TEXAS TECH UNIVERSITY™

From the Field to Geostationary Orbit: Mapping Lightning with Python

Eric C. Bruning

TTU Department of Geosciences

Atmospheric Science Group

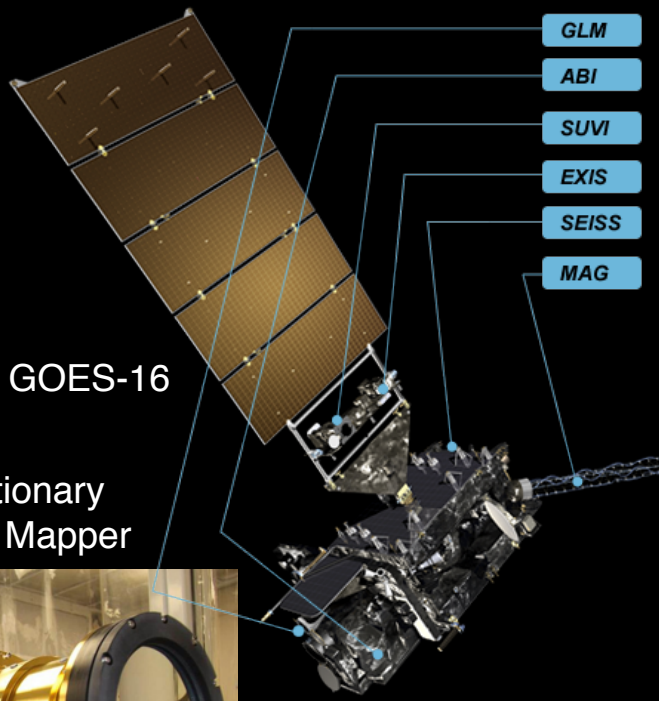
@deeplycloudy

SciPy, Austin, TX

11:30 AM, 12 July 2017

Earth, Ocean and Geo Science Track

This work supported by NSF-1352144 and NASA/ NOAA through the GOES-R GLM Validation program



GOES-16

Source: goes-r.gov

Geostationary Lightning Mapper

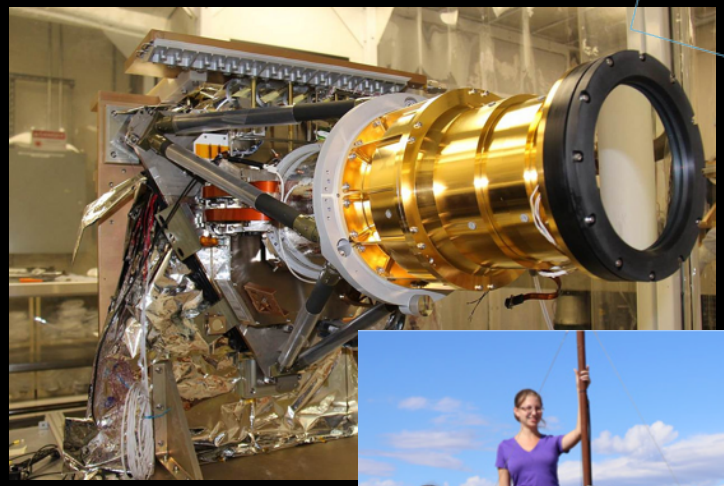
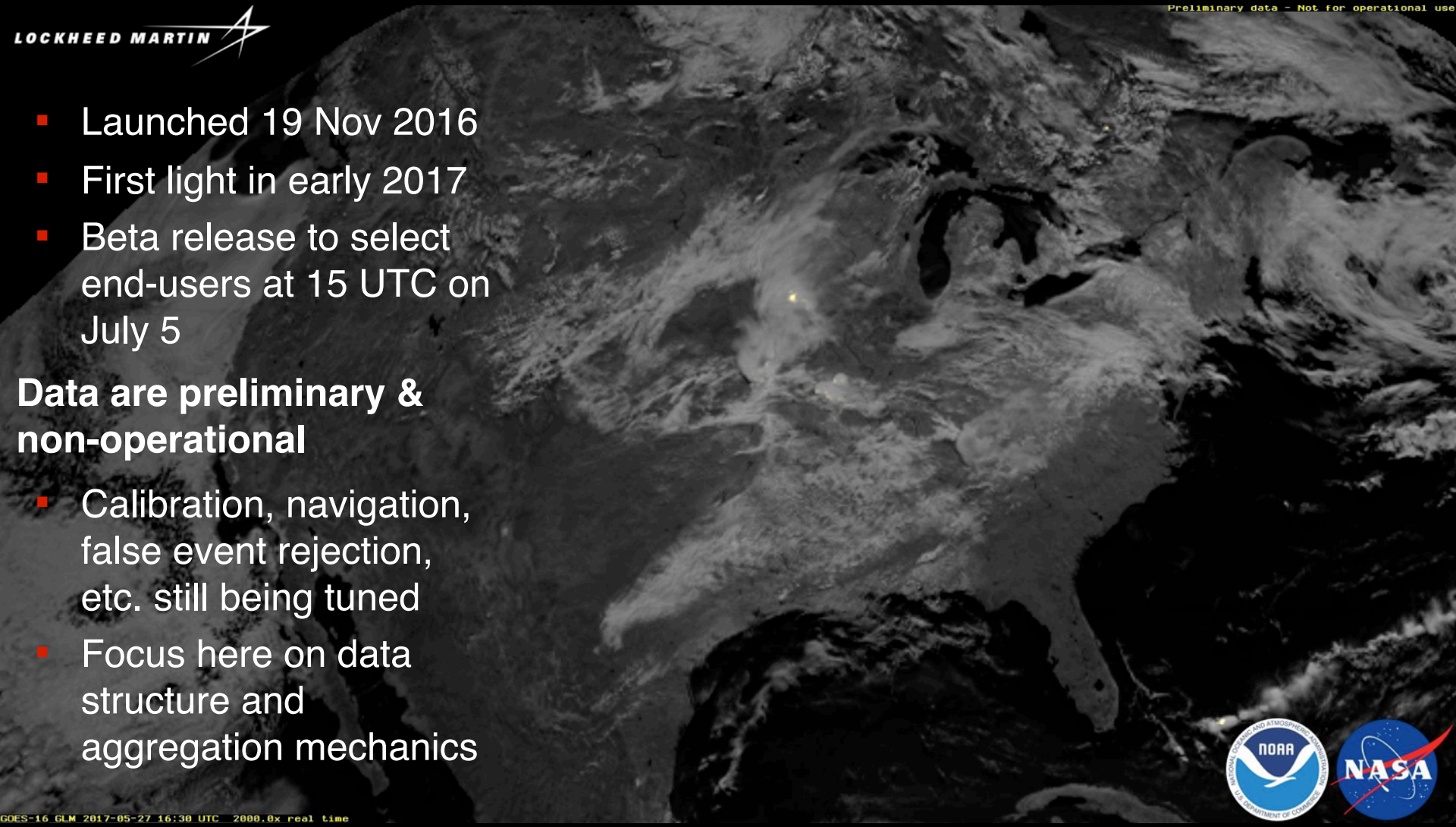


