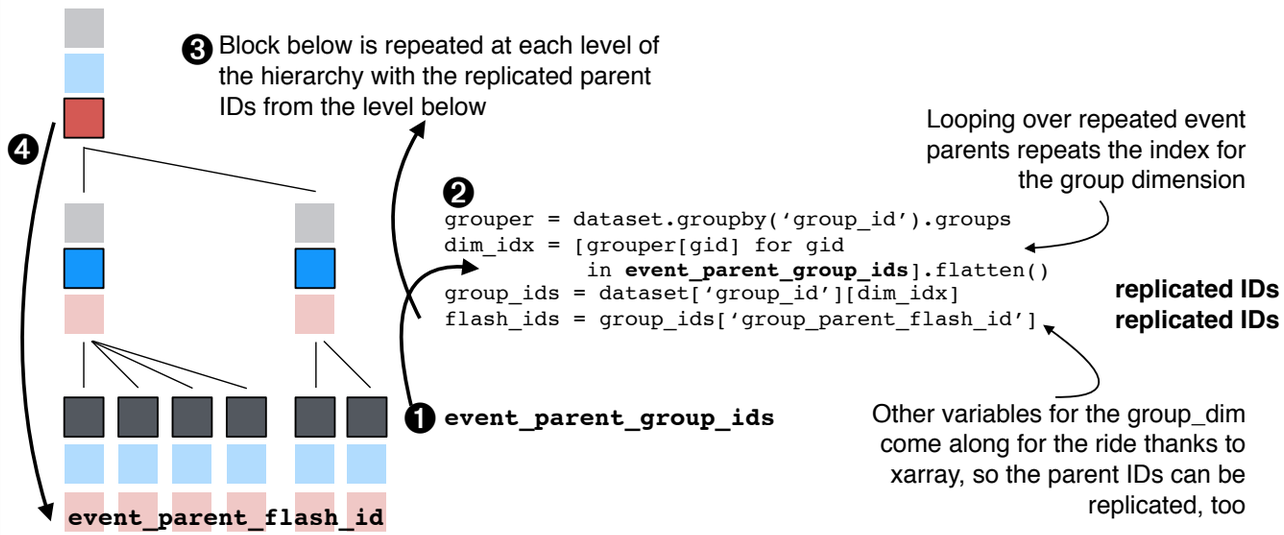
If we somehow can replicate the flash_ids down to the event_id level, we can use the same logic to get the flash_child_event_count.

REPLICATION USING ONETOMANYTRAVERSAL

User code

```
flash_ids = traversal.replicate_parent_ids('flash_id',
                                            'event_parent_group_id')
event_parent_flash_id = xr.DataArray(flash_ids,
                                     dims=['event_dim',])
dataset['event_parent_flash_id'] = event_parent_flash_id
```

Under the hood *Ascend the hierarchy*, replicating to match the event dimension



A replication of data from the flash level to the event level is performed from the bottom up, by ascending from the event_parent_group_id to the flash_id level.

The event_parent_group_ids are used in the group_id grouby, which results in replicated indexes along the group dimension. This in turn can be used to index the group_ids and group_parent_flash_ids, which replicates them to the size of the event dimension. The replicated group_parent_flash_ids are used at the flash_id level to get the replicated flash_ids.

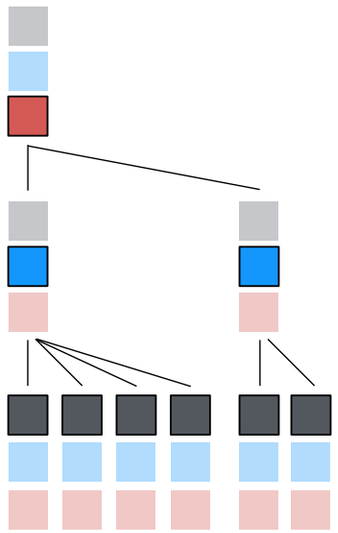
The process can be repeated up the hierarchy as many times as necessary.

