Pipelines

(of the data variety)

Data as a Service

- Ingesting data from a large variety of sources
- Standardizing, mapping columns, and centralizing for dashboards and analytics
- Know where the data is coming from, able to track back to raw inputs

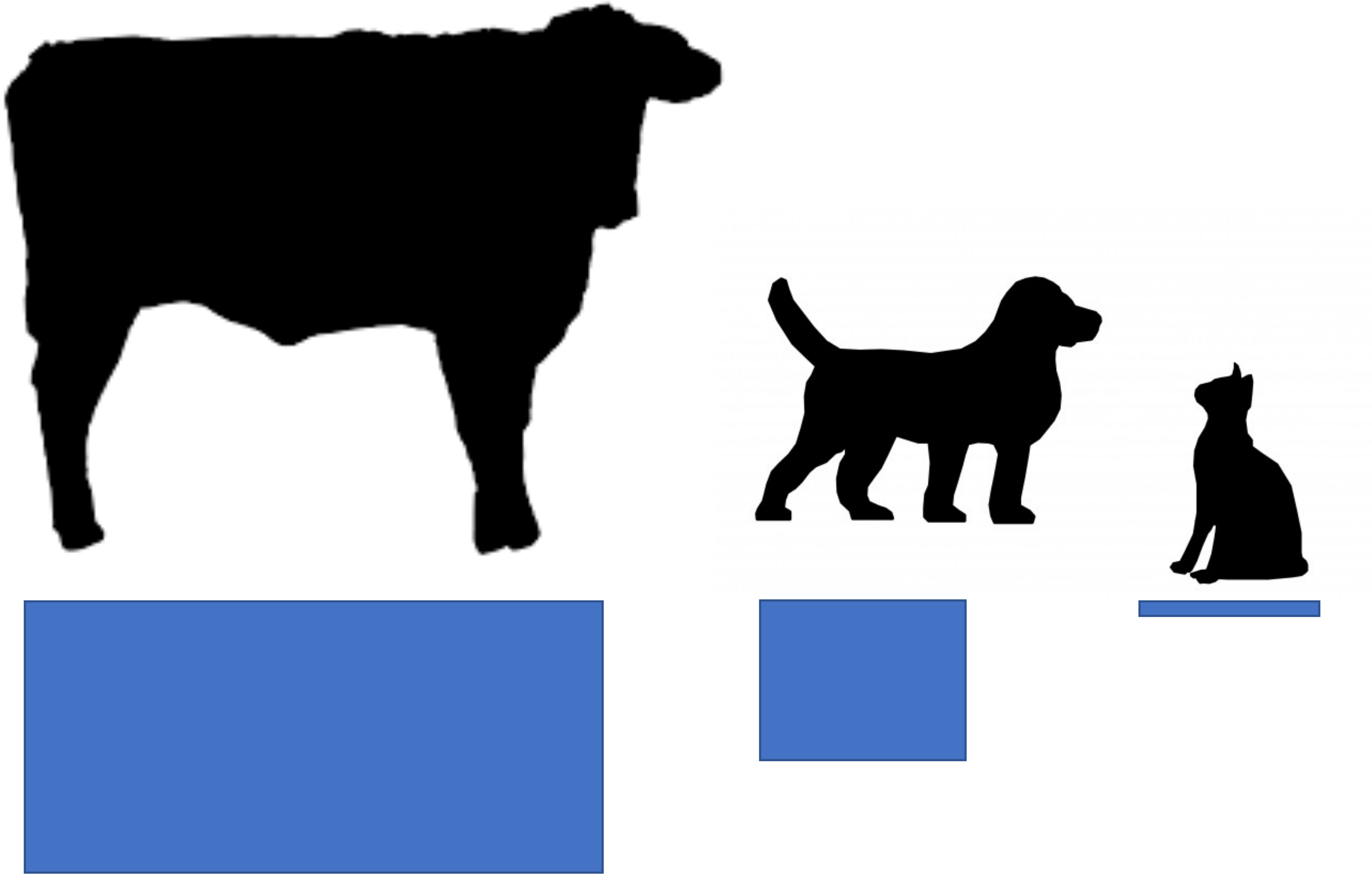
Load Tracking

- Use uuids
  - Unlike autoincrement ids obscures any underlying details or counts of other loads
  - Allows relatively risk free merges of data loads potentially coming in from multiple sources
- Have a tracking table that timestamps the time a given dataset hit your internal servers
  - Web response timestamp
  - FTP file created time
  - Etc
- Have supporting table structures for status, approvals, etc.
  - Should be able to show how long each point in your pipeline takes

