

LECTURE 23a

Data Ops and Pipelines

April 17, 2024

Data 101, Fall 2024 @ UC Berkeley

Aditya Parameswaran <https://data101.org/sp24>



Automating Data Operations

Lecture 23, Data 101/Info 258 Spring 2024

Automating Data Operations
dbt Overview
dbt Models
dbt Tests
Spark as a Workflow System
Pub-Sub and MQs
Workflow Systems

A very pragmatic topic: Automating Data Processing

In the “real world” you don’t deal with data sets, you deal with **data feeds**, or **data streams**.

- Data “arrives” on a regular basis.
- When data arrives, you need to “do the right thing.”

Things to consider when designing **data pipelines** to automate data processing:

1. How does data “arrive”?
2. What kinds of things do you need do upon arrival?
3. How do you automate the work?
4. How do you maintain quality and handle exceptions?

1. How does data “arrive”?

Push: To some location

- Files appear in a staging folder, e.g. in AWS S3; or
- Tables get populated in a data warehouse, e.g. in BigQuery; or
- A software system API provides the data
 - E.g. publish-subscribe (**pub-sub**) or **message queueing** systems (more later)

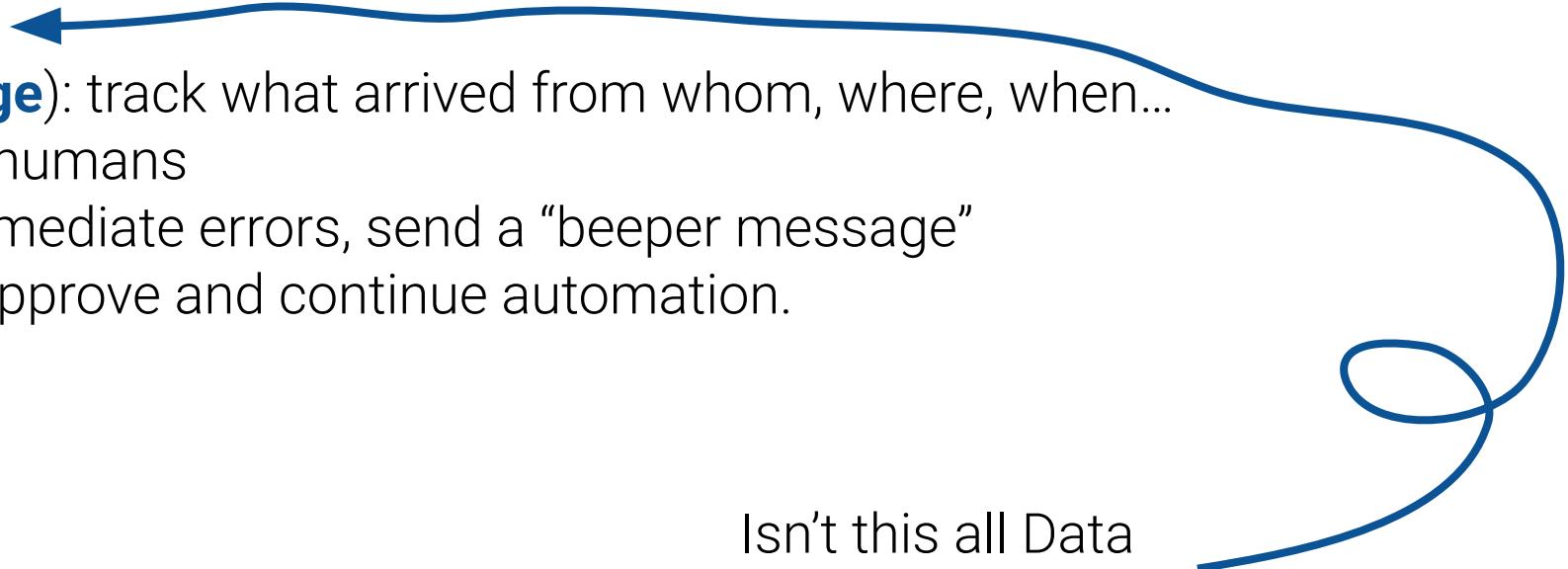
Pull: Extract data from some source

- Via scheduling software; or
- On receipt of an event, e.g. from pub-sub or MQ systems
 - Or slack or email or ... really anything

2. What do you do upon data arrival?

Many possible options, and you should do all of them:

- Run **data quality (DQ)** tests
 - E.g. check the new data has expected schema, integrity constraints, data cleaning checks
- Load into “the right place” (a data warehouse or a curated set of directories)
- Run transformation logic
- Log Metadata (**Data Lineage**): track what arrived from whom, where, when...
- Pass along info/control to humans
 - e.g., when there are immediate errors, send a “beeper message” to someone who can approve and continue automation.



Isn't this all Data Transformation?

2b. Isn't this all Data Transformation? A: Yes, but...

Yes: Data Engineering is largely automated data transformation

...**But:** In many contexts, this involves tasks across many software systems.

Our SQL-centric ELT this semester has been **very clean**.

- Not unrealistic -- some places do this for their main operations.

However, many pipelines involve different systems for some/all pieces of the puzzle:

- **Derived data:** Preparation/Cleaning/Transformation systems, Entity Resolution/Standardization systems
- **Data Structure Maintenance:** Indexes, Materialized View updates, Data Cube updates
- **Data Security/Compliance Systems:** “masking” private data, encryption
 - See: EU’s GDPR, or California’s CCPA for privacy regulations
- **Metadata storage:** catalogs, data lineage (tracking where data came from and what happened)
- And so on, and so on.

We need to automate these many tasks/systems and their handoffs

- **And** deal with exceptions, human interventions/approvals, etc.

3. How do we automate the work?

Data pipelines are automated with more kinds of systems! 😊

Workflow Systems: manage processes (human or software)

- With process dependencies (order of activity)
- Activiti, Airflow, BPL, Luigi, ...

Dataflow Systems: manage data movement through operations (software)

- Handle data dependencies (inputs/outputs)
- Spark
- dbt + SQL
- Flink, Dataflow, etc.

Messaging/Queueing/Eventing: hand off data reliably from one system to another

- Kafka, RabbitMQ, AWS SQS, MSMQ, MQSeries, ...

Practically: Many times you will use combinations of these, and it can be a matter of taste/experience which to use where.

- You'll learn it "on the street..."
- Or, you'll subscribe to company "religion" and be part of many debates

4. How do you maintain quality and handle exceptions?

Data Quality and **Software Quality** share similarities, but also have many differences.

The notion of a “**test**” is now **data-centric**:

- logical (schemas, constraints, etc)
- statistical (outliers, distributional drift)
- etc. Quite different from software debugging!

However, SW Eng DevOps principles are getting more play in DataOps pipelines.

- Test-driven development
- **Continuous Integration** / Continuous Deployment

We'll look at **dbt** as an emerging example.

dbt Overview

Lecture 23, Data 101/Info 258 Spring 2024

Automating Data Operations
dbt Overview
dbt Models
dbt Tests
Spark as a Workflow System
Pub-Sub and MQs
Workflow Systems

3. How do we automate the work?

Data pipelines are automated with more kinds of systems! 😊

Workflow Systems: manage processes (human or software)

- With process dependencies (order of activity)
- Activiti, Airflow, BPL, Luigi, ...

Dataflow Systems: manage data movement through operations (software)

- Handle data dependencies (inputs/outputs)
 - Spark
 - dbt + SQL 
 - Flink, Dataflow, etc.
- (more later)

Messaging/Queueing/Eventing: hand off data reliably from one system to another

- Kafka, RabbitMQ, AWS SQS, MSMQ, MQSeries, ...

Practically: Many times you will use combinations of these, and it can be a matter of taste/experience which to use where.

- You'll learn it "on the street..."
- Or, you'll subscribe to company "religion" and be part of many debates

dbt (data build tool) for ELT pipelines and data quality

dbt (the “data build tool”) is a commercial open-source project.

- free to use the CLI (command-line interface, i.e., terminal) version
- pay dbt labs for cloud-based scheduling, CI-git integration, GUI

A **declarative specification** framework for:

- **pipelines of ELT-style SQL** transformations
- **data quality tests**

dbt Core and Cloud are composed of different moving parts working harmoniously.

All of them are important to what dbt does — transforming data—the ‘T’ in ELT. When

you execute `dbt run`, you are running a model that will transform your data

without that data ever leaving your warehouse.

[\[dbt docs\]](#)

```
[tmp]> dbt init my_project
Running with dbt=0.19.1
Creating dbt configuration folder at /Users/jmh/.dbt
```

The expectation is that you connect this to an existing Data Warehouse (i.e., SQL database)..

dbt (data build tool) for ELT pipelines and data quality

dbt (the “data build tool”) is a commercial open-source project.

- free to use the CLI (command-line interface, i.e., terminal) version
- pay dbt labs for cloud-based scheduling, CI-git integration, GUI

A **declarative specification** framework for:

- **pipelines of ELT-style SQL** transformations
- **data quality tests**

This tool is very recent!

