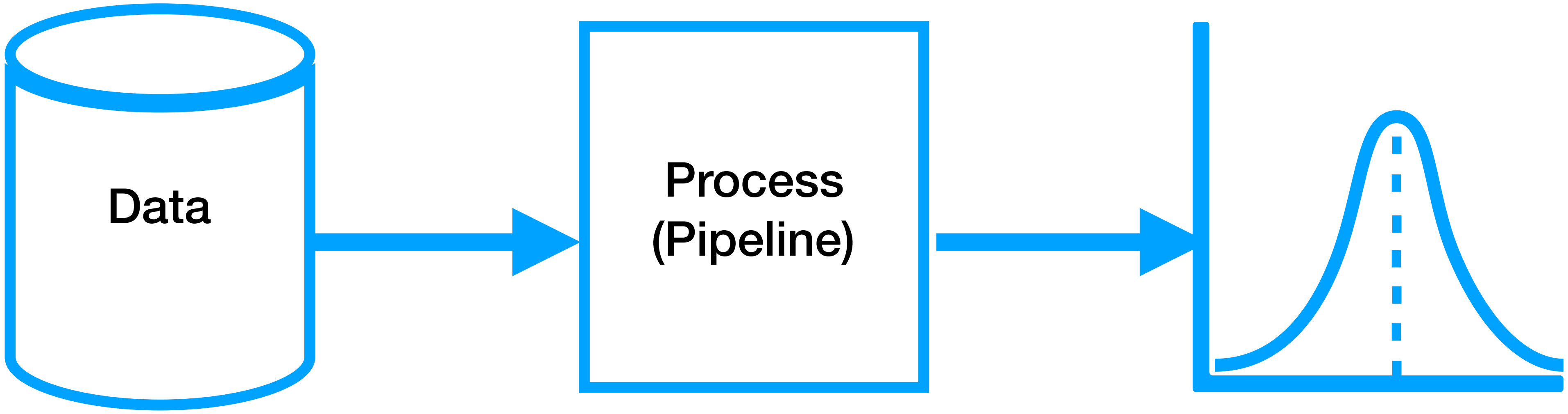
Workflow / Patterns

Lecture 05

Outline

- Writing for pipeline evolution
- Practical patterns for big data
- Interacting with the file-system