Photo: Tina Fuentes

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

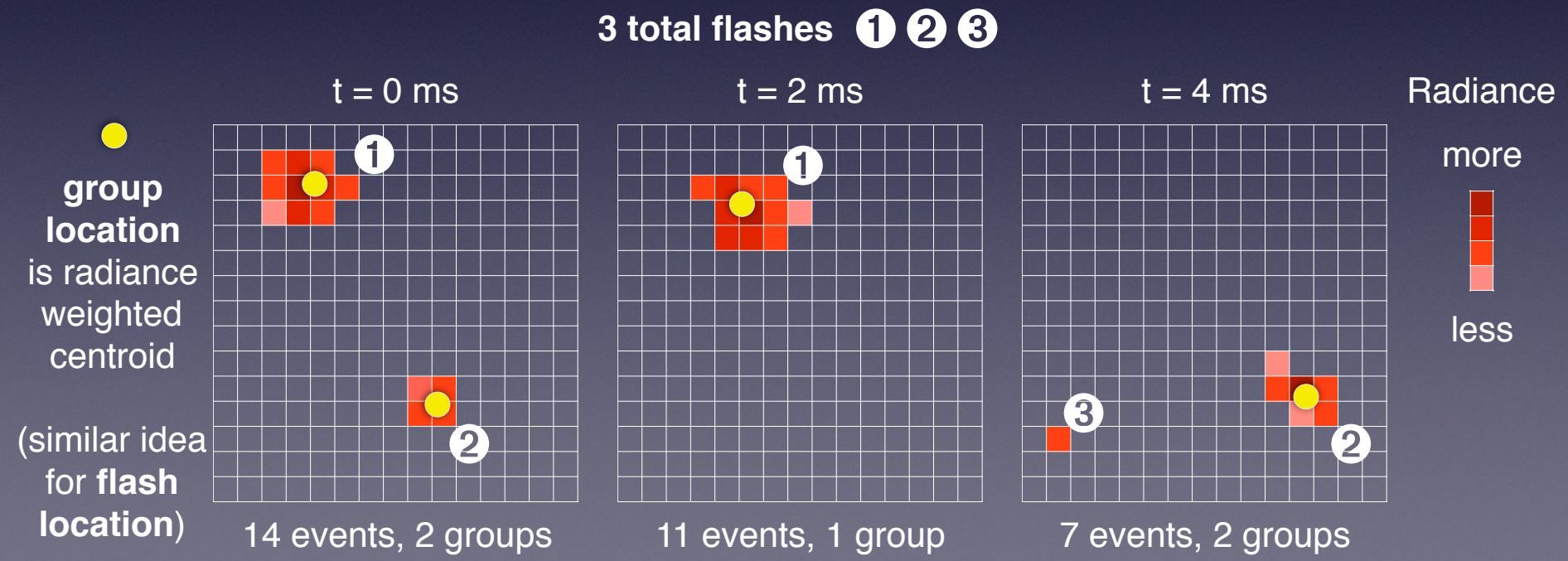
Data are preliminary & non-operational

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

27 May 2017; source: goes-r.gov

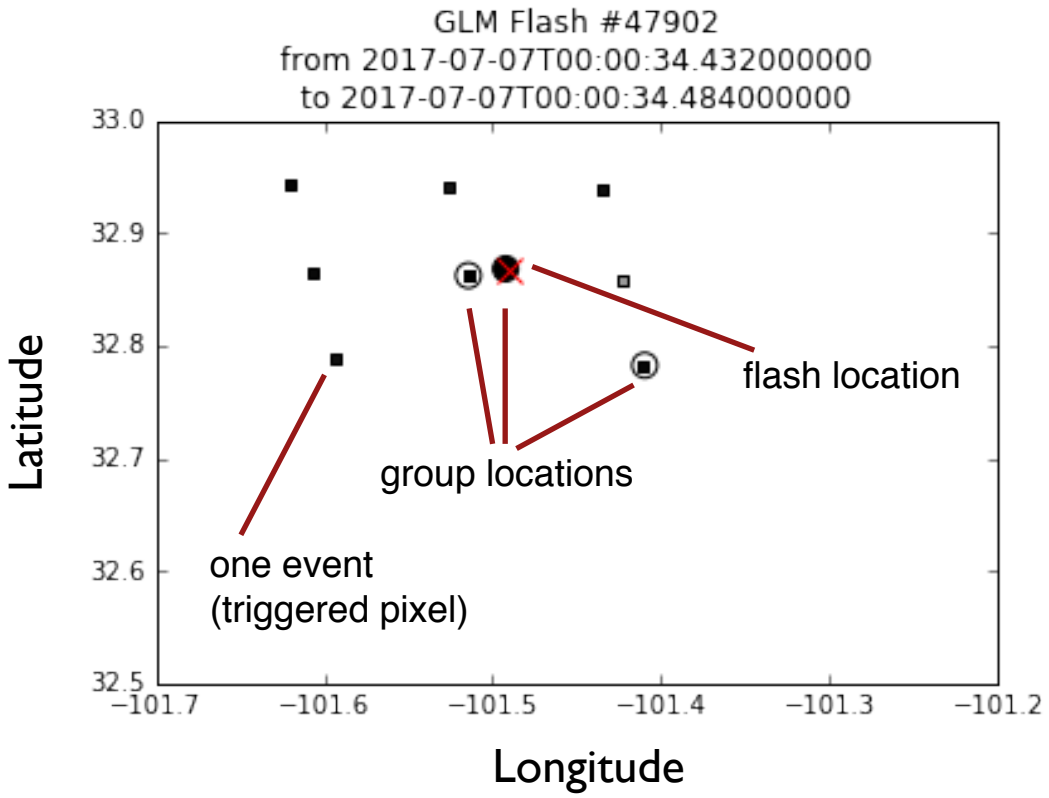
Space-based lightning sensing: Detects optical pulses from lightning at 500 fps
(e.g., GOES-R GLM, 8 km nominal pixel)

- **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



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

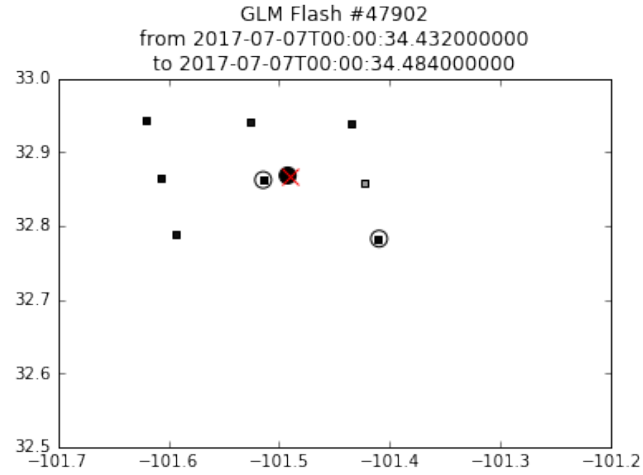
Data from a relatively low flash rate storm: O(10) flashes per minute

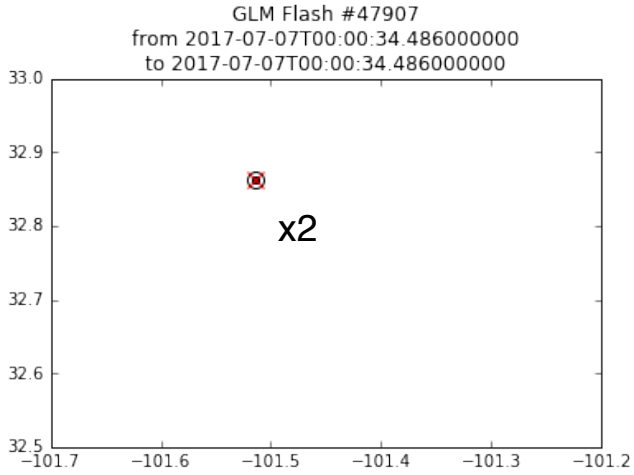


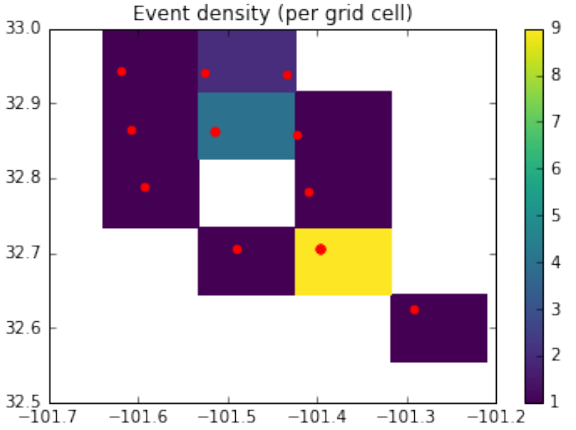
Thanks to Ryan May for access to test data from the Unidata GRB/S3 feed

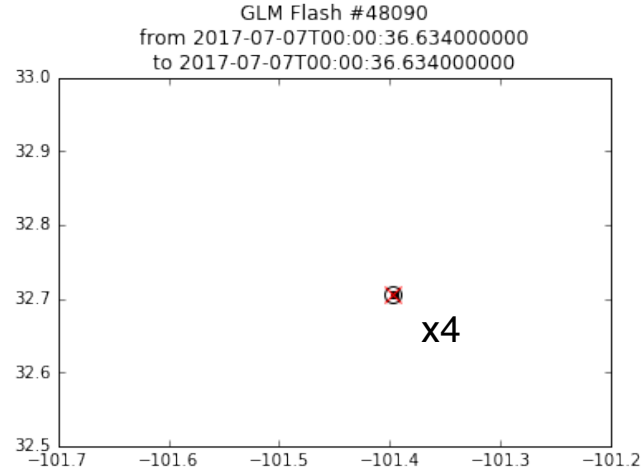
Imagery (a heatmap) is easier to digest than points, so count up events and flashes on a grid. Example: 8 flashes in 20 seconds (10,000 frames)

