Cloud Architecture for Data Warehouse Transformations

Mihir Parikh

Abstract— **Businesses require data to be transformed in a manner that has domain specific value. This means that transformation logic is often identical for different data consumers. New data practices leverage data warehouses as the optimal place to perform such transformations. However, tooling and development infrastructure for building and maintaining transformation logic is lacking. Dbt is a tool that allows data to be transformed into models which can be consumed. Dbt helps apply mature software development practices that easily transfer to the cloud.**

I. Introduction

DATA from different sources are extracted and loaded into data warehouses to be used by a variety of consumers. More often than not, a number of data transformations apply to many downstream consumers. However, these transformations often occur in Business Intelligence (BI) tools that use their own proprietary markdown languages, expensive third party ETL solutions, or in one-off scripts that live outside the development workflow. Transformation code represents business value and should be held to the same software development standards as application code.

A. Mature Software Development Principles

Data engineering and analytics in many organizations are considered outside the scope of software development. However, it is more prudent than ever that sound software development principles are applied when working in rapidly changing cloud-based systems. Version control, automated testing, CI/CD, DRY, conventions for organizing code, and parallelizing development and production environments are practices that should be considered.

Version control not only makes reverting breaking changes easier, but also, provides a means of documentation. Ideally it offers reasoning for why a change was made and at the very least the identity of the person who made the change. Version control conventions combat the side-effects of ad hoc scripts run from a local machine that are difficult to audit.

Automated testing affords confidence in the code that is written and a form of documentation that communicates the intent of the code itself. If an automated test suite is difficult to write it shows that either the requirements are too ambiguous or that the code is too brittle and should be refactored. It also provides a means for rapid feedback which reduces the bottleneck of getting to production. [12]

Continuous integration/continuous deployment (CI/CD) is an automated means of promoting new code to a higher environment. [1] CI/CD pipelines allow code to get out of the development phase quicker by automating complicated deployment processes. This allows analysts and non-engineers from having to worry about the technical minutia of deployments while at the same time maintaining agility. A CI/CD pipeline is contingent on having certain checks in place such as a robust automated test suite.

DRY referrers to “don’t repeat yourself” which is a fundamental software development practice. [2] Data transformation logic is notoriously scattered and often repetitive. For example, it is easy to have the same code repeated for two different data consumers. If the business rules deviate, the code has to be fixed in two places. Mature software projects are able to compose code to create value without having to repeat existing code. In order to practice DRY there are some implicit code organization requirements that must be satisfied.

To encourage code reusability, code must be organized in a way that files are accessible and located in a predictable place within the same project. In addition, there has to be language consistency. These conventions have to be decided by the team or a tool, such as DBT, can be used to provide documented structure.

Finally, mature software development practices establish consistency between the pre-production and production environments. Consistency can be achieved by virtualization. [3] Developing code within a Docker container that runs in every environment provides confidence that it will work in production.

B. Data Warehouse

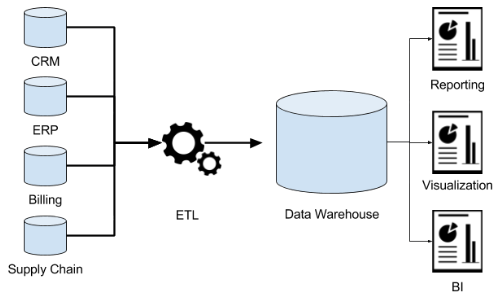


Fig. 1 Data Warehouse [4]

A data warehouse is constructed by integrating data from multiple heterogeneous sources that support analytical reporting, structured and/or ad hoc queries, and decision making. Data warehousing involves data cleaning, data integration, and data consolidations. [4] Figure 1 shows an example data warehouse with a variety of sources and consumers.

C. DBT

Dbt is a tool that performs the 'T' in ELT. It does not extract or load data but is effective at transforming data within the warehouse. Transform after load architecture is known as ELT (extract, load, transform). ELT has become commonplace because of the power of modern analytic databases. Data warehouses like Redshift, Snowflake, and BigQuery are extremely performant and very scalable such that at this point most data transformation use cases can be much more effectively handled in-database rather than in some external processing layer. Dbt is a tool for writing and executing data transformation jobs that run inside the warehouse. Dbt's only function is to take code, compile it to SQL, and then run it against the warehouse. Figure 2 diagrams where Dbt would exist in a data pipeline.

At its most basic level, Dbt has two components: a compiler and a runner. Dbt code is written in a text editor and then invoked from the command line. Dbt compiles all code into raw SQL and executes that code against the configured data warehouse. In addition, Dbt provides a testing framework for data and schema tests. [5]

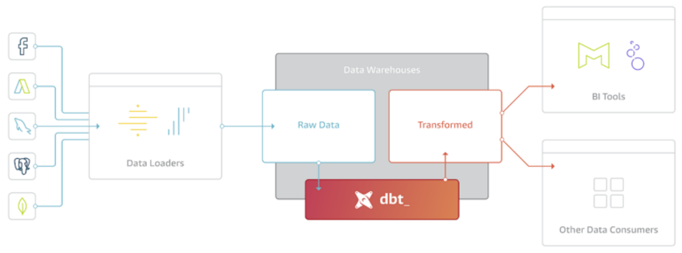


Fig. 2 Dbt's Role in a data pipeline [5]

D. Docker

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow developers to package up an application and its dependencies. By doing so, applications can run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.