- ≤v0.X in 2016, then v1.0 in 2021
- various other companies standardize on dbt format

We'll do a quick tour of dbt today! There is no associated project, but instead you will get a sense of how these tools work.

dbt	
Developer(s)	dbt-Labs
Initial release	December 3, 2021; 23 months ago
Stable release	1.6.5 / October 2, 2023; 35 days ago [1]
Repository	github.com/dbt-labs/dbt-core ↗
Written in	Python
Operating system	Microsoft Windows, macOS, Linux
Available in	Python
Type	Data analytics, data management
License	Apache License 2.0
Website	docs.getdbt.com ↗

[[wikipedia](#)]

a dbt project

A dbt project has several key components.

models: SQL transformations

- `models/*.sql`: 1 query per file

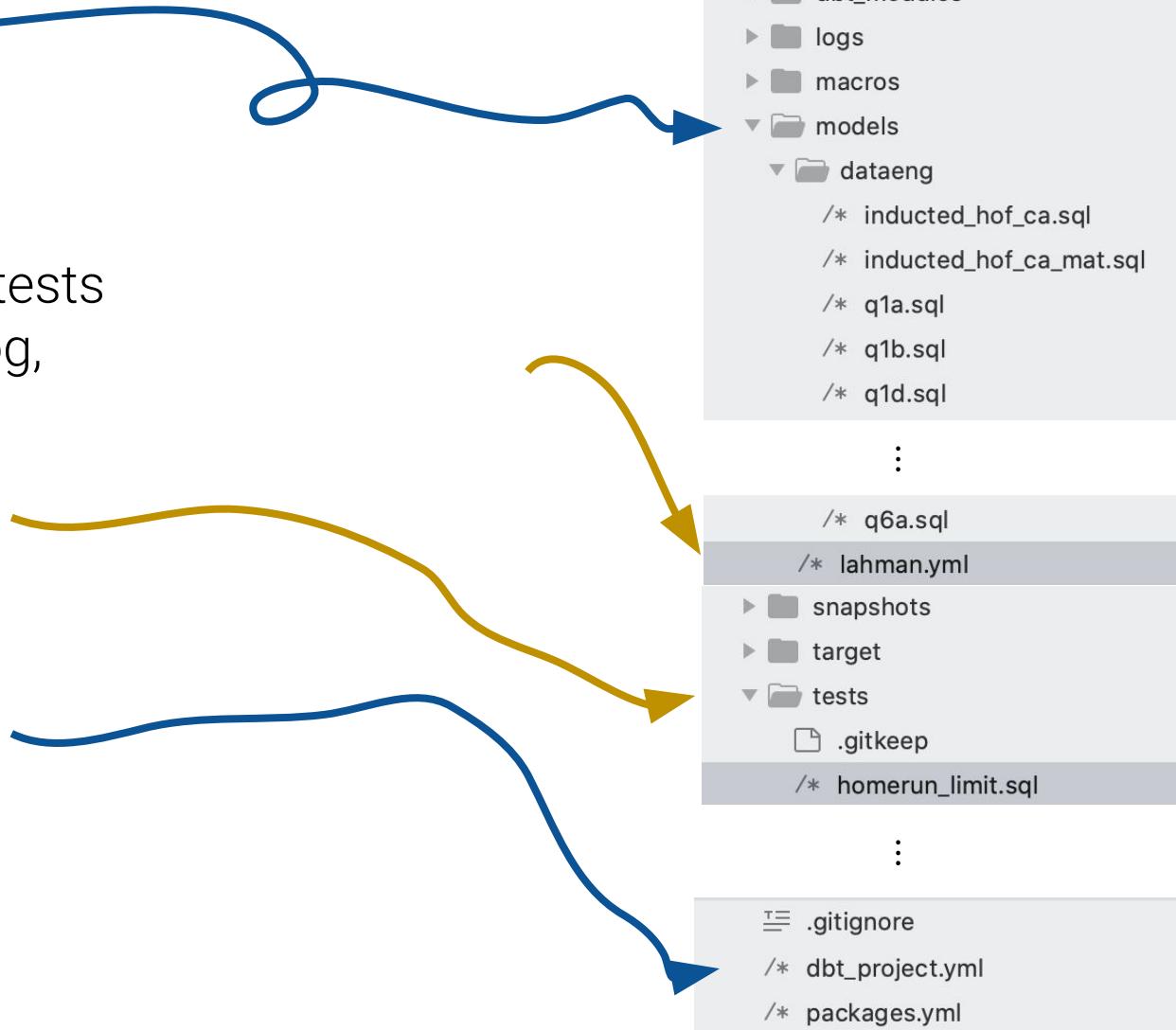
tests

- `models/*.yml` schema integrity tests
 - specified in YAML (JSON analog, but more readable)
- `tests/*.sql`: 1 query per file
 - Passes if it returns 0 rows

`dbt_project.yml`

- required project configuration file, with lots of boilerplate

`models/*.yml` also contains optional metadata



dbt Models

Lecture 23, Data 101/Info 258 Spring 2024

Automating Data Operations
dbt Overview
dbt Models
dbt Tests
Spark as a Workflow System
Pub-Sub and MQs
Workflow Systems

Running models in dbt is dbt run

```
[baseball]> dbt run
Running with dbt 0.19.1
Found 14 models, 25 tests, 0 snapshots, 0 analyses, 309 macros, 0 operations, 0
seed files, 30 sources, 0 exposures

20:23:37 | Concurrency: 1 threads (target='dev')
20:23:37 |
20:23:37 | 1 of 14 START view model jmh.inducted_hof_ca.....
[RUN]
20:23:37 | 1 of 14 OK created view mode
[CREATE VIEW in 0.06s]
20:23:37 | 2 of 14 START table model jm
[RUN]
20:23:37 | 2 of 14 OK created table mod
[SELECT 13 in 0.07s]
20:23:37 | 3 of 14 START view model jmh
[RUN]
20:23:37 | 3 of 14 OK created view mode
[CREATE VIEW in 0.03s]
20:23:37 | 4 of 14 START view model jmh
[RUN]
20:23:37 | 4 of 14 OK created view mode
[CREATE VIEW in 0.03s]
```

Models can reference
models to create a
transformation pipeline.

Several materializations of models:

- view (default)
- table
- incremental

```
baseball -- bash -- 80x24
20:23:37 | 5 of 14 START incremental model jmh.q3a_fulljoin.....
[RUN]
20:23:37 | 5 of 14 OK created incremental model jmh.q3a_fulljoin.....
[INSERT 0 0 in 0.10s]
20:23:37 | 6 of 14 START incremental model jmh.q3a_innerjoin.....
[RUN]
20:23:37 | 6 of 14 OK created incremental model jmh.q3a_innerjoin.....
[INSERT 0 0 in 0.07s]
20:23:37 | 7 of 14 START incremental mode
[RUN]
20:23:37 | 7 of 14 OK created incremental mode
[INSERT 0 0 in 0.07s]
20:23:37 | 8 of 14 START incremental mode
[RUN]
20:23:37 | 8 of 14 OK created incremental mode
[INSERT 0 0 in 0.06s]
20:23:37 | 9 of 14 START view model jmh.q5a.....
[RUN]
20:23:37 | 9 of 14 OK created view model jmh.q5a.....
[CREATE VIEW in 0.03s]
20:23:37 | 10 of 14 START view model jmh.q6a.....
[RUN]
20:23:37 | 10 of 14 OK created view model jmh.q6a.....
[CREATE VIEW in 0.03s]
20:23:37 | 11 of 14 START view model jmh.q5a.....
[RUN]
20:23:37 | 11 of 14 OK created view model jmh.q5a.....
[CREATE VIEW in 0.03s]
20:23:37 | 12 of 14 START view model jmh.q6a.....
[RUN]
20:23:37 | 12 of 14 OK created view model jmh.q6a.....
[CREATE VIEW in 0.03s]
20:23:37 | 13 of 14 START view model jmh.q1d.....
[RUN]
20:23:37 | 13 of 14 OK created view model jmh.q1d.....
[CREATE VIEW in 0.02s]
20:23:37 | 14 of 14 START view model jmh.q1e.....
[RUN]
20:23:37 | 14 of 14 OK created view model jmh.q1e.....
[CREATE VIEW in 0.02s]
20:23:37 |
20:23:37 | Finished running 9 view models, 1 table model, 4 incremental models i
n 1.98s.

Completed successfully

Done. PASS=14 WARN=0 ERROR=0 SKIP=0 TOTAL=14
baseball]>
```

(1/2) Model Materialization: View

Simple model in q1a.sql

- Specifies a derived result named q1a

Note: By default, results are materialized as **views** (b/c transformations pipelined)

The image shows a software interface with two main panes. On the left is a 'FOLDERS' tree view:

- baseball
- analysis
- data
- dbt_modules
- logs
- macros
- models
 - dataeng
 - /* induced_hof_ca.sql
 - /* induced_hof_ca_mat.sql
 - /* q1a.sql

On the right is a code editor window titled 'q1a.sql' containing the following SQL query:

```
1 SELECT namefirst, namelast, p.playerid, yearid
2 FROM people AS p,
3      halloffame AS hof
4 WHERE
5 AND
```

(2/2) Model Materialization: View

Simple model in q1a.sql

- Specifies a derived result named q1a

Note: By default, results are materialized as **views** (b/c transformations pipelined)

Source references in q1a.sql

- Specifies a derived result named q1b (a view, by default)

Note the jinja macro in curly braces!

```
{{ source('baseball', 'people') }}
```

- This still references the **people** table.
- However, now it references whatever people table is specified in model/*.yaml configs.

Source references provide nice hygiene via a **level of indirection** so that these models can be retargeted to other databases or schemas as needed.

The screenshot shows a file tree on the left with the following structure:

- FOLDERS
 - baseball
 - analysis
 - data
 - dbt_modules
 - logs
 - macros
 - models
 - dataeng
 - /* inducted_hof_ca.sql
 - /* inducted_hof_ca_mat.sql
 - /* q1a.sql

The file q1a.sql contains the following SQL code:

```
1 SELECT namefirst, namelast, p.playerid, yearid
2 FROM people AS p,
3 halloffame AS hof
4 WHERE
5 AND
```

The screenshot shows the file q1b.sql with the following content:

```
1 SELECT namefirst, namelast, p.playerid, yearid
2 FROM {{ source('baseball', 'people') }} AS p,
3 (SELECT * FROM {{ source('baseball', 'halloffame') }}) AS hof
4 WHERE
5
```

Side note: the yaml file

```
{{ source('baseball', 'people') }}
```

Source references in q1a.sql

- Specifies a derived result named q1b (a view, by default)

Note the jinja macro in curly braces!

```
{{ source('baseball', 'people') }}
```

- This still references the **people** table.
- However, now it references whatever people table is specified in model/*.yaml configs.

Source references provide nice hygiene via a **level of indirection** so that these models can be retargeted to other databases or schemas as needed.