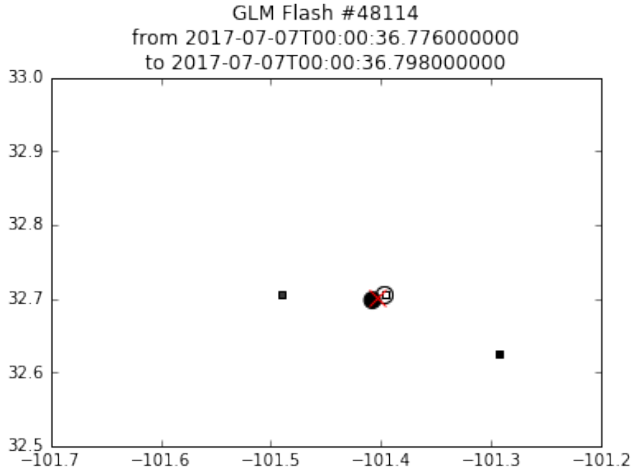
- *Event density: how many pixels detected light in each grid cell?*
- *Flash extent density: how many flashes produced light in each grid cell?*

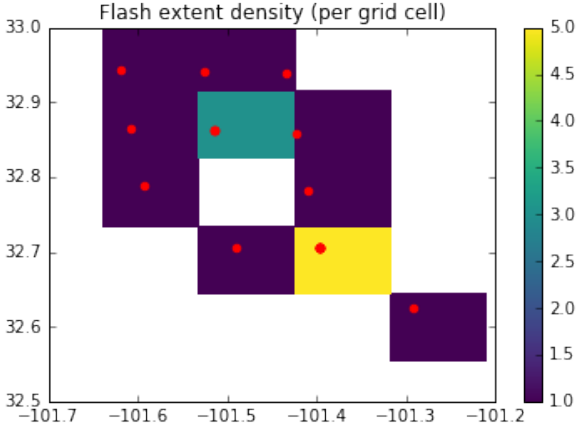






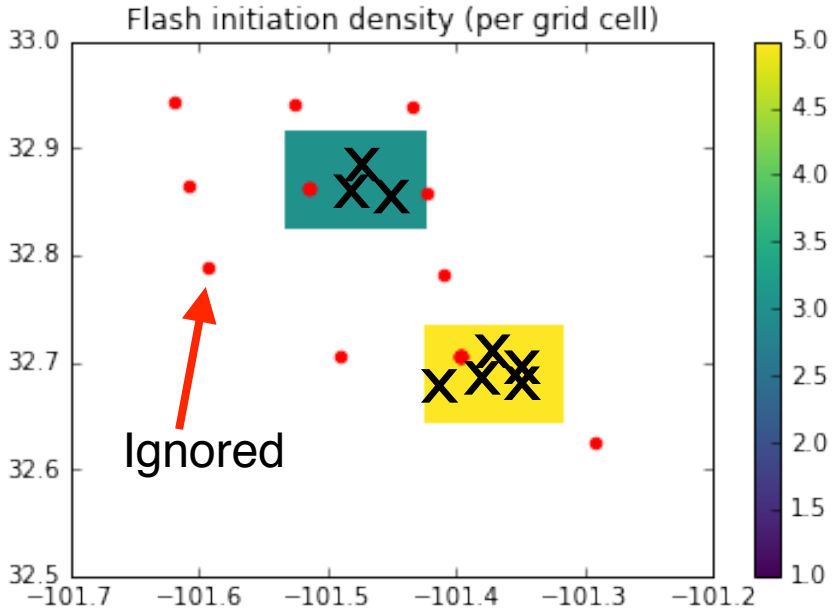


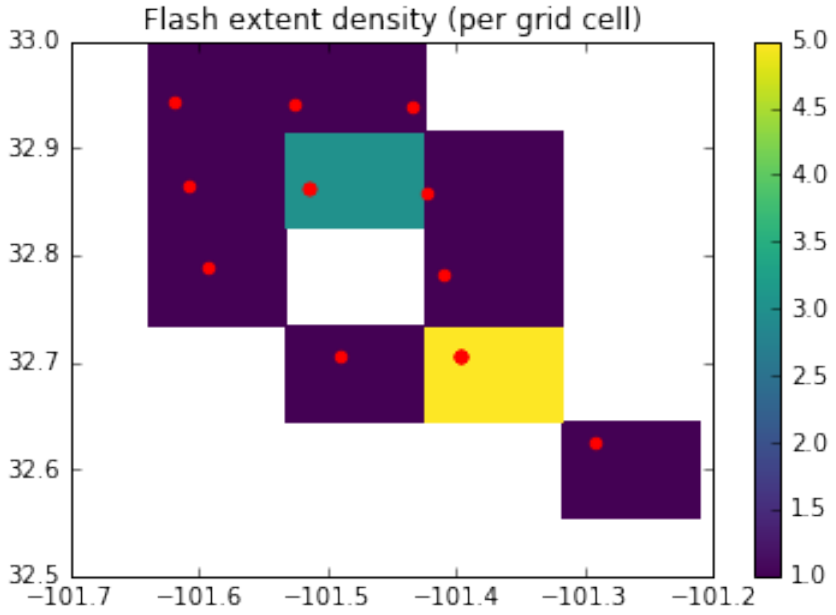




IMAGERY USING THE EVENT-FLASH LINKAGE IS EASIER TO DIGEST THAN IMAGERY MADE FROM FLASH POINTS ALONE

- 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.