The End Result

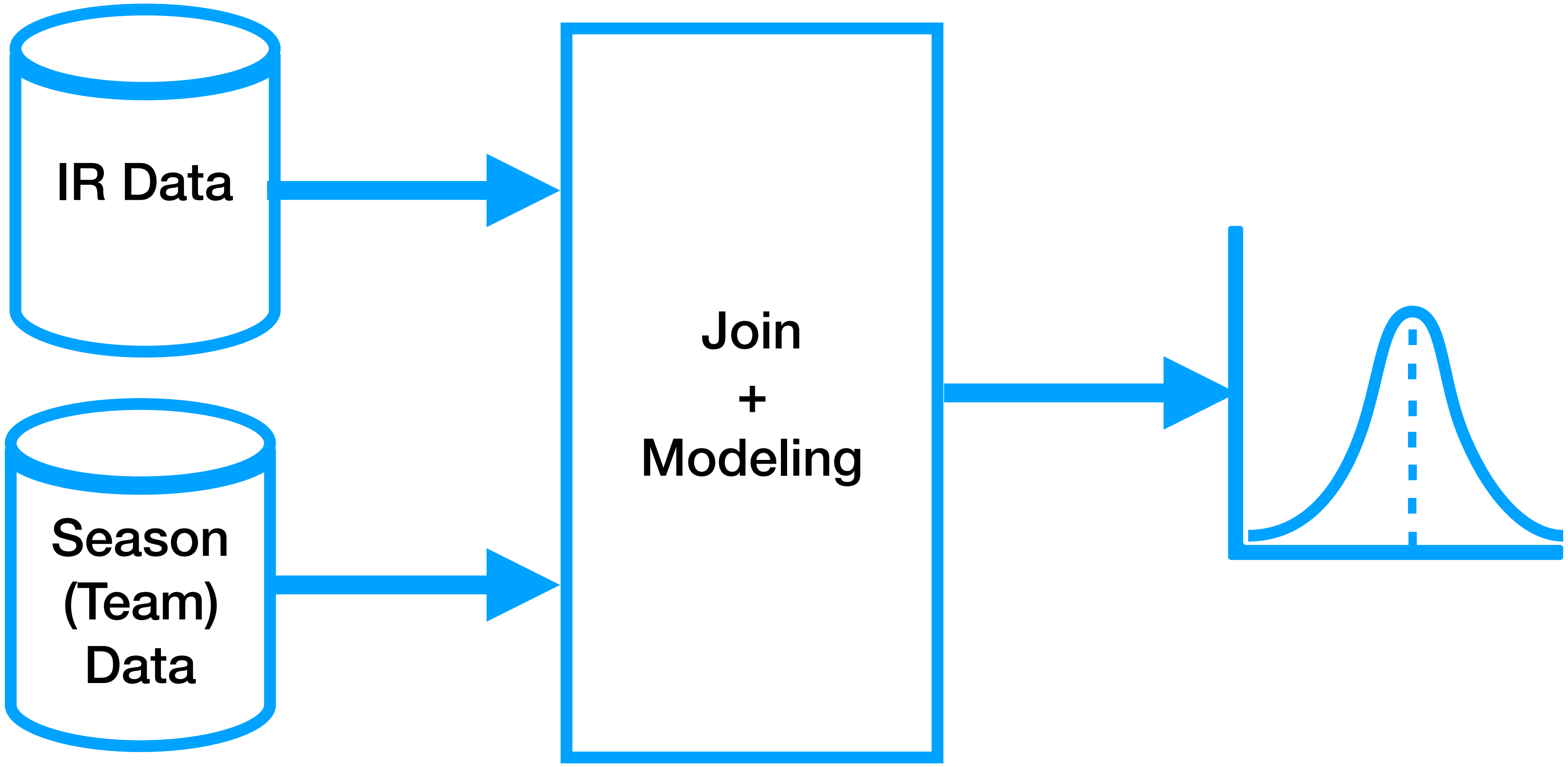


- **The clean abstraction of a finished pipeline.**
- **Reality is an evolving process:**
  - **Investigation: simple => grown in scope and sophistication.**
  - **Data becomes multi-faceted (sources; time; space)**
