Joey Zuhusky – oneThreeBiotech take home project.

Part 1

Most of the choices I've made in the schema design are based on the observed data-model displayed on the Drug Bank website. We'll have a primary table which stores DrugBank info about the Drug, e.g. SMILES and the drug name. The DrugbankID will serve as the primary key / unique identifier for drugs. The drug targets are enumerated on the site as a list, so I've broken those targets out into a separate table. The same for the alternate identifiers for the Drugs. I've also created action and alt-identifier type tables, so we can enumerate the types of actions and identifiers seen during the scrape, and thus give each of these types unique integer keys for easy and clear joins based on integers.

Table 1:

Naturally, the first table will store information about specific drugs. We choose relevant fields like the "drugbank_id", a human-readable "name" and the SMILES formula, which is unique to a drug. The table design below makes the assumption that we uniquely identify Drugs by their "Drug Bank" ID. If we were to have multiple data sources, which could provide multiple identifiers to unique drugs. So then, it would make sense to not uniquely identify drugs by their "drugbank_id", but by some other universal identifier, perhaps something we designate internally. Then, other identifiers would be stored in another table, with a reference to our unique drug identifier.

```
CREATE TABLE drugs (
    drugbank_id char(7) PRIMARY KEY,
    name varchar NOT NULL,  -- Something "Human Readable",
    smiles varchar
);
```

Table 2:

Next, we need to store "Drug identifiers", which are just other data sources which are referring to the same drug in question. Since Drug Bank is our primary data source, we keep the DrugbankID in the main drugs table and use that ID as the primary key / identifier, but have alternate identifiers stored in a separate table.

Before we store the alternate identifiers, it's natural to enumerate what type of alternate Identifiers we have, hence the table below "drug_identifier_types". These identifier types are updated as we see new identifier types when scraping the drug bank website. Having this reference table makes joins and where clauses simpler in my opinion. It's also nice to be able to enumerate all of the different alternate drug identifier types without having to make a DISTINCT query on another table.

E.g. some rows in the drug_identifier_types table might be:

(1, KEGG Drug),

(2, KEGG Compound),
(3, Therapeutic Targets Database),
…

```
CREATE TABLE drug_identifier_types (
    identifier_type_id SERIAL UNIQUE,
    identifier_type_name varchar UNIQUE
);
```

Table 3:

We store the actual alternative identifiers in this table, with references to the Unique Drug Bank ID, and the Type of the Identifier.

```
CREATE TABLE drug_identifiers (
    drugbank_id char(7) references drugs(drugbank_id),
    alt_identifier_value varchar NOT NULL,
    alt_identifier_type_id int references  drug_identifier_types(identifier_type_id) NOT NULL,
    alt_identifier_url varchar
);
```

Table 4:

Similar to our process of enumerating the types of drug identifiers, we also have a table to enumerate the different actions for drug targets.

```
CREATE TABLE drug_action_types (
    action_type_id SERIAL UNIQUE,
    action_type varchar UNIQUE
);
```

Table 5:

When we store the drug targets, and their corresponding actions, we use references to the type table to indicate the drug action for a particular target. There should be multiple rows in this table for targets which have multiple drug actions.

```
CREATE TABLE drug_targets (
    drugbank_id char(7) references drugs(drugbank_id),
    gene_name varchar NOT NULL,
    action_type_id int references drug_action_types(action_type_id),
    UNIQUE(drugbank_id, gene_name, action_type_id)
);
```

Perhaps as the application matured, I'd imagine we could have a table(s) with data concerning different genes, and have this relation "point" to that data.

Part 2:

DAGs

When building data ingestion tools, it's wise to not re-invent the wheel. Generally, a good and commonly used model for data ingesting pipelines at a high-level is a Directed Acyclic Graph / DAG. This is also the model that I have personally worked with (Kensho – Adagio {note "dag" in adagio}, in-house data ingestion framework), and the model I know is behind popular tools such as Apache Airflow and Luigi, which are commonly used open-source tools for designing data-pipelines. Each node in the DAG represents a quantum of work to be done in the pipeline before the result of that work is handed off to another downstream node.

Breaking the ingestion pipeline into discrete steps helps improve coding efficiency by identifying bits of systems to be designed and implemented by engineers. Because each step is defined by clear input and output contracts, the work itself can be more easily split up among engineers, while producing less coding dependencies.

Engineers and designers can also reuse nodes and/or bits of pipelines for other workflows or pipelines. This could be helpful if we are dealing with hundreds of datasets where there is bound to be overlap.

It also becomes easier to monitor pipelines, especially if things break. Since the problem of ingesting data has been discretized into a DAG, we can monitor each step in the graph-path / pipeline to see exactly where in the ingestion pipeline things went wrong, and quickly debug the situation, since we know exactly what bit of code pertains to each node / step.

DAGs also scale well. If a large amount of data is to be ingested, DAGs provide a natural way to parallelize and speed up processes. If the source of the data and the destination (sink) remain the same, the steps in the pipeline can be duplicated for subsets of the input and run in parallel to speed things up.

In the end, I think that data ingestion pipelines are very common, and there are a number of good ways to design them. DAGs and frameworks that build on top of DAGs conceptually, in my opinion are natural ways to think about and build data ingestion pipelines.

Other considerations

In my experience, the teams I've worked on have always found it useful to not only break down problems into discrete steps, but also to consider more nice-to-haves when designing pipelines. I've found it incredibly useful to collect and value meta-data about the pipelines.

By having readily available answers to questions like:

- Where did this data come from?
- When was this data last updated / touched?
- What code / step in the pipeline touched it?

It can rapidly speed up the process of debugging systems and ensuring data-quality. Considering meta-data about data pipelines and basic information about the provenance and history of the data in our systems, in my opinion is incredibly important to maintaining data pipelines.

Many times, open source frameworks provide the means for collecting this data. If not, I believe it's prudent to be considering these questions when building ingestion tools / pipelines.