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
```

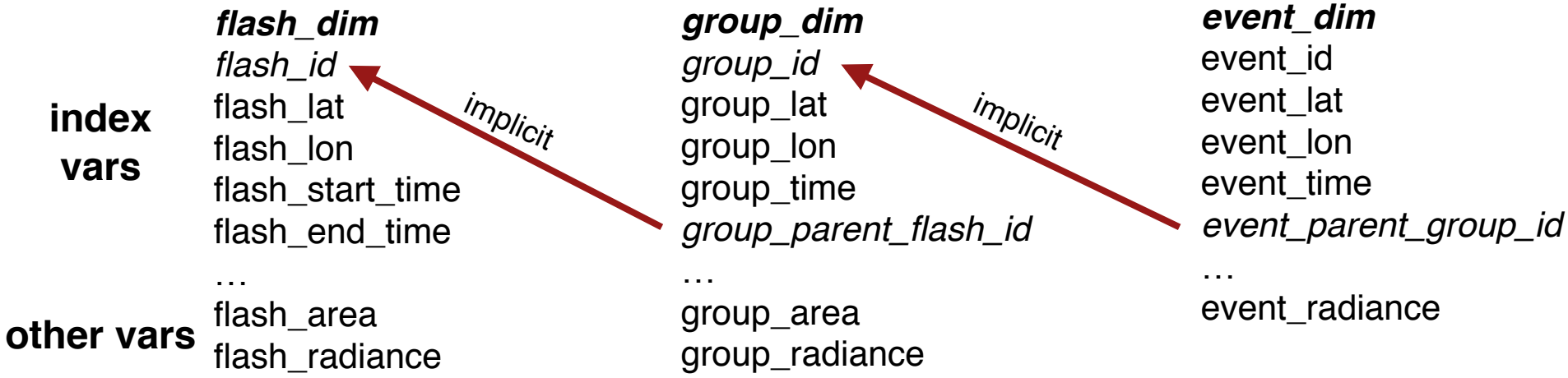
Lightning data are odd

- 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

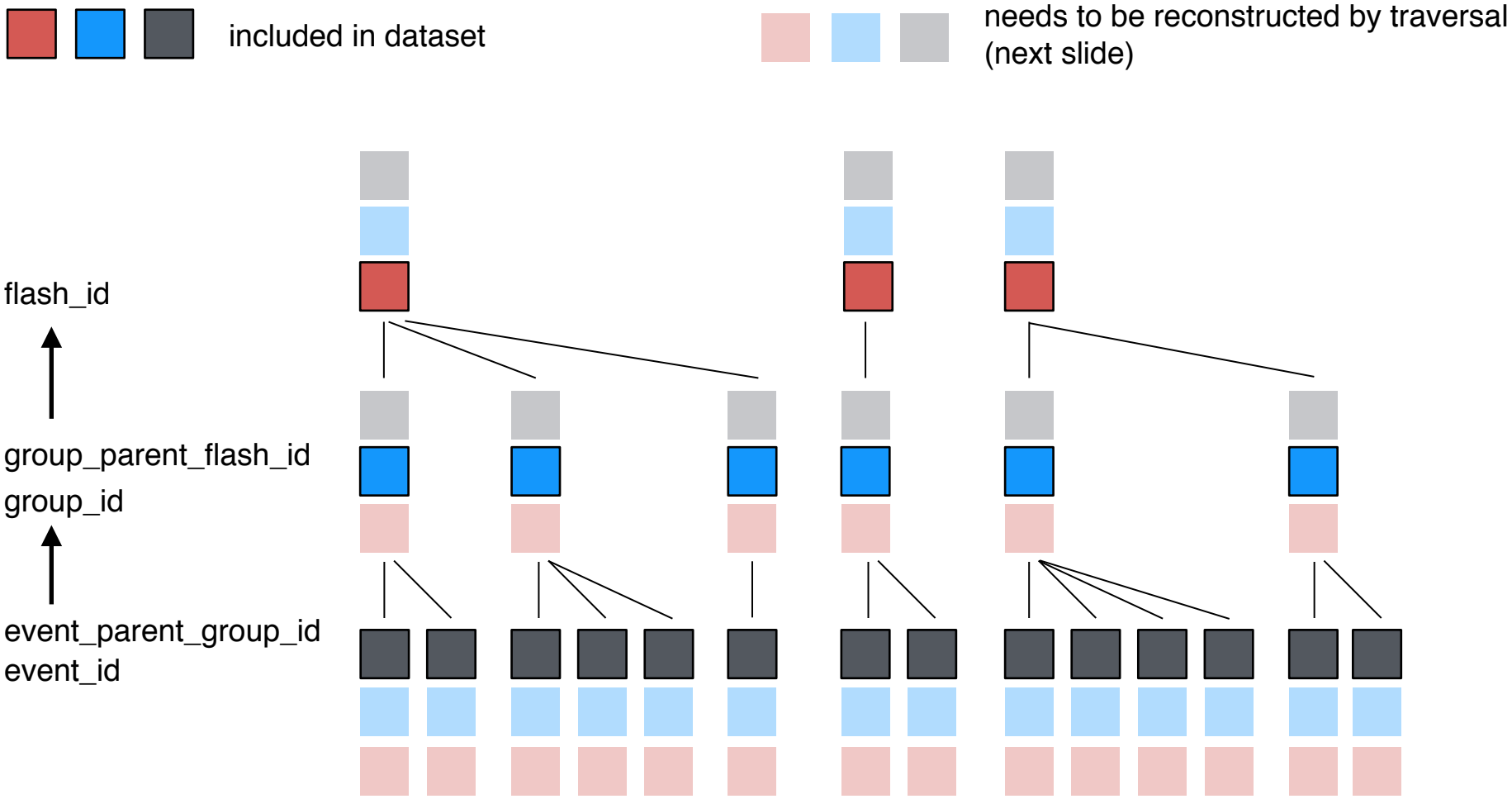
GLM data storage model

goes-r.gov, Product Definition and User's Guide
Vol. 5: Level 2 products

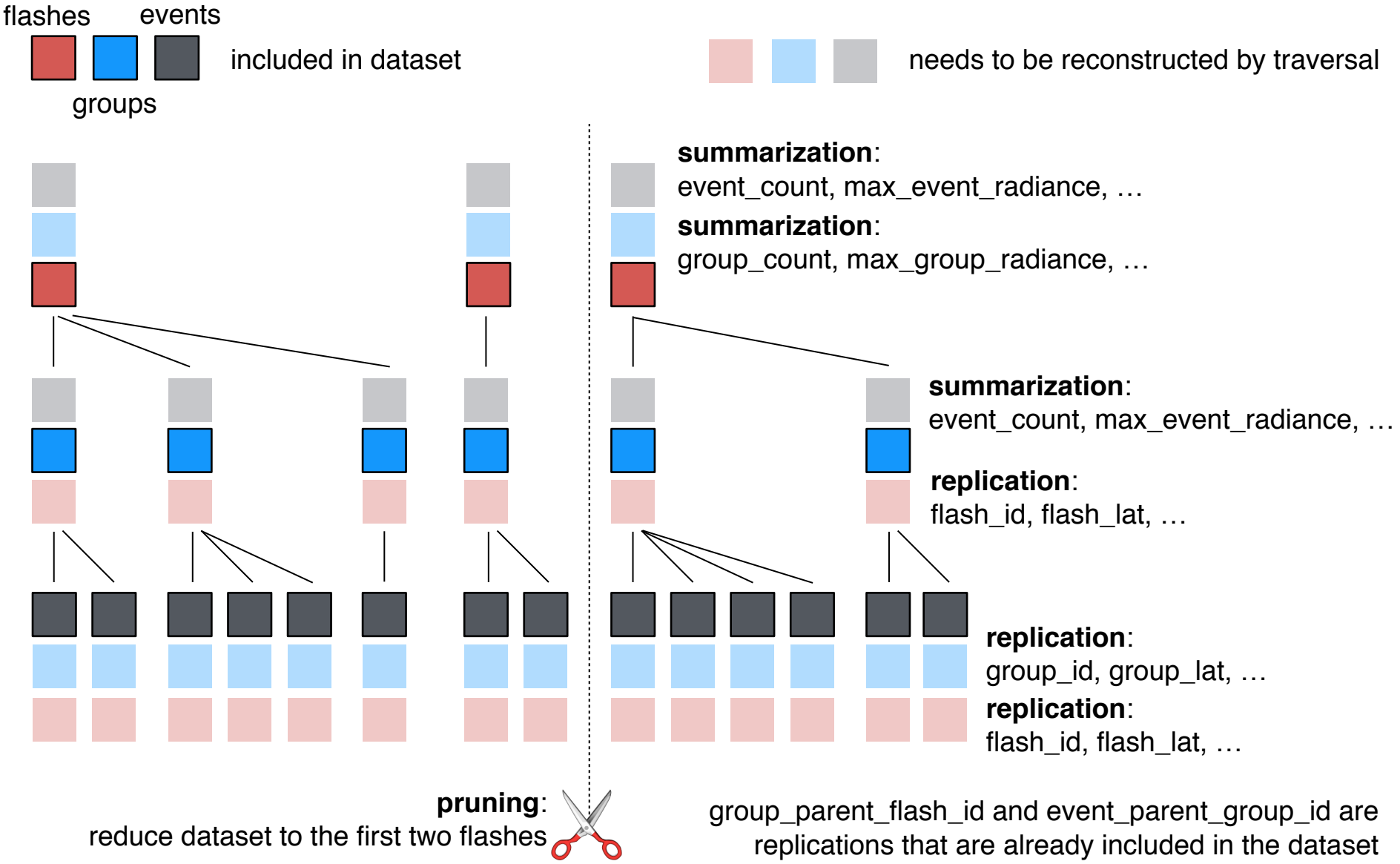
- NetCDF files with good metadata
  - *Climate and Forecast (CF) conventions where possible*
- *Stored as a CF `point featureType`*
  - *The only possibility for arbitrary locations and times*
  - *The CF spec includes an "indexed ragged array," but it's only a partial solution because it describes connections 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?*



Hierarchical linkage among dimensions



GOAL: AUTOMATE THE TRAVERSAL OF ARBITRARILY MANY LEVELS OF THE HIERARCHY



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)
```