  - **Analyses become refined**

Example: Football

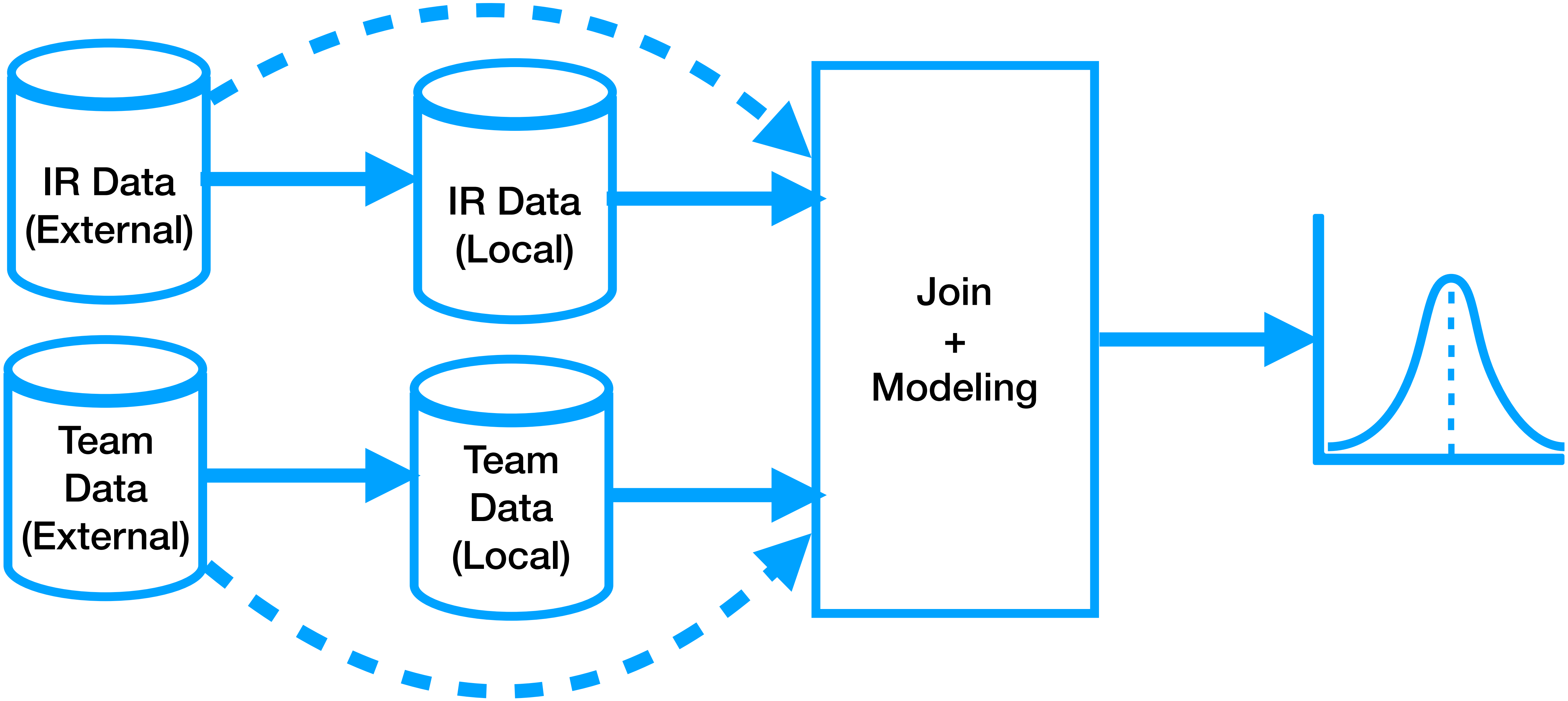
- Topic: Football players and injuries
- Data: Injured Reserve (IR) list & Football Statistics
- Narrow Question: Do injuries affect team success?
- Not to be forgotten: Broader social context!