PRUNING USING ONETOMANYTRAVERSAL

User code

```
flash_ids = [1,24,5,6]
reduced_dataset = traversal.reduce_to_entities('flash_id', flash_ids)
```

Under the hood *Descend the hierarchy*, building an index for each dim



❶

```
indexer = {}
grouper = dataset.groupby('flash_id').groups
dim_idx = [grouper[eid] for eid in flash_ids]
indexer['flash_dim'] = dim_idx
```

❷

```
grouper = dataset.groupby('group_parent_flash_id').groups
counts = [grouper[eids] for eid
          in flash_ids]
dataset = dataset[indexer]
group_ids = dataset['group_id'].data
indexer = {}
```

❸ Block above is repeated using reduced dataset and IDs of the level above

❹ Repeat the process for any entities above the flash level (if we had them) by ascending the hierarchy

The final thing we want to do is pruning, which proceeds here by using the flash_id groupby to index and select the flash_ids.

We then use those flash_ids at the group level to loop over the group_parent_flash_ids, building up an index of groups that match the

At this point we subset the dataset, and then get the new group_ids to be used with event_parent_group_id. The index/subset process is thereby repeated down the hierarchy.

If we had a level above flashes, we could also ascend the hierarchy, and eliminate the entities at that level who where not parents of any of the requested flashes.

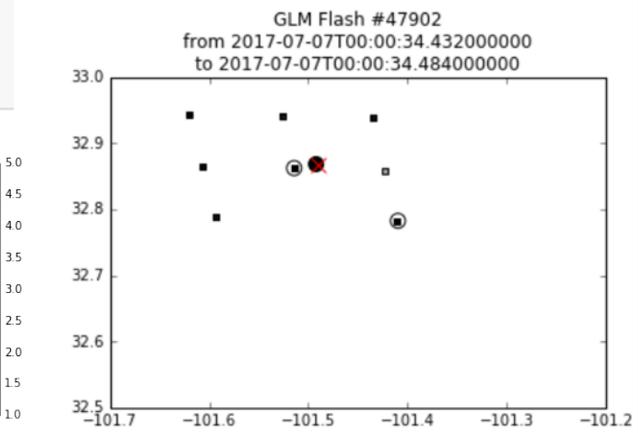
This gives us a reduced dataset.

Interactive use of hierarchical dataset pruning

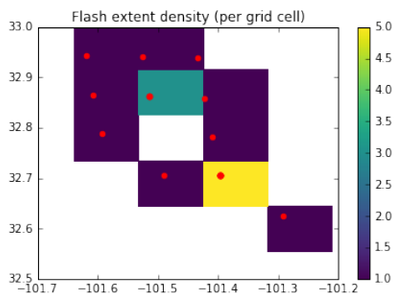
Interactively view flashes one at a time

```
from glmtools.plot.locations import plot_flash

# fl_id_vals = list(glm.dataset.flash_id.data)
flash_ids = [47902, 47907, 47912, 48090, 48096, 48098, 48112, 48114] # these are in West Texas
fl_id_vals = flash_ids
fl_id_vals.sort()
flash_slider =  widgets.SelectionSlider(
    description='Flash',
    options=fl_id_vals,
)

axis_range = (-101.7, -101.2, 32.5, 33)

def do_plot(flash_id):
    this_flash = glm.get_flashes([flash_id])
    fig = plot_flash(glm, flash_id)
    plt.axis(axis_range)
    plt.show()
interactor = widgets.interact(do_plot, flash_id=flash_slider)
```

Flash 47902



The flash extent density plots earlier in the talk were created using the flash ID replication process.



Reduced datasets are useful in exploring the data – including exploring flashes one at a time. We have used the pruning feature of the traversal code to make a nice interactive widget for flipping through flashes – or a subset of flashes –in the notebook using a slider widget.