The diagram illustrates the flow of source references. A yellow arrow points from the Jinja macro {{ source('baseball', 'people') }} in q1a.sql to the 'sources' section of the lahman.yaml file. Another yellow arrow points from the 'sources' section to the 'people' entry in the same file. A third yellow arrow points from the 'people' entry to the 'name: people' line in the lahman.yaml file. A fourth yellow arrow points from the 'name: people' line to the 'people' entry in the packages.yml file at the bottom. A fifth yellow arrow points from the 'people' entry in packages.yml to the 'people' entry in lahman.yaml. A sixth yellow arrow points from the 'people' entry in lahman.yaml back up to the 'sources' section. A seventh yellow arrow points from the 'sources' section back up to the Jinja macro in q1a.sql. A eighth yellow arrow points from the 'sources' section to the 'lahman.yml' file at the top right.

```
version: 2
sources:
  - name: baseball
    database: baseball
    schema: public
    tables:
      - name: tablename
      - name: allstarfull
      - name: alt
      - name: appearances
      - name: awardsmanagers
      - name: awardsplayers
      - name: awardssharemanagers
      - name: awardsshareplayers
      - name: batting
      - name: battingpost
      - name: teams
      - name: collegeplaying
      - name: fielding
      - name: fieldingof
      - name: fieldingpost
      - name: halloffame
      - name: homegames
      - name: managers
      - name: managershalf
      - name: master
      - name: parks
      - name: people
      - name: pitching
      - name: pitchingpost
      - name: salaries
      - name: schools
      - name: seriespost
      - name: states
      - name: teamsfranchises
      - name: teamshalf
  /* baseball
  analysis
  data
  dbt_modules
  logs
  macros
  models
  dataeng
  /* inducted_hof_ca.sql
  /* inducted_hof_ca_mat.sql
  /* q1a.sql
  /* q1b.sql
  /* q1d.sql
  /* q1e.sql
  /* q3a_fulljoin.sql
  /* q3a_innerjoin.sql
  /* q3a_leftjoin.sql
  /* q3a_rightjoin.sql
  /* q4a.sql
  /* q4e.sql
  /* q5a.sql
  /* q6a.sql
  /* lahman.yml
  snapshots
  target
  tests
  gitignore
  dbt_project.yml
  /* packages.yml
```

Model Materialization: Table

Sometimes, we may want to specify that results are materialized as **tables**.

- Also specify this as Jinja macros!

When we dbt run this project:

- `inducted_hof_ca_mat` becomes a table, not a view.
- Note: tables are **recomputed** every time we dbt run the project!

FOLDERS

- baseball
 - analysis
 - data
 - dbt_modules
 - logs
 - macros
- models
 - dataeng
 - /* inducted_hof_ca.sql
 - /* inducted_hof_ca_mat.sql
 - /* q1a.sql
 - /* q1b.sql
 - /* q1d.sql

```
inducted_hof_ca_mat.sql x
1 {{ config(materialized='table') }}
2
3 SELECT DISTINCT namefirst, namelast, p.playerid, s.schoolid, hof.yearid
4   FROM {{ source('baseball', 'people') }} AS p,
5        {{ source('baseball', 'halloffame') }} AS hof,
6        {{ source('baseball', 'collegeplaying') }} AS cp,
7        {{ source('baseball', 'schools') }} AS s
8 WHERE
9
10
11
12
13
```

```
q1a.sql x
1 SELECT namefirst, namelast, p.playerid, yearid
2   FROM people AS p,
3        halloffame AS hof
4 WHERE
5   AND
```

simple model in `q1a.sql`,
materialized as a view by default

table model in
`inducted_hof_ca_m
at.sql`, explicitly
materialized as a table

q3a_innerjoin becomes a table on first run

Incremental models are built as tables in your `data.warehouse`. The first time a model is run, the table is built by transforming *all* rows of source data. On subsequent runs, dbt transforms *only* the rows in your source data that you tell dbt to filter for, inserting them into the target table which is the table that has already been built.

Often, the rows you filter for on an incremental run will be the rows in your source data that have been created or updated since the last time dbt ran. As such, on each dbt run, your model gets built incrementally.

Using an incremental model limits the amount of data that needs to be transformed, vastly reducing the runtime of your transformations. This improves warehouse performance and reduces compute costs.

```
1 {{ config(materialized='incremental') }}
2
3 SELECT p.*,
4
5   FROM
6     {{ source('baseball', 'collegeplaying') }} AS cp
7
8   {% if is_incremental() %}
9     WHERE cp.yearid > (SELECT
10       MAX(yearid)
11       FROM {{ this }})
12   {% endif %}
```

On subsequent runs, lines 10 & 11 are added to limit the work

Incremental models provide a facility for simple “**materialized view maintenance**” **outside** the database.

q3a_innerjoin becomes a table on first run

Incremental models are built as tables in your `data.warehouse`. The first time a model is run, the table is built by transforming *all* rows of source data. On subsequent runs, dbt transforms *only* the rows in your source data that you tell dbt to filter for, inserting them into the target table which is the table that has already been built.

Often, the rows you filter for on an incremental run will be the rows in your source data that have been created or updated since the last time dbt ran. As such, on each dbt run, your model gets built incrementally.

Using an incremental model limits the amount of data that needs to be transformed, vastly reducing the runtime of your transformations. This improves warehouse performance and reduces compute costs.

Incremental models provide a facility for simple “**materialized view maintenance**” **outside** the database.

- Only handles inserts and “upserts” (i.e., replace tuples using unique key)
- No delete handling, no general update handling

On subsequent runs, lines 10 & 11 are added to limit the work

Principle from early lectures: “**selection pushdown** is good”...!!

- ...But it depends on the DBMS you’re using ([docs: When should I use an incremental model?](#))
- Up to you to figure out if it’s a performance win 😊

We've seen how with `{ { source('baseball', 'people') } }` models can reference other tables with a layer of indirection.

Models can also **reference other models** with `ref`:

```
{ { ref('inducted_hof_ca') } }
```

Specifying `ref` dependencies is how dbt models **implicitly specify a pipeline of models** querying models.

Note: cannot have cycles of ref dependencies, i.e., the graph representing all refs must be a DAG

The most important function in dbt is `ref()`; it's impossible to build even moderately complex models without it. `ref()` is how you reference one model within another. This is a very common behavior, as typically models are built to be "stacked" on top of one another. Here is how this looks in practice:

A screenshot of a terminal window illustrating the use of `ref()`. On the left, a file tree shows a directory structure with folders like baseball, analysis, data, dbt_modules, logs, macros, and models. Inside the models folder, there is a subfolder named dataeng containing several SQL files: `inducted_hof_ca.sql`, `inducted_hof_ca_mat.sql`, `q1a.sql`, `q1b.sql`, and `q1d.sql`. On the right, a code editor window displays a SQL file named `q1d.sql`. The code in the editor is:

```
1 SELECT
2 FROM {{ ref('inducted_hof_ca') }}
```

Two blue arrows point from the text `ref('inducted_hof_ca')` in the code editor back to the `inducted_hof_ca` file in the file tree on the left, demonstrating that the `ref()` call refers to the specific file in the dataeng folder.

dbt Tests

Lecture 23, Data 101/Info 258 Spring 2024

Automating Data Operations
dbt Overview
dbt Models
dbt Tests
Spark as a Workflow System
Pub-Sub and MQs
Workflow Systems

a dbt project

A dbt project has several key components.

models: SQL transformations

- `models/*.sql`: 1 query per file

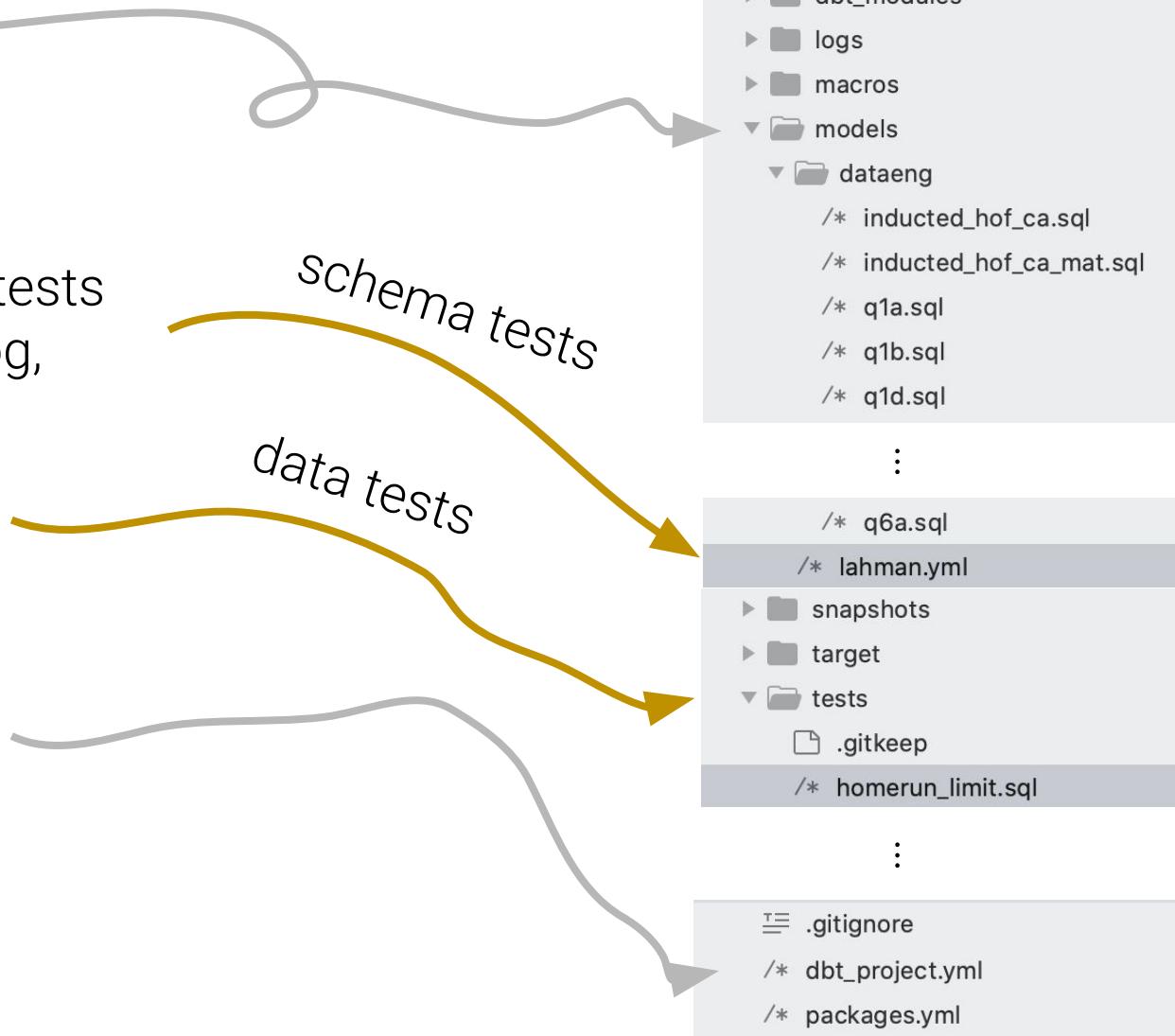
tests

- `models/*.yml` schema integrity tests
 - specified in YAML (JSON analog, but more readable)
- `tests/*.sql`: 1 query per file
 - Passes if it returns 0 rows