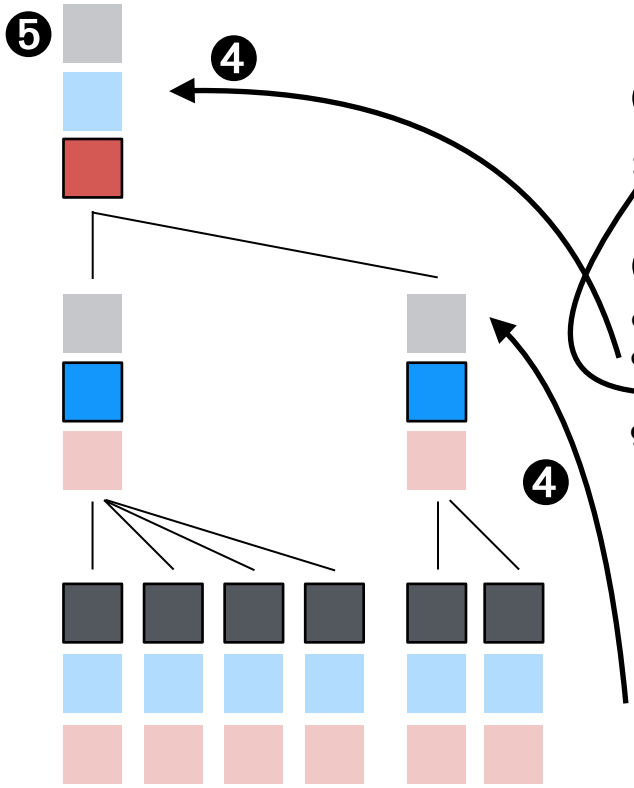
- Traverse and index hierarchical `xarray` datasets
  - *Other variables are along for the ride when indexing along a dimension*
- Automatically sets up two `Dataset.groupby` operations for each dimension
  - *e.g.*, `group_id` *and* `group_parent_flash_id`