Do Injuries Affect Teams?



- Player level model? Team Level?
- Even answering basic questions requires iteration:
    - Requires breaking up the pipeline; collecting data on disk

Pipeline, Refined.

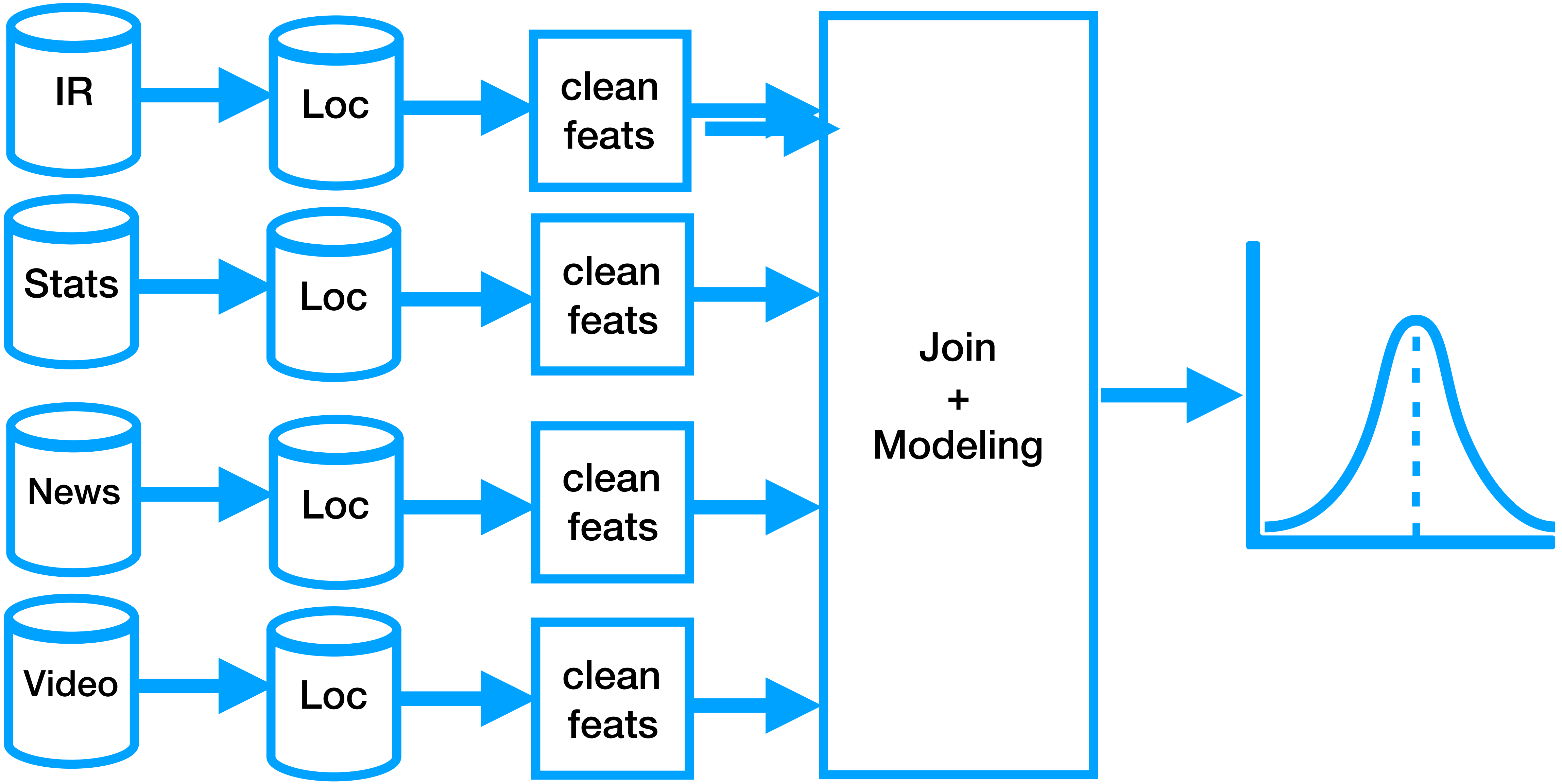


- Persist data locally to save processing and cost, ease debugging.
- Configured to *also* run w/o persistence (e.g post-exploration).

Considerations when Designing a Pipeline

- Further questions: What causes player injury? What rule changes would have largest affect? New technologies?
- Possible new data sources: player-level data; play-by-play data; news stories; game video.
- For each of these datasets:
  - Small data? (Yes? Don't bother optimizing!)
  - Heterogeneous or large? (Yes? Don't join prematurely!)

(Future) Pipeline



- Each step is an abstraction hiding complicated processing

Possible Pattern

```
# Data Ingestion Funcs: may write to disk or return in Mem
data_ing = {'IR': data_ing_IR, 'Team': data_ing_Team, ...}
# Data Processing Funcs: may read from disk, or in mem objs
procs = {'IR': proc_IR, 'Team': proc_Team, ...}

joined = ...
for d in datasets:
    ing_func, proc_func = data_ing['d'], data_proc['d']
    dataset = proc_func(ing_func(...))
    joined = joined.merge(dataset)
out = mdl(joined)
```

Patterns: what and where

- Data ingestion code can *either* write to disk or output DF
- Processing code can *either* read from disk, or input DF
- 'Driver functions' build the pipeline from small functions:
  - Implement logic for persisting/reading to/from disk.
  - Creates a *directed acyclic processing graph (DAG):*
    - *Nodes are (data) sources and targets*
    - *between edges that process the data.*

Patterns: How

- Processing DAGs help organize and structure the code.
- The processing pipeline hides the complexity of code *up close* and manages it as the code grows.
- As code and data grow, need patterns to manage efficiency.

Memory and Speed

- Need ways to manage memory and speed constraints, without too much compromise on code generality.

- The *streaming vs batch* processing paradigm:

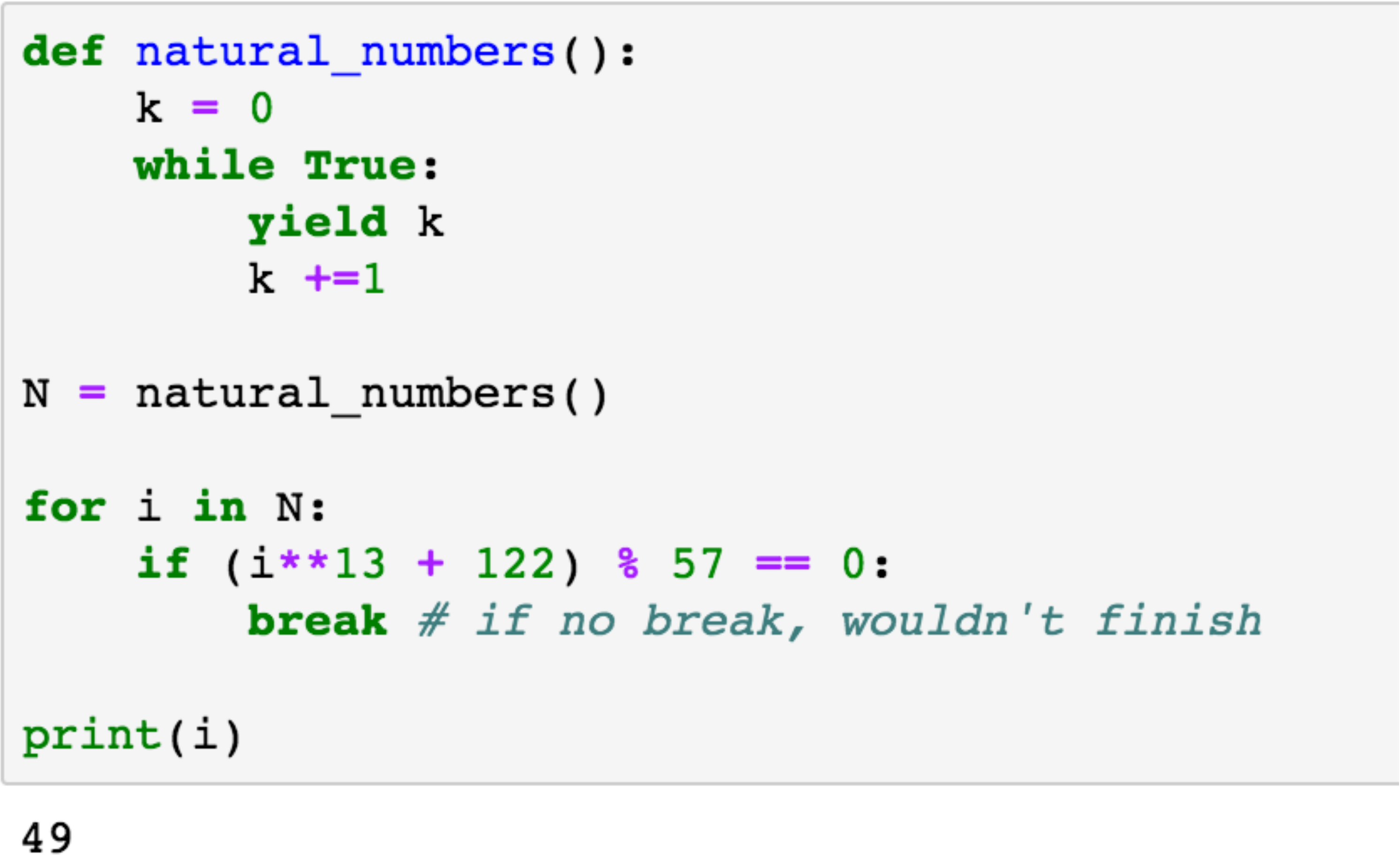
  - Streaming: processing data as it comes requires a small memory footprint.

  - Batch: processing data in larger chunks allows for efficient memory allocation (fast).

- Reality: a spectrum (from batch-size 1 to ∞).

Streaming in Python

Iterators *stream* through elements of an iterable; only the current element is stored in memory.



```python
def natural_numbers():
    k = 0
    while True:
        yield k
        k +=1

N = natural_numbers()

for i in N:
    if (i**13 + 122) % 57 == 0:
        break # if no break, wouldn't finish

print(i)
```

```
49
```

Streaming though the natural numbers

Examples: Streaming

- Iterating through bytes, integers, lines...
- wget: download *buffered byte streams* (e.g. 256K buffers)
- Iterating though larger 'chunks' or 'batches':
    - Choose a csv chunksize? Choose a convenient 'unit' of processing? (year, page, person, observation-group)
- When data comfortably fits in memory, a batch can contain all the data.

Processing & Streaming

- Write library functions that:
  - inputs a chunk & outputs a processed chunk.
- Create a 'driver function' that:
  - regulates (the size of) data intake, and
  - compose library functions, feeding one chunk at a time.
- May involve:
  - Writing many files to disk,
  - Collecting processed chunks into appropriate 'units'

Streaming huge files

- Question: what should a chunk be for course domains? (i.e. what is the unit to iterate through?)
- If non-tabular organize many files on disk (~partitions)
    - Store in many files for possible parallelism advantages
- If tabular, make a relational database (e.g. local sqlite)