`dbt_project.yml`

- required project configuration file, with lots of boilerplate

`models/*.yml` also contains optional metadata



Running Tests

```
baseball]> dbt test
Running with dbt=0.19.1
Found 14 models, 25 tests, 0 snapshots, 0 analyses, 309 macros, 0 operations, 0
seed files, 30 sources, 0 exposures

20:26:36 | Concurrency: 1 threads (target='dev')
20:26:36 |
20:26:36 | 1 of 25 START test homerun_limit.....
[RUN]
20:26:36 | 1 of 25 FAIL 1 homerun_limit.....
[FAIL 1 in 0.04s]
20:26:36 | 2 of 25 START test not_null_inducted_hof_ca_mat_namefirst.....
[RUN]
20:26:36 | 2 of 25 PASS not_null_inducted_hof_ca_mat_namefirst.....
[PASS in 0.01s]
20:26:36 | 3 of 25 START test not_null_inducted_hof_ca_mat_namelast.....
[RUN]
20:26:36 | 3 of 25 PASS not_null_inducted_hof_ca_mat_namelast.....
[PASS in 0.01s]
20:26:36 | 4 of 25 START test not_null_inducted_hof_ca_mat_playerid.....
[RUN]
20:26:36 | 4 of 25 PASS not_null_inducted_hof_ca_mat_playerid.....
[PASS in 0.01s]
20:26:36 | 5 of 25 START test not_null_inducted_hof_ca_mat_schoolid.....
```

```
baseball]> dbt test
Running with dbt=0.19.1
Found 14 models, 25 tests, 0 snapshots, 0 analyses, 309 macros, 0 operations, 0
seed files, 30 sources, 0 exposures

20:26:37 |
20:26:37 | Finished running 25 tests in 1.88s.

Completed with 3 errors and 0 warnings:

Failure in test homerun_limit (tests/homerun_limit.sql)
  Got 1 result, expected 0.

  compiled SQL at target/compiled/baseball/tests/homerun_limit.sql

Failure in test unique_inducted_hof_ca_mat_playerid (models/lahman.yml)
  Got 2 results, expected 0.

  compiled SQL at target/compiled/baseball/models/lahman.yml/schema_test/unique_
inducted_hof_ca_mat_playerid.sql

Failure in test unique_inducted_hof_ca_playerid (models/lahman.yml)
  Got 2 results, expected 0.

  compiled SQL at target/compiled/baseball/models/lahman.yml/schema_test/unique_
inducted_hof_ca_playerid.sql

Done. PASS=22 WARN=0 ERROR=3 SKIP=0 TOTAL=25
baseball]>
```

Running tests in dbt is dbt test

```
[baseball]: dbt test
Running with dbt 0.10.1
Found 14 models, 25 tests, 0 snapshots, 0 analyses, 309 macros, 0 operations, 0
seed files, 30 sources, 0 exposures

20:26:36 | Concurrency: 1 threads (target='dev')
20:26:36 |
20:26:36 | 1 of 25 START test homerun_limit.....
[RUN]
20:26:36 | 1 of 25 FAIL 1 homerun_limit.....
[FAIL 1 in 0.04s]
20:26:36 | 2 of 25 START test not_null_inducted_hof_ca_mat_na
[RUN]
20:26:36 | 2 of 25 PASS not_null_inducted_hof_ca_mat_namefir
[PASS in 0.01s]
20:26:36 | 3 of 25 START test not_null_inducted_hof_ca_mat_na
[RUN]
20:26:36 | 3 of 25 PASS not_null_inducted_hof_ca_mat_namelast
[PASS in 0.01s]
20:26:36 | 4 of 25 START test not_null_inducted_hof_ca_mat_pl
[RUN]
20:26:36 | 4 of 25 PASS not_null_inducted_hof_ca_mat_playerid
[PASS in 0.01s]
20:26:36 | 5 of 25 START test not_null_inducted_hof_ca_mat_sc
```

```
baseball] 20:26:37 |
baseball] 20:26:37 | Finished running 25 tests in 1.88s.
baseball] Completed with 3 errors and 0 warnings:
baseball]   Failure in test homerun_limit (tests/homerun_limit.sql)
baseball]     Got 1 result, expected 0.
baseball]     compiled SQL at target/compiled/baseball/tests/homerun_limit.sql
baseball]   Failure in test unique_inducted_hof_ca_mat_playerid (models/lahman.yml)
baseball]     Got 2 results, expected 0.
baseball]     compiled SQL at target/compiled/baseball/models/lahman.yml/schema_test/unique_
inducted_hof_ca_mat_playerid.sql
baseball]   Failure in test unique_inducted_hof_ca_playerid (models/lahman.yml)
baseball]     Got 2 results, expected 0.
baseball]     compiled SQL at target/compiled/baseball/models/lahman.yml/schema_test/unique_
inducted_hof_ca_playerid.sql
baseball] Done. PASS=22 WARN=0 ERROR=3 SKIP=0 TOTAL=25
baseball]>
```

Schema Tests

models/*.yml schema integrity tests

- specified in YAML (JSON analog, but more readable)

Like SQL DDL constraints, but mid-pipeline!

- unique, not_null
- accepted_values
- relationships

Caveats:

- Single-column by default!
- ...but side approaches available to test composite columns
 - [“Can I test the uniqueness of two columns?” docs](#)

The screenshot shows a code editor with a dark theme. On the left is a file tree (FOLDERS) containing:

- baseball
- analysis
- data
- dbt_modules
- logs
- macros
- models
- dataeng
 - /* induced_hof_ca.sql
 - /* induced_hof_ca_mat.sql
 - /* q1.sql
 - /* q1b.sql
 - /* q1d.sql
 - /* q1e.sql
 - /* q3a_fulljoin.sql
 - /* q3a_innerjoin.sql
 - /* q3a_leftjoin.sql
 - /* q3a_rightjoin.sql
 - /* q4a.sql
 - /* q4e.sql
 - /* q5a.sql
 - /* q6a.sql
- /* lahman.yml
- snapshots
- target
- tests
- .gitignore
- /* dbt_project.yml
- /* packages.yml
- README.md

The right pane shows the contents of the `lahman.yml` file:

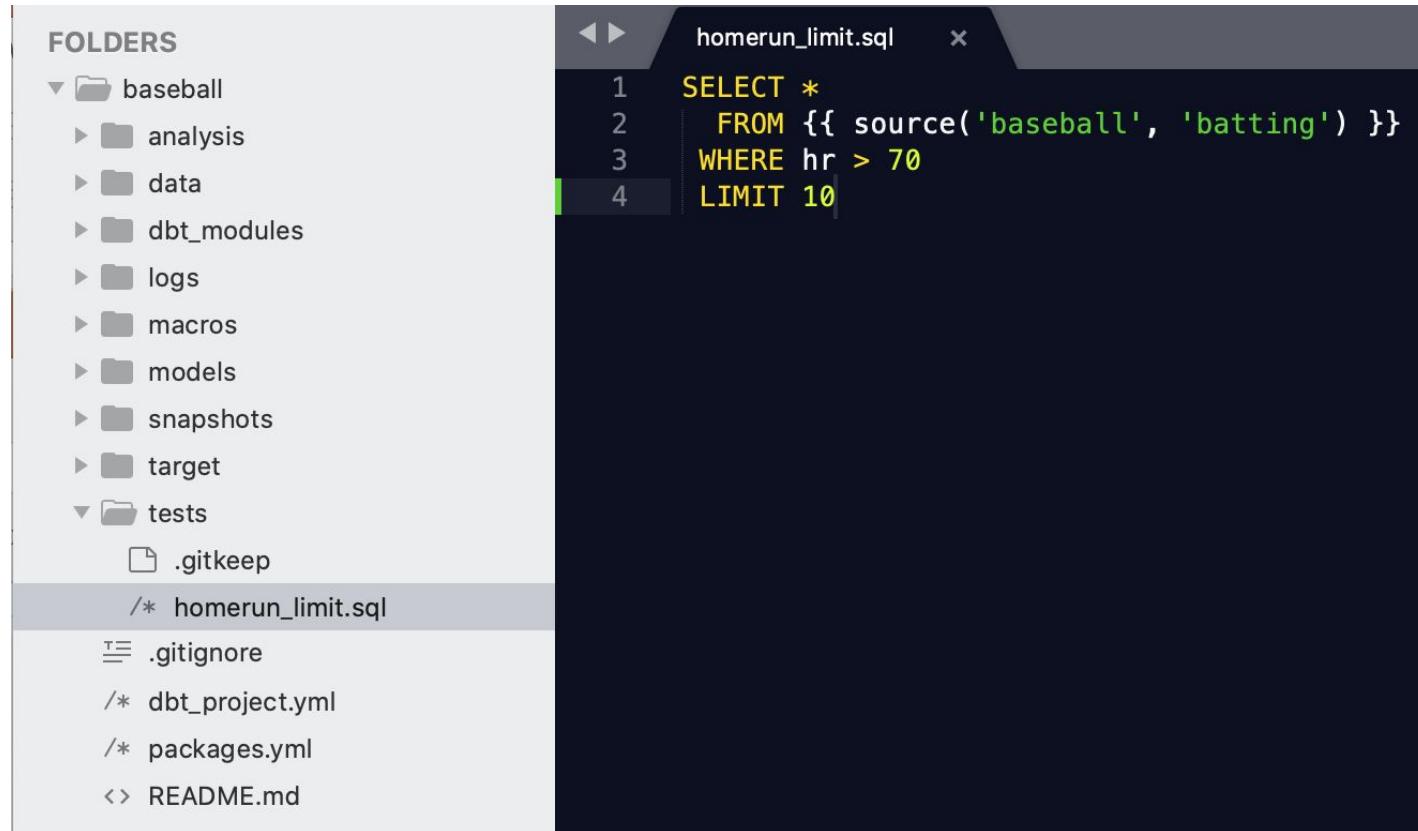
```
models:  
  - name: induced_hof_ca  
    columns:  
      - name: namefirst  
        tests:  
          - not_null  
      - name: namelast  
        tests:  
          - not_null  
      - name: playerid  
        tests:  
          - unique  
          - not_null  
        relationships:  
          to: source('baseball', 'people')  
          field: 'playerid'  
  - name: schoolid  
    tests:  
      - not_null  
      - name: yearid  
        tests:  
          - not_null  
  - name: induced_hof_ca_mat  
    columns:  
      - name: namefirst  
        tests:  
          - not_null  
      - name: namelast  
        tests:  
          - not_null  
      - name: playerid  
        tests:  
          - unique  
          - not_null  
        relationships:  
          to: source('baseball', 'people')  
          field: 'playerid'  
  - name: schoolid  
    tests:  
      - not_null  
      - name: yearid  
        tests:  
          - not_null
```