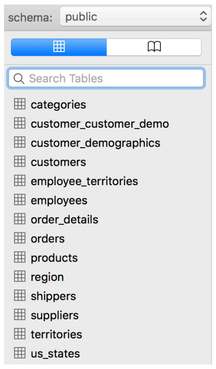
In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, it uses the same Linux kernel as the system that it is running on and only requires applications be shipped with dependencies not present on the host. This gives a significant performance boost and reduces the size of the application itself.

E. Outline

The remainder of the paper goes through the process of using Dbt to perform transformations to sample Ecommerce data and the path for deploying a Dbt project to the cloud using Amazon Web Services (AWS).

II. Building Dbt Models

The process of developing and deploying a Dbt project is best explained through an example. I created a Postgres RDS instance in AWS and seeded it with sample Ecommerce data. The Postgres database simulates a data warehouse and the seeding script simulates the extract and load tasks by taking data from a specified source and loading it into the warehouse. Figure 3 shows all the database tables loaded into the warehouse. My goal is to create a Customer Invoices table which joins pieces of Employee, Region, Territories, and Employee Territories tables together.


schema: public

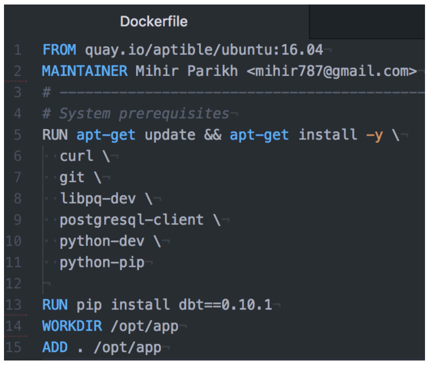
Search Tables

- categories
- customer_customer_demo
- customer_demographics
- customers
- employee_territories
- employees
- order_details
- orders
- products
- region
- shippers
- suppliers
- territories
- us_states

Fig. 3 Data warehouse tables after loading

A. *Installing Dbt in a Docker Container*

There are a number of ways to install Dbt on your local machine. However, your cloud setup likely does not run OSX or the particular version of Windows you may be using. Thus, it is important to establish consistency from the beginning. Docker allows you to pick a base image to build off which you can use to run both locally and in the cloud. I chose a fairly standard Ubuntu:16.04 image and used python-pip commands to install Dbt. Figure 4 shows the Dockerfile.


Dockerfile

```
FROM quay.io/aptible/ubuntu:16.04
MAINTAINER Mihir Parikh <mihir787@gmail.com>
# ------------------------------------------------
# System prerequisites
RUN apt-get update && apt-get install -y \
  curl \
  git \
  libpq-dev \
  postgresql-client \
  python-dev \
  python-pip

RUN pip install dbt==0.10.1
WORKDIR /opt/app
ADD . /opt/app
```

Fig. 4 Dockerfile for installing Dbt and its necessary dependencies

B. *Dbt Project Setup*

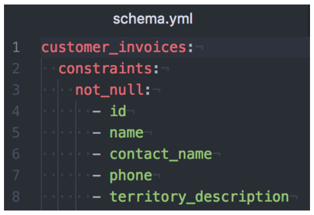
Dbt provides an initialize command that creates the basic directory structure of the project. The two main files that need to be created are the *'profiles.yml'* and *'project.yml'*. The *'profiles.yml'* defines the database connection of the RDS data warehouse and the schema that will be used to store the transformed data. The *'project.yml'* defines the directory structure of the models. Thus, it defines which schema models in the directory belong to. [5] This is useful for implementing staged models that do not get exposed to data consumers.

C. *Developing the Customer Invoice Model*

Creating a new model is done by creating a new SQL file *'customer_invoices.sql*'. Each new model must define how to materialize the data. In this case I wanted to create a table, but views are also a valid option. The rest of the code is raw SQL. I can use all SQL commands supported by the data warehouse. However, Dbt does provide some niceties for referencing other models in the project via the *'{{ ref() }}'* command. This also sets up a dependency tree which specifies the order in which models should be built.

D. *Automated Test of Customer Invoices*

Dbt provides a built-in testing framework that allows assertions to be made on built models. Schema tests are primarily used to make simple assertions such as not null, accepted values, uniqueness, and relationships to other models. Figure 5 shows some of the constraints specified for Customer Invoices.

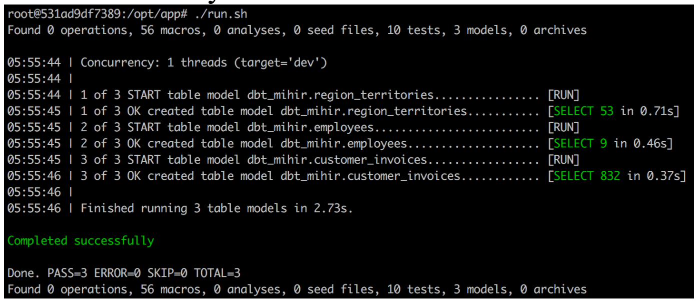

schema.yml

```
customer_invoices:
  constraints:
    not_null:
      - id
      - name
      - contact_name
      - phone
      - territory_description
```

Fig. 5 Tests for Customer Invoice model

E. *Compile, Build, Test*

The Dbt run command allows the model code to be compiled and executed. This creates models in the defined schema. Having built the models, I can run the test suite to ensure that the transformed data meets expectations. Figure 6 shows the output of the build command. Figure 7 shows the output of the test command. Take note that both commands are executed directly in the container.