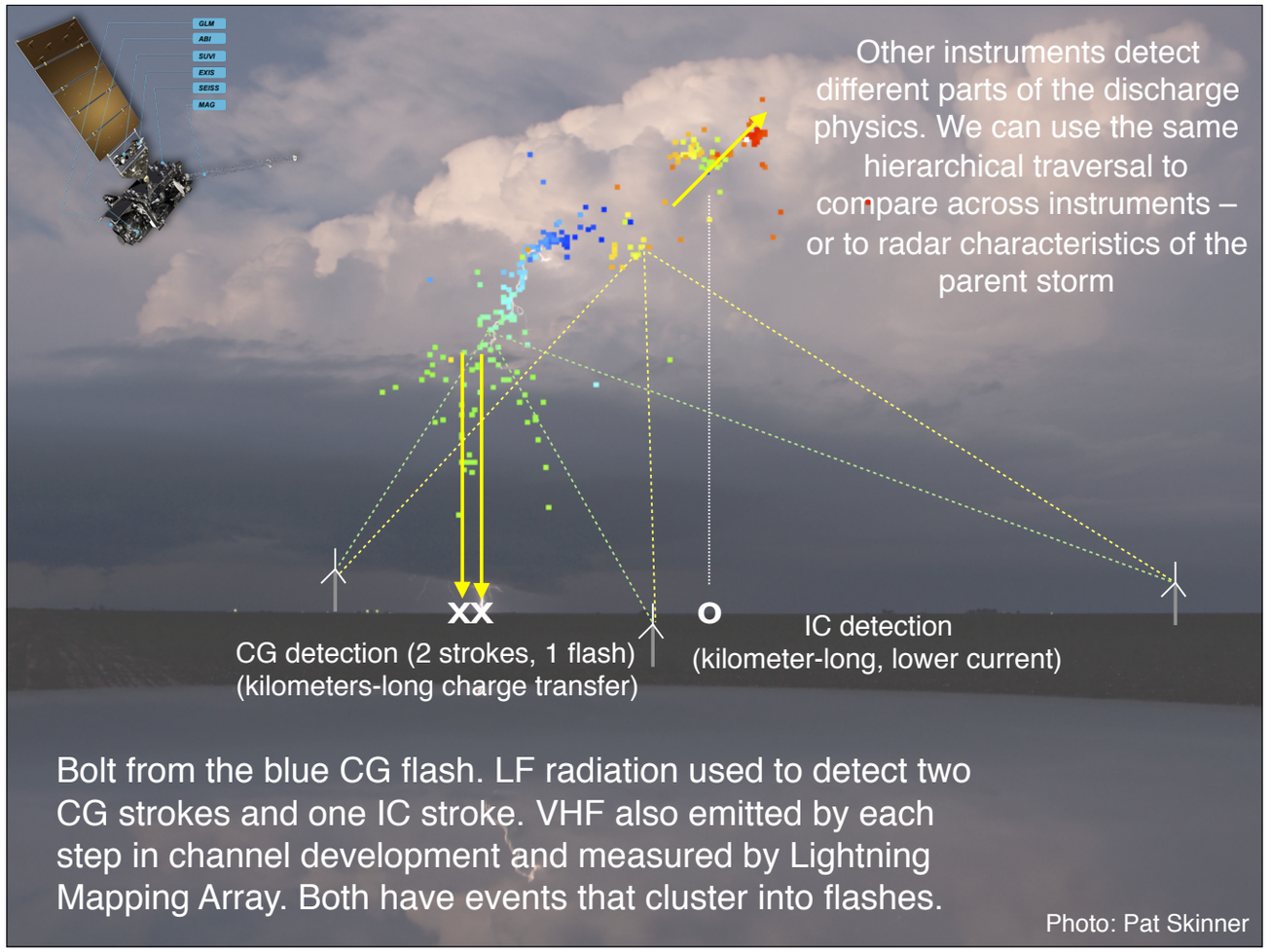
We also used the traversal class to create the flash extent density plots.

SUMMARY

- Lightning data are best represented by a data model that is neither as structureless as points nor as rigid as grids. It is a foreign key relationship that is more frequently encountered in database schemas.
- The foreign-key relationship can be specified in a human-friendly way, and traversed using `xarray`'s `groupby` and indexing logic
  - *CDM and CF standards don't support it, but this talk prototypes how they might, and why it matters.*
  - *Code later this afternoon at https://github.com/deeplycloudy/scipy2017*
- Our summarization, replication, and pruning needs are solved in a standardized way, and we can analyze and plot data without laborious looping.

so, in summary, lightning data are best represented by a foreign key relationship that sits somewhere between rigid grids and strictureless points. It is a foreign key relationship that we can traverse with xarray.

We have written a class that solves the traversal problem in general, and that lets us get about the business of doing science.



In fact, as we bring other instruments to the table during validaiton, they will see different parts of the discharge physics. In principle, we can use the same hierarchical traversal to compare across instruments – or even to radar characteristics of the parent storm.

In that spirit, I'll leave you with this final view of a variety of other detections we can get for this bolt from the blue flash, each of which has some degree of clustering of events into a larger flash entity.