Yellow arrows point from the text "Like SQL DDL constraints, but mid-pipeline!" and the list item "unique, not_null" to the `tests:` and `relationships:` sections in the `lahman.yml` code.

tests/*.sql: 1 query per file

Yes—these are just plain old .sql files.

- Any check you like!
- But keep it quickish.
It's supposed to be a quick test!
- **Pass: return 0 rows**
- **Fail:** return something “small”
(of your own design)



The screenshot shows a code editor interface with a sidebar and a main panel. The sidebar on the left is titled "FOLDERS" and lists several directories: baseball, analysis, data, dbt_modules, logs, macros, models, snapshots, target, and tests. Inside the tests directory, there are files named .gitkeep, homerun_limit.sql (which is currently selected), .gitignore, dbt_project.yml, packages.yml, and README.md. The main panel on the right displays the contents of the homerun_limit.sql file. The code is a single SQL query:

```
1 SELECT *  
2   FROM {{ source('baseball', 'batting') }}  
3 WHERE hr > 70  
4 LIMIT 10
```

Continuous Integration (CI): the practice of merging all developers' working copies to a shared mainline several times a day [[wikipedia](#)]

Make sure every code checkin forces **dbt test** to run! Example workflows:

- Force a test at client prior to github commit with commit hooks [[link](#)]
- Or, pay for dbt labs CI integration of dbt with github
- Or, you can use a github-integrated CI tool like Travis/Jenkins

In all cases, you need to ensure database connectivity.

dbt test in DataOps/CI Testing

Continuous Integration (CI): the practice of merging all developers' working copies to a shared mainline several times a day [[wikipedia](#)]

Make sure every code checkin forces `dbt test` to run! Example workflows:

- Force a test at client prior to github commit with commit hooks [[link](#)]
- Or, pay for dbt labs CI integration of dbt with github
- Or, you can use a github-integrated CI tool like Travis/Jenkins

In all cases, you need to ensure database connectivity.

Standard CI/CD (continuous development) has code go through 2 or 3 different deployments.

- **Dev** (Development) → **Test** (or CI) → **Prod** (Production)

In a **data warehouse**, typically done with different schemas or databases for each stage.

- Specified via outputs and target in the `~/.dbt/profiles.yml` file.
- Queries then source and generate outputs in the appropriate schema/database
- Need to keep track of which version of the code (git tag) is associated with each deployment.

Two Slides on Spark/Algebra

Lecture 23, Data 101/Info 258 Spring 2024

Automating Data Operations
dbt Overview
dbt Models
dbt Tests
Spark as a Workflow System
Pub-Sub and MQs
Workflow Systems

3. How do we automate the work?

Data pipelines are automated with more kinds of systems! 😊

Workflow Systems: manage processes (human or software)

- With process dependencies (order of activity)
- Activiti, Airflow, BPL, Luigi, ...

Dataflow Systems: manage data movement through operations (software)

- Handle data dependencies (inputs/outputs)
- Spark
- dbt + SQL
- Flink, Dataflow, etc.

Messaging/Queueing/Eventing: hand off data reliably from one system to another

- Kafka, RabbitMQ, AWS SQS, MSMQ, MQSeries, ...

Practically: Many times you will use combinations of these, and it can be a matter of taste/experience which to use where.

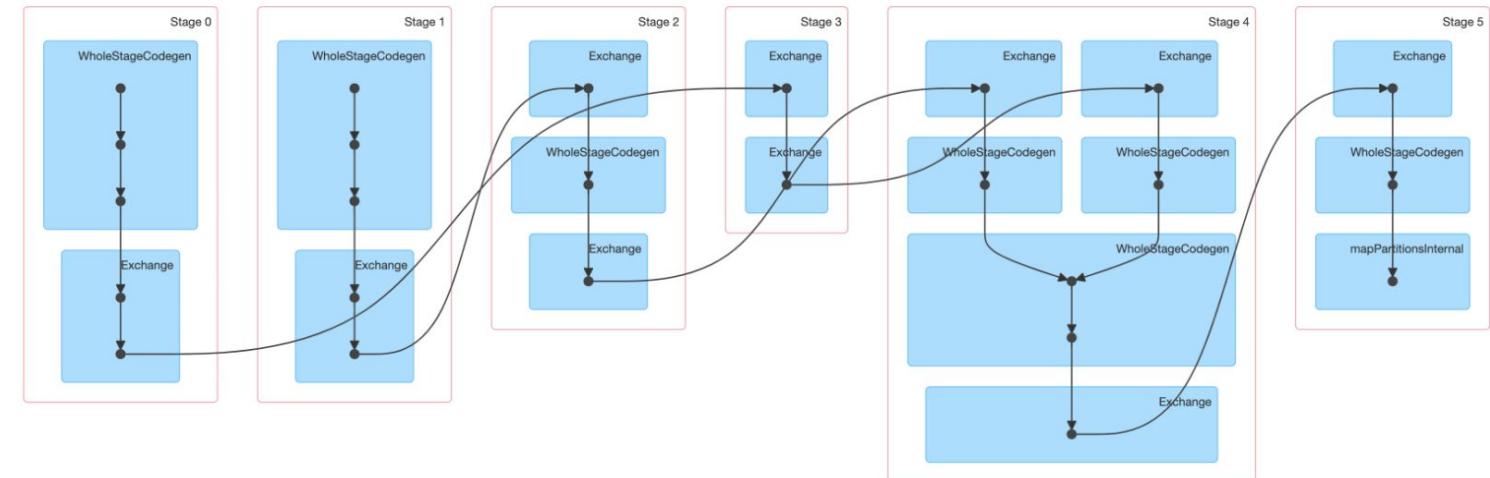
- You'll learn it "on the street..."
- Or, you'll subscribe to company "religion" and be part of many debates

Dataflow: Spark, Google Dataflow, etc.

Like Pandas, lets you define dataflows functionally.

Scala

```
1 val ds1 = spark.range(1, 10000000)
2 val ds2 = spark.range(1, 10000000, 2)
3 val ds3 = ds1.repartition(7)
4 val ds4 = ds2.repartition(9)
5 val ds5 = ds3.selectExpr("id * 5 as id")
6 val joined = ds5.join(ds4, "id")
7 val sum = joined.selectExpr("sum(id)")
8 sum.show()
```



Spark “RDDs” are akin to Database Views.

- A record of the code needed to compute results from data on demand
- But in a dataflow format akin to a query plan in a database
 - I.e., EXPLAIN in Postgres



Cloud
DataFlow

Why choose imperative/algebraic forms over SQL? Or why not?

These is an **incremental** process to writing code in algebraic forms (Rel Alg, Spark, Pandas).

- Build up your complex transformation pipelines one step at a time.
- Good for exploratory thinking, debugging, incremental complexity.

By contrast, SQL wants us to “**declare**” our outputs.

- Stepwise reasoning is harder, though CTEs/WITH and views do help.
- But syntax is unwieldy and CTEs are inconvenient to debug.

However, we'll claim that these distinctions are **form, not function**.

- SQL can do about anything that algebras can do!
- Standard, clean semantics, portable—even compared with modern Spark
- Also: Transactional support, lots of SQL tools, etc.

dbt is a tool to build in SQL “ergonomics” for writing complex Transformation scripts.

Three Slides on Pub-Sub and Message Queues

Lecture 23, Data 101/Info 258 Spring 2024

Automating Data Operations
dbt Overview
dbt Models
dbt Tests
Spark as a Workflow System
Pub-Sub and MQs
Workflow Systems

3. How do we automate the work?

Data pipelines are automated with more kinds of systems! 😊

Workflow Systems: manage processes (human or software)

- With process dependencies (order of activity)
- Activiti, Airflow, BPL, Luigi, ...

Dataflow Systems: manage data movement through operations (software)

- Handle data dependencies (inputs/outputs)
- Spark
- dbt + SQL
- Flink, Dataflow, etc.