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



❶ **flash_ids**

❷
```
grouper = dataset.groupby('group_parent_flash_id').groups
counts = [len(grouper[eid]) for eid
          in flash_ids]
group_ids = dataset['group_id'].data
```

❸ Block above is repeated at each level of the hierarchy, generating counts at that level

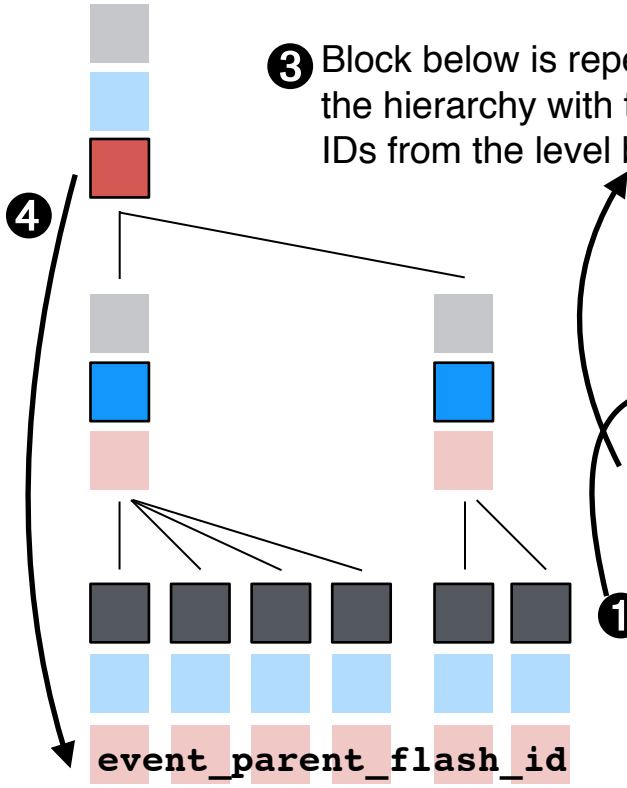
❺ **flash_child_event_count** is found using the same logic on replicated event_parent_flash_id

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



❸ Block below is repeated at each level of the hierarchy with the replicated parent IDs from the level below

❹

❷

```
grouper = dataset.groupby('group_id').groups
dim_idx = [grouper[gid] for gid
           in event_parent_group_ids].flatten()
group_ids = dataset['group_id'][dim_idx]
flash_ids = group_ids['group_parent_flash_id']
```

Looping over repeated event parents repeats the index for the group dimension

replicated IDs
replicated IDs

❶ **event_parent_group_ids**

Other variables for the group_dim come along for the ride thanks to xarray, so the parent IDs can be replicated, too

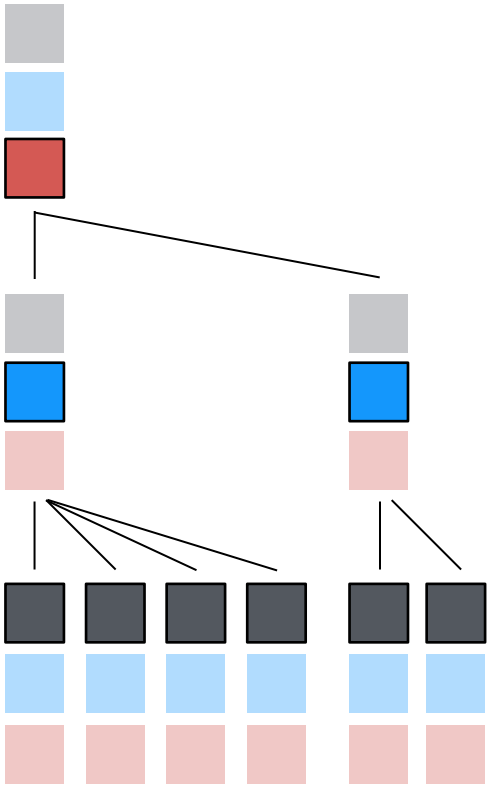
Pruning using OneToManyTraversal

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

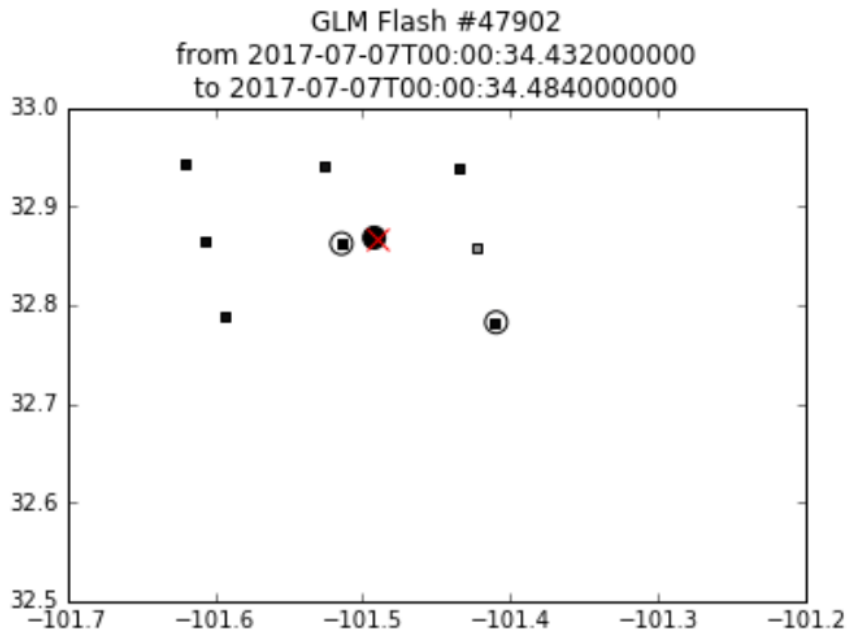
❹ Repeat the process for any entities above the flash level (if we had them) by ascending the hierarchy