Cat/Cow/Dog/Fish

- Have a standard strategy for how you load, query, and archive data depending on the size and rate of growth
- Abstract the mechanics of moving data such that end users and analysis have a standard way of accessing data from a source
- Standardize archive strategies based on size and growth rate of data

Simple Naming Conventions



Dog Grooming

- Dog table users will be structure agnostic
- Define parent table structure and breed of dog, setting appropriate partitioning column.
- Enter table parameters id's per partition, date range per partition, or hashed partition count into a dog meta data table.
- Call dog maintenance code prior to and possibly after use.
- Dog grooming code is agnostic in regards to table usage. Data could originate from timescale statistical views, or from a TX low and slow insert, dog grooming code doesn't care.
- End user code is partition structure agnostic. Even if table is used in ten places, basic underlying partitions can be changed without updating code in ten different places.
- Grooming will anticipate needed partitions and provide, in the case of hashed partitions will flesh out hash structure automatically.
- Even if something goes severely wrong, worse case the __default table will get data dumped into it. Grooming will adjust partitions appropriately to deal with the data retroactively.
- Three columns are required, rest is up to the user

Streaming Data

Simplified to “FISH”

Use Timescale DB to abstract streaming work

End-Users access all data through an “ANIMAL” class

Archive Strategies

- If the data that a given analysis is derived from can easily be regenerated saving monthly or even quarterly snapshots can be adequate
- Depending on the frequency with which a given analysis is updated not every output needs to be saved
    - Hold just the X most recent runs, and end of week or end of month snapshots
    - Where X should be set to allow for quick recovery to a reliable state should an error be detected in a recent load
        - Example, a daily load should keep the 11 most recent loads and then end of month snapshots. This allows the engineer running to go on vacation for a week and come back with the most recent good load still in the system, assuming that a bust occurred just after they left

Raw Data

- Pull in the data agnostic to future analysis
- Process data only AFTER you have the ability to corroborate your data matches source data and validation that import/subsequent checks are ok
- Have a load table that accurately represents the source data with minimal data cleaning

Load Checks

- Have automated checks on incoming loads
  - Validate that the initial download in to the system is correct
  - Check sums if possible (compare a couple of sums at source and destination)
  - Did the ingestion process run for a reasonable amount of time
  - Light weight but cause data processing to fail fast
- Manually validate the data at key steps
  - Airplanes don’t always fly on autopilot
  - Automate checks once they prove to be useful and you have honed in on key metrics
  - Once you’ve ran a manual check twice it should be automated

Build Views

- Combine multiple data streams into coherent analysis
- For analysis that combines multiple data sources allow each underlying source to update at the cadence that is most efficient for that source
    - Other analysis may need more frequent updates of a particular source
- Use load ids and many-to-many tables to keep track of which versions of the data went in to a given analysis
- Think of each step as a building block that could potentially feed in to many other sources

Rolling Updates

- Set a standard for how you timestamp incoming data
    - If scraping a website the timestamp reflects the response time of the remote server
    - If from a shared file the timestamp is the data created in the shared folder
    - The timestamp should represent the moment a given piece of data entered a given system’s perimeter
- When doing an analysis store the max timestamp used
- Combining these two methods it then becomes possible to know which data was available at the time a given analysis was run

State of the States

- Regularly run high-level checks of the whole system
- Silent failures or small leaks of the data are possible
- No automated system is going to catch all failures so manual reviews are vital
- Evaluate indexes and long running queries at this time

Alerts

- Have alerts on success of jobs as well as any type of failure
- Tier the alerts
  - Yellow should alert on lower-level failures to establish patterns of issues, but no immediate response should be taken
  - Orange should alert on bust that may require more immediate issues and further investigations, ie a count is outside of a normal range
  - Red should happen for when a system or load failed and should require immediate attention.
- Keep red alerts to a minimum to prevent going numb to responses

Trend Alerts

- Have a server that runs more in depth checks on loads offline of your normal load systems
- Individually the queries that are ran should be small, but if checking for every variant of X, for example, then collectively these alerts are enough of a load on systems to require their own system
- These alerts can be both for more nuanced data quality checks and signals in the data