These systems address
Q1: How data arrives.



Messaging/Queueing/Eventing: hand off data reliably from one system to another

- Kafka, RabbitMQ, AWS SQS, MSMQ, MQSeries, ...

Practically: Many times you will use combinations of these, and it can be a matter of taste/experience which to use where.

- You'll learn it "on the street..."
- Or, you'll subscribe to company "religion" and be part of many debates

(1/3) Pub-Sub (publish-subscribe) and Message Queue Software Systems

Software solutions to the data handoff b/t **producer** and **consumer processes** in a pipeline.



e.g., **transactional database**
behind a website

- Needs to quickly handle customer/user access

e.g., **data warehouse** for monthly analytics

- Needs high-capacity and durable storage

e.g., **employee “beeper” system** for emergencies

- Needs low latency for QA alerts

(2/3) Pub-Sub (publish-subscribe) and Message Queue Software Systems

Software solutions to the data handoff b/t **producer** and **consumer processes** in a pipeline.

- **Asynchronous management** (analogy: checking mail in a mailbox)
 - A data storage problem: asynchrony requires persistent storage to buffer unprocessed msgs
 - **Producer**: needs persistent storage; **Consumer**: needs ability to check at any time
- **Reliability**
 - **Producer**: make sure that message has been queued **durably**
 - **Consumer**: receive every message, and handle it to completion **atomically**

These requirements translates into an ideal: **exactly-once delivery**.

- Atomic enqueue, Atomic dequeue-and-process
- Queue ACK: durable receipt from producer
- Client ACK: commit of message-handling transaction



see: distributed transactions
(not covered)

Realistically, in many cases: **at least-once delivery**

- Consumer (client) takes responsibility to ensure **idempotent** message handling (i.e., that doing something multiple times is the same as doing it once)

(2/3) Pub-Sub (publish-subscribe) and Message Queue Software Systems

Publish-Subscribe (aka message bus)

- Apache **Kafka** (originally dev'ed @LinkedIn), now owned/managed by Confluent
- even earlier: TIBCO, 1997 (The Information Bus Company)
- Consumers declare persistent **subscriptions** to topics like “Updates-to-table-X”
- SW handles A **stream of messages**, each labeled with ≥ 1 topics, which are published to subscribers
- Kafka: stores a **durable log** of a window of past events
- Optimized for **high-throughput data volumes**, e.g., web server data



Persistent Message Queue

- RabbitMQ
- Register “**routes**” for forwarding messages to consumers
- **Stream** of messages to be routed
- Consumers acknowledge handoff
- “Exactly once” delivery
- **No big log** – each message deleted
- Optimized for **low-latency** (speed per message)



Two Slides on Workflow Systems

Lecture 23, Data 101/Info 258 Spring 2024

Automating Data Operations
dbt Overview
dbt Models
dbt Tests
Spark as a Workflow System
Pub-Sub and MQs
Workflow Systems

3. How do we automate the work?

Data pipelines are automated with more kinds of systems! 😊

Workflow Systems: manage processes (human or software)

- With process dependencies (order of activity)
- Activiti, Airflow, BPL, Luigi, ...



Dataflow Systems: manage data movement through operations (software)

- Handle data dependencies (inputs/outputs)
- Spark
- dbt + SQL
- Flink, Dataflow, etc.

Messaging/Queueing/Eventing: hand off data reliably from one system to another

- Kafka, RabbitMQ, AWS SQS, MSMQ, MQSeries, ...

Practically: Many times you will use combinations of these, and it can be a matter of taste/experience which to use where.

- You'll learn it "on the street..."
- Or, you'll subscribe to company "religion" and be part of many debates

(1/2) Workflow Systems

Workflow systems orchestrate a bunch of separate processes or API calls, including but not limited to:

- Control order of invocation
- Programming constructs like conditionals (IF) and loops over system
- Handle (potentially long) asynchrony in request/response
- Allow for steps that involve humans!
- Run reliably over long periods of time, despite failures of components or the workflow machines
 - Often storing the state of the workflow in a SQL DBMS
 - Sometimes guaranteeing transition between steps transactionally
- Provide features to schedule and monitor progress

(1/2) Workflow System: Apache Airflow

A **DAG (directed acyclic graph)** of (ideally idempotent) **tasks** passing small amounts of data.

- DAGs are constructed programmatically in Python
 - Simple syntax for defining tasks and setting up “edges” between them
 - DAG “shape” can be dynamic at runtime
 - Exception logic to fail, skip, reschedule a task

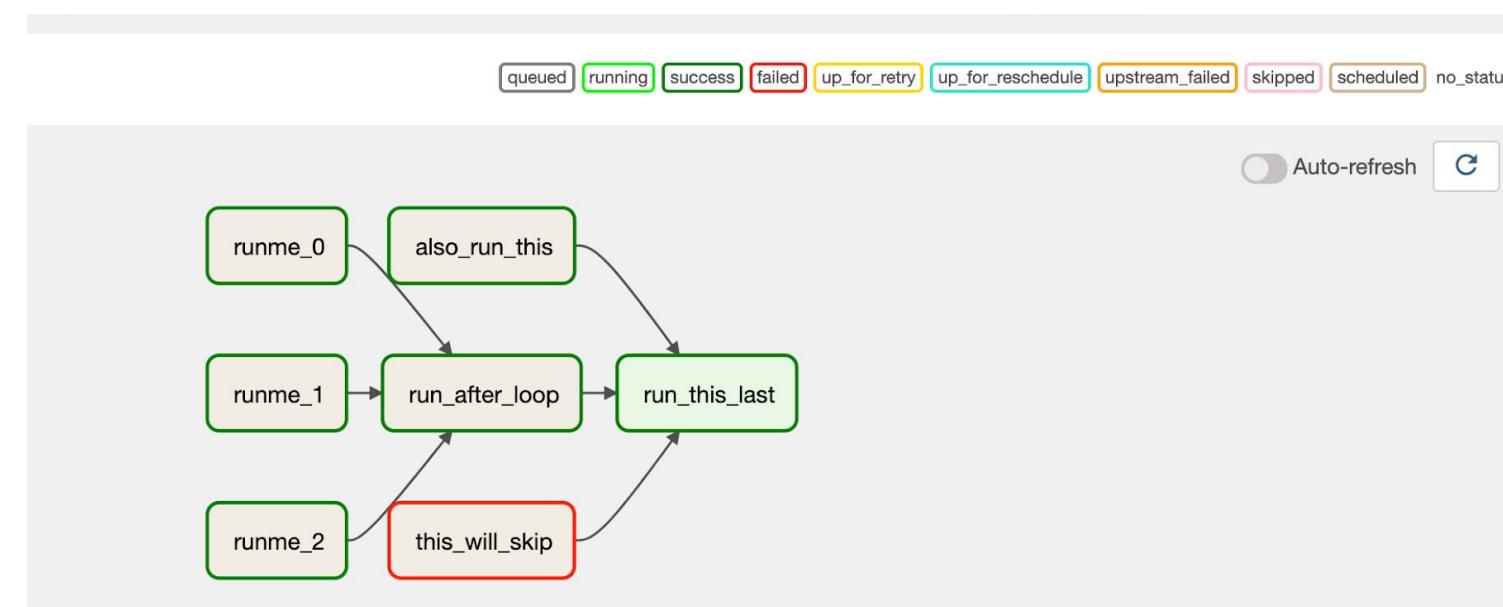
Tasks (Nodes) in the DAG are user-defined.

- Python code, bash calls,
- emails, Slack msgs, API calls, etc.

Scales up via use of message queues

Schedules and tracks Runs

- “Sensors” to check for time passing or data arriving in filesystem or database



[Extra] dbt Essentials

Lecture 23, Data 101/Info 258 Spring 2024

We don't have an associated project this semester on dbt, so these slides are here for your reference in case you do want to build your own dbt project.

These slides cover essential components of dbt that are needed for a functional project.

dbt_project.yml

- Lines 5-7 configure handy metadata
- Line 10 specifies the connection in **~/.dbt/profiles.yml**
- Lines 12-25 are boilerplate for the standard directory hierarchy generated by **dbt init**
- Lines 34-39 provide defaults for our models
 - contained in a database named "baseball"
 - by default, model results are views, and not materialized

The screenshot shows a code editor window with the file 'dbt_project.yml' open. The code is color-coded for syntax. Several sections of the code are highlighted with yellow boxes:

- Line 5: `name: 'baseball'`
- Line 10: `profile: 'baseball'`
- Line 34: `models:`
- Line 36: `+database: baseball`
- Line 37: `# Applies to all files under models/dataeng/`
- Line 38: `dataeng:`
- Line 39: `materialized: view`

```
1  # Name your project! Project names should contain only lowercase characters
2  # and underscores. A good package name should reflect your organization's
3  # name or the intended use of these models
4  # name: 'baseball'
5  name: 'baseball'
6  version: '1.0.0'
7  config-version: 2
8
9  # This setting configures which "profile" dbt uses for this project.
10 profile: 'baseball'
11
12 # These configurations specify where dbt should look for different types of files.
13 # The `source-paths` config, for example, states that models in this project can be
14 # found in the "models/" directory. You probably won't need to change these!
15 source-paths: ["models"]
16 analysis-paths: ["analysis"]
17 test-paths: ["tests"]
18 data-paths: ["data"]
19 macro-paths: ["macros"]
20 snapshot-paths: ["snapshots"]
21
22 target-path: "target" # directory which will store compiled SQL files
23 clean-targets: # directories to be removed by `dbt clean`
24   - "target"
25   - "dbt_modules"
26
27
28 # Configuring models
29 # Full documentation: https://docs.getdbt.com/docs/configuring-models
30
31 # In this config, we tell dbt to build all models in the baseball/ directory
32 # as views. These settings can be overridden in the individual model files
33 # using the `{{ config( ) }}` macro
34 models:
35 baseball:
36   +database: baseball
37   # Applies to all files under models/dataeng/
38   dataeng:
39     materialized: view
```

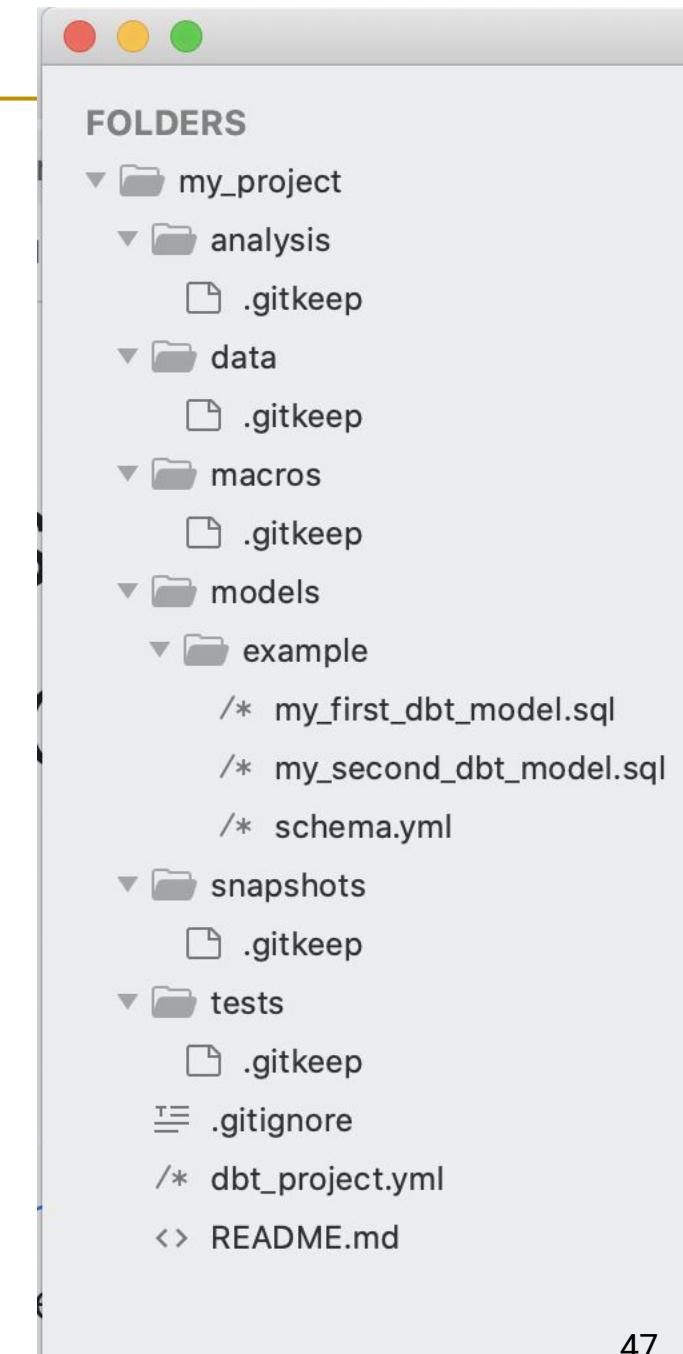
A Basic dbt Repo 3/4

Data: Csv's for "seed data" (e.g. static lookup tables)

Analysis: Extra SQL queries you don't need to run

Macros: String-based macros in the Jinja language

Snapshots: SQL queries to materialize historical data as it's being updated



A Basic dbt Repo 4/4

Also `~/ .dbt/profiles.yml`

- DB connectivity information
- Per dbt “profile”
 - Matches your `dbt_project.yml`



The screenshot shows a code editor window titled "profiles.yml". The file contains YAML configuration for dbt profiles. It defines three profiles: "default", "prod", and "target: dev". The "default" profile is for a PostgreSQL database on localhost with user "jmh" and password "jmh", port 5432, dbname "dbttest_dev", schema "jmh", and threads 1. It includes comments for keepalives_idle, search_path, role, and sslmode. The "prod" profile is identical to "default". The "target: dev" section specifies a target of "dev" and an output profile "dev" which is identical to the "default" profile. A note at the bottom states that the profile needs to match the one in dbt_project.yml.

```
1 default:
2   outputs:
3     dev:
4       type: postgres
5       host: localhost
6       user: jmh
7       password: jmh
8       port: 5432
9       dbname: dbttest_dev
10      schema: jmh
11      threads: 1
12      keepalives_idle: 0 # default 0, indicating the system default
13      # search_path: [optional, override the default postgres search_path]
14      # role: [optional, set the role dbt assumes when executing queries]
15      # sslmode: [optional, set the sslmode used to connect to the database]
16
17 prod:
18   type: postgres
19   host: localhost
20   user: jmh
21   password: jmh
22   port: 5432
23   dbname: dbttest_prod
24   schema: jmh
25   threads: 1
26   keepalives_idle: 0 # default 0, indicating the system default
27   # search_path: [optional, override the default postgres search_path]
28   # role: [optional, set the role dbt assumes when executing queries]
29   # sslmode: [optional, set the sslmode used to connect to the database]
30
31 target: dev
32
33 baseball: # this needs to match the profile: in your dbt_project.yml file
34   target: dev
35   outputs:
36     dev:
37       type: postgres
38       host: localhost
39       user: jmh
40       password: jmh
41       port: 5432
42       dbname: baseball
43       schema: jmh
44       threads: 1
45       keepalives_idle: 0 # default 0, indicating the system default
46       # search_path: [optional, override the default postgres search_path]
47       # role: [optional, set the role dbt assumes when executing queries]
48       # sslmode: [optional, set the sslmode used to connect to the database]
```

Lineage DAG in dbt docs

dbt docs generate
dbt docs serve

Note that there's no notion of "tasks" here, just
data lineage (data "dependencies")

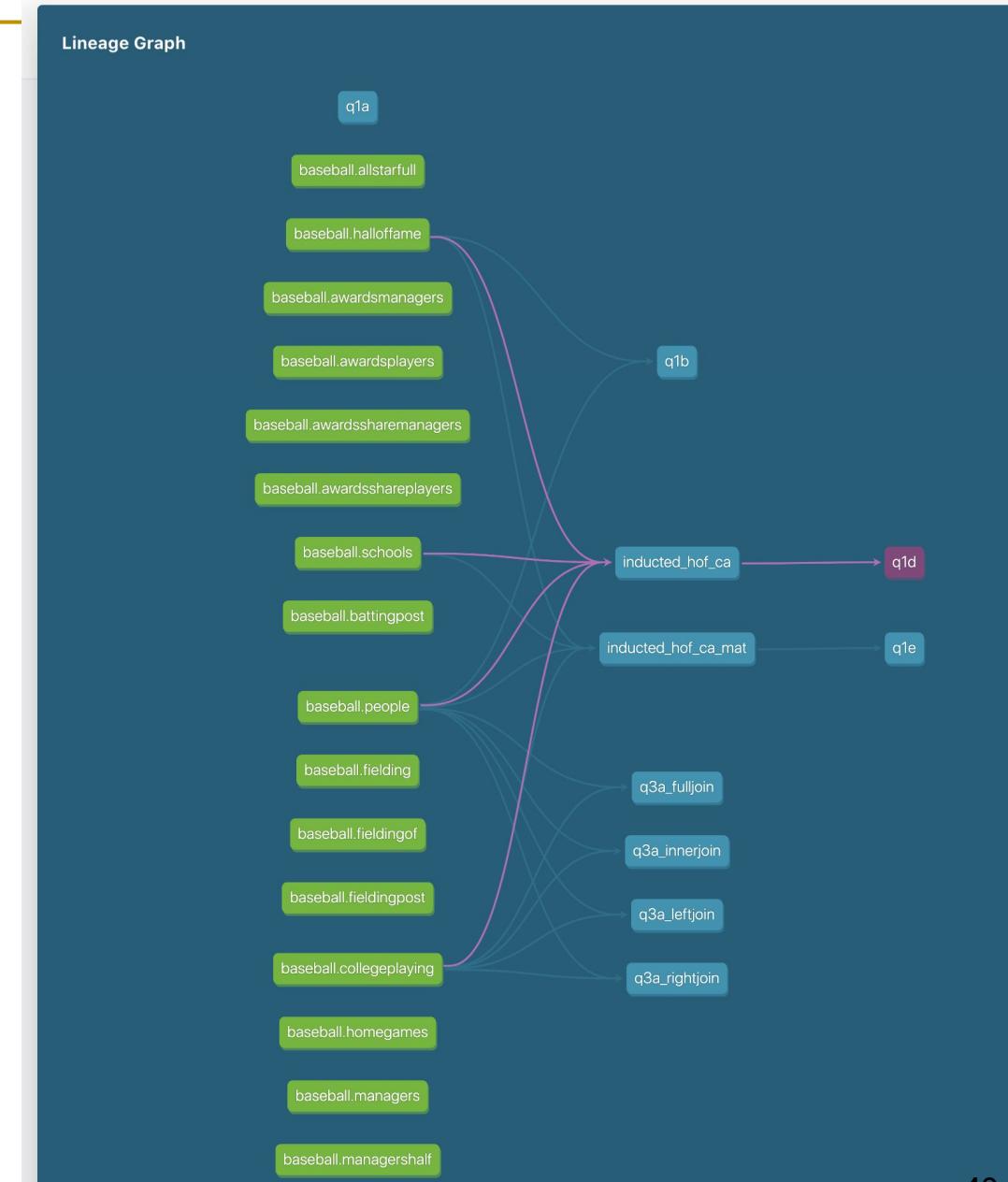
- We dbt run the whole DAG

All non-incremental models are rebuilt from scratch each time

Incremental models capture the notion of "change propagation"

- Though fairly primitive in dbt
- There are custom products that are better at "Change Data Capture" and materialized view maintenance

Other nice stuff in dbt docs, poke around!



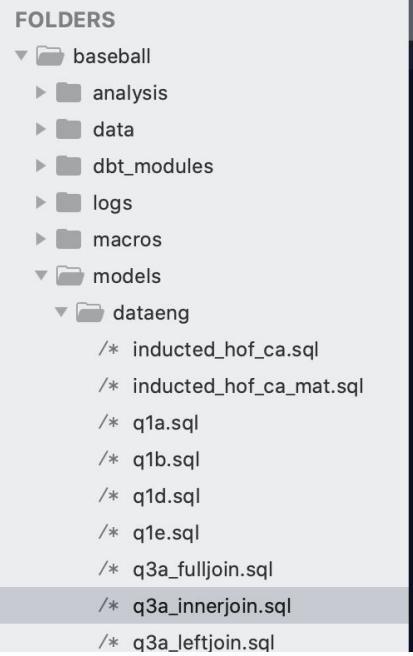
Jinja Macros

Recall that SQL only lets you query rows

- SELECT and FROM lists are “hard-wired”, not dynamic

Jinja can help

- Essentially, parameterized queries via string macros



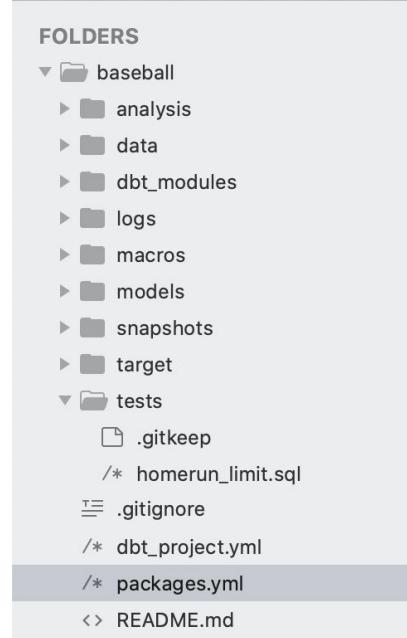
```
q3a_innerjoin.sql x
1 {{ config(materialized='incremental') }}
2
3 SELECT p.*,
4       {{ dbt_utils.star(from=source('baseball', 'collegeplaying'),
5                          except=['playerid']) }}
6   FROM {{ source('baseball', 'people') }} AS p
7     INNER JOIN {{ source('baseball', 'collegeplaying') }} AS cp
8           ON p.playerid = cp.playerid
9 {% if is_incremental() %}
10    WHERE cp.yearid > (SELECT MAX(yearid) FROM {{ this }})
11    AND p.debut > (SELECT MAX(debut) FROM {{ this }})
12 {% endif %}
```

A blunt instrument

- Easy to write queries that don't work or are undesirable
- Keep it simple

Can pull in libraries like dbt_utils from git

- packages.yml at top level
- E.g. star command in the upper right lets you choose which columns you want in the SELECT list more flexibly



```
packages.yml x
1 packages:
2   - git: "https://github.com/fishtown-analytics/dbt-utils.git"
3     revision: 0.6.4
4
```

FOLDERS

- ▼ baseball
- ▶ analysis
- ▶ data
- ▶ dbt_modules
- ▶ logs
- ▶ macros
- ▼ models
 - ▼ dataeng
 - /* inducted_hof_ca.sql
 - /* inducted_hof_ca_mat.sql
 - /* q1a.sql
 - /* q1b.sql
 - /* q1d.sql
 - /* q1e.sql
 - /* q3a_fulljoin.sql
 - /* q3a_innerjoin.sql
 - /* q3a_leftjoin.sql

q3a_innerjoin.sql

```
1 {{ config(materialized='incremental') }}
```

```
2 
```

```
3 SELECT p.*,
       {{ dbt_utils.star(from=source('baseball', 'collegeplaying'),
                          except=['playerid']) }}
```

```
4           | INNER JOIN {{ source('baseball', 'people') }} AS p
5           | ON p.playerid = cp.playerid
6           | WHERE cp.yearid > (SELECT MAX(yearid) FROM {{ this }})
7           | AND p.debut > (SELECT MAX(debut) FROM {{ this }})
8           | {% if is_incremental() %}
9           | WHERE cp.yearid > (SELECT MAX(yearid) FROM {{ this }})
10          | AND p.debut > (SELECT MAX(debut) FROM {{ this }})
11          | {% endif %}
```

```
12 
```

FOLDERS

- ▼ baseball
- ▶ analysis
- ▶ data
- ▶ dbt_modules
- ▶ logs
- ▶ macros
- ▶ models
- ▶ snapshots
- ▶ target
- ▼ tests
 - .gitkeep
 - /* homerun_limit.sql
 - ≡ .gitignore
 - /* dbt_project.yml
 - /* packages.yml

packages.yml

```
1 packages:
```

```
2   - git: "https://github.com/fishtown-analytics/dbt-utils.git"
3     revision: 0.6.4
4 
```

READMED.md

How to Debug Models & Tests

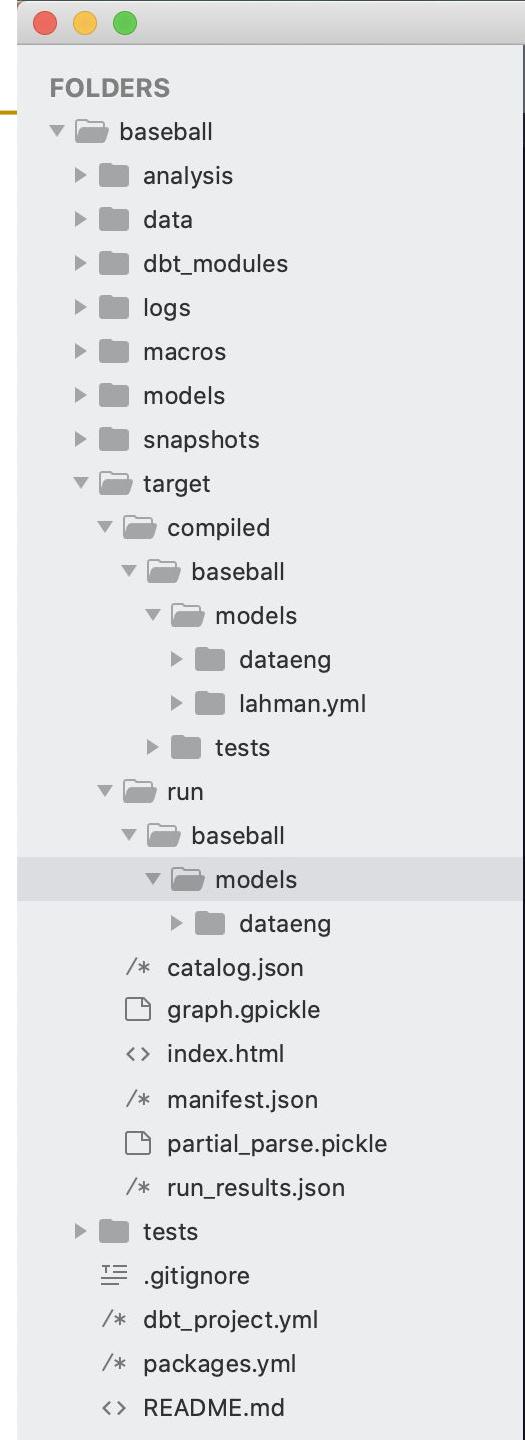
target/compiled

- models and tests after Jinja processing complete

run

- the DDL that runs to build the models

Can clean this all up with `dbt clean`



By default, **each model execution is a transaction**.

- But the DBMS computes views inside whatever query uses them.
- So materialization **via view** (vs table/incremental) affects transaction boundaries.

Note: Can specify pre-/post-“hooks” on models to run more SQL statements before/after

- And can specify each hook as outside or inside the model’s transaction

dbt is a declarative specification framework for:

- **pipelines of ELT-style SQL** transformations
- **data quality tests**.

```
homerun_limit.sql  x
1 SELECT *
2   FROM {{ source('baseball', 'batting') }}
3 WHERE hr > 70
4 LIMIT 10
```

```
q1a.sql  x
1 SELECT namefirst, namelast, p.playerid, yearid
2   FROM people AS p,
3        halloffame AS hof
4 WHERE
5   AND
```

```
inducted_hof_ca_mat.sql  x
1 {{ config(materialized='table') }}
2
3 SELECT DISTINCT namefirst, namelast, p.playerid, s.schoolid, hof.yearid
4   FROM {{ source('baseball', 'people') }} AS p,
5        {{ source('baseball', 'halloffame') }} AS hof,
6        {{ source('baseball', 'collegeplaying') }} AS cp,
7        {{ source('baseball', 'schools') }} AS s
8 WHERE
```

True or False?

1. The `homerun_limit.sql` test passes if it returns 0 rows.

2. The `q1a.sql` model materializes a table called `q1a` in the database.

3. The `q1a.sql` model materializes a table called `inducted_hof_ca_mat.sql` in the database.



dbt is a declarative specification framework for:

- **pipelines of ELT-style SQL** transformations
- **data quality tests**.

```
homerun_limit.sql  x
1 SELECT *
2   FROM {{ source('baseball', 'batting') }}
3 WHERE hr > 70
4 LIMIT 10
```

```
q1a.sql  x
1 SELECT namefirst, namelast, p.playerid, yearid
2   FROM people AS p,
3        halloffame AS hof
4 WHERE
5 AND
```

```
inducted_hof_ca_mat.sql  x
1 {{ config(materialized='table') }}
2
3 SELECT DISTINCT namefirst, namelast, p.playerid, s.schoolid, hof.yearid
4   FROM {{ source('baseball', 'people') }} AS p,
5        {{ source('baseball', 'halloffame') }} AS hof,
6        {{ source('baseball', 'collegeplaying') }} AS cp,
7        {{ source('baseball', 'schools') }} AS s
8 WHERE
```

True or False?

1. The `homerun_limit.sql` test passes if it returns 0 rows.

True! If test fails, return rows for debugging

2. The `q1a.sql` model materializes a table called `q1a` in the database.

False! dbt models are views by default. Views computed on every `dbt run`

3. The `q1a.sql` model materializes a table called `inducted_hof_ca_mat.sql` in the database.

True! use curly braces (Jinja macros) to explicitly materialize tables or reference other tables.