Interactive use of hierarchical dataset pruning

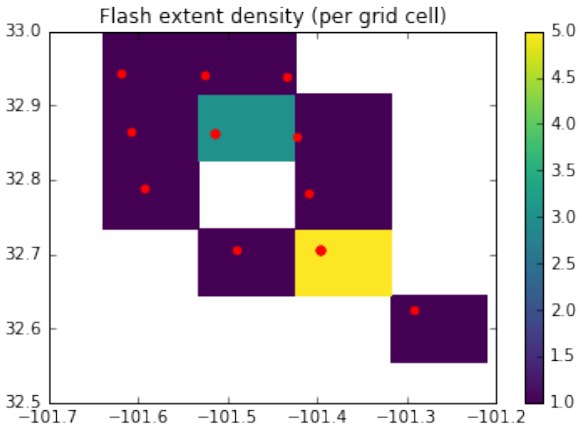
Interactively view flashes one at a time

```
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

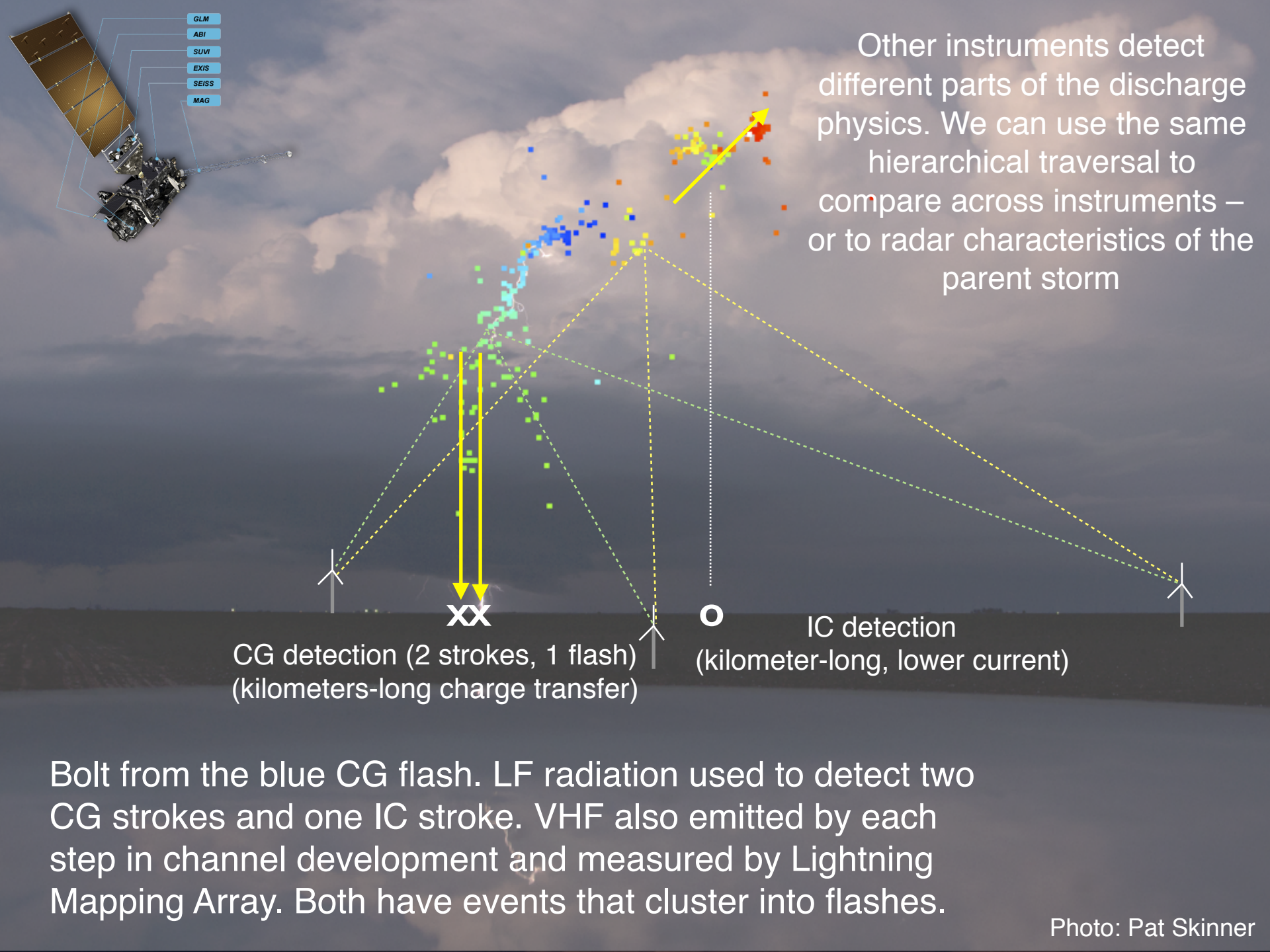


The flash extent density plots earlier in the talk were created using the flash ID replication process.



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.


GLM
ABI
SUVI
EXIS
SEISS
MAG
Other instruments detect different parts of the discharge physics. We can use the same hierarchical traversal to compare across instruments – or to radar characteristics of the parent storm
XX
O
CG detection (2 strokes, 1 flash)
(kilometers-long charge transfer)
IC detection
(kilometer-long, lower current)
Bolt from the blue CG flash. LF radiation used to detect two CG strokes and one IC stroke. VHF also emitted by each step in channel development and measured by Lightning Mapping Array. Both have events that cluster into flashes.
Photo: Pat Skinner