Ribbon Data

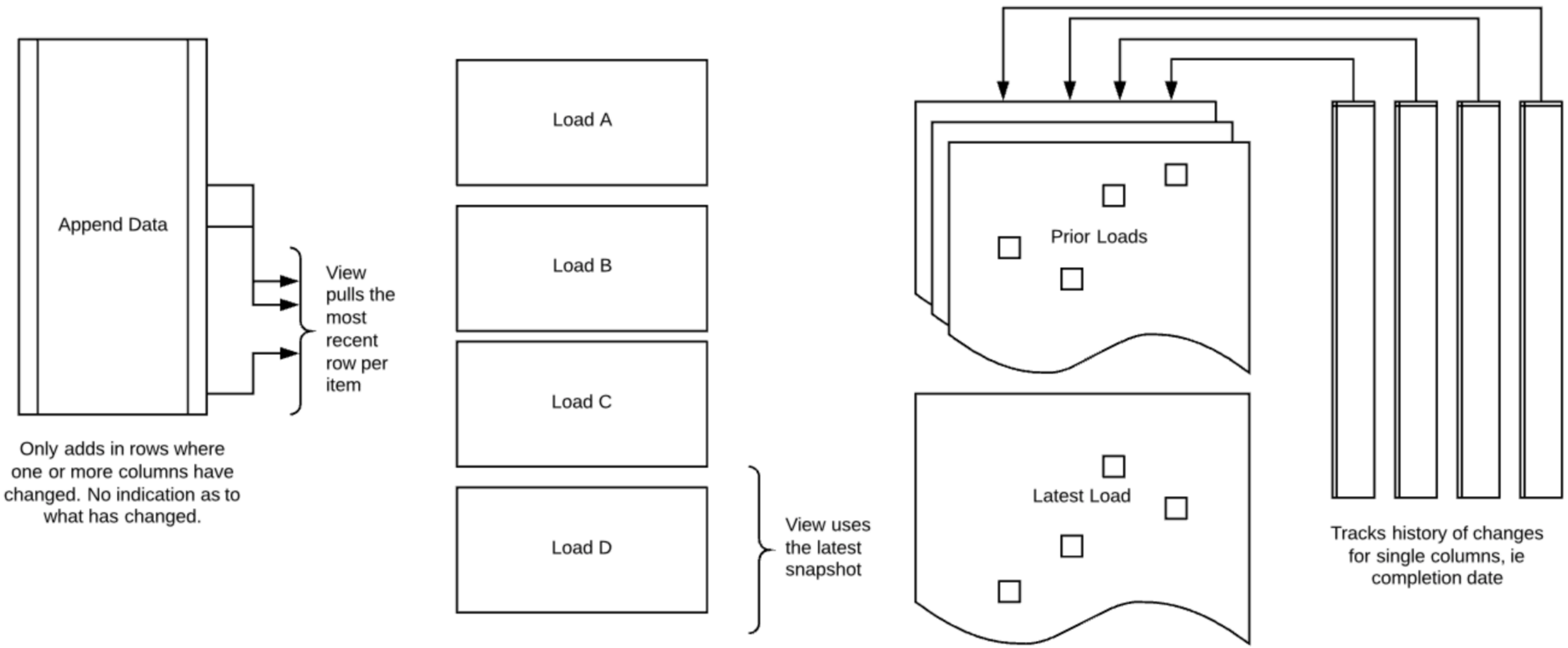
- For each load of data check against prior versions of the data
- Store the changes for targeted fields individually in a table for faster historical analysis
  - Row counts
  - Noting which rows of data changed where and when
- Consolidates historical datasets into a change log
  - If done on at the level of change per key column allows a very detailed look at how the data is changing over time
  - Ability to quantify what is typical for an update at a very detailed level

Append, Replace, Ribbon

Append

Replace

Ribbon



View pulls the most recent row per item

Only adds in rows where one or more columns have changed. No indication as to what has changed.

View uses the latest snapshot

Tracks history of changes for single columns, ie completion date

View still uses the most recent snapshot, but has leveraged values from previous loads, and merges in the individual column trackers to allow for filters like "permits that are new in this load"

Ribbon Advantages

- Each ribbon table tracks changes in a single field load over load
- Narrow means that the history of changes remains fast to query
- Clear WHAT has been changed and added load over load
- For metadata fields this provides an objective measure of growth
- For value fields the amount changed load over load generates a vector illustrating how the values are changing

Expense Reports Example

- Best example of this would be looking at expenses being filed for a team
- Pulling a snapshot shows the rate at which expenses are being submitted
- If a team (or individual) is lagging on submitting their reports seeing the changes across each snapshot gives an idea of where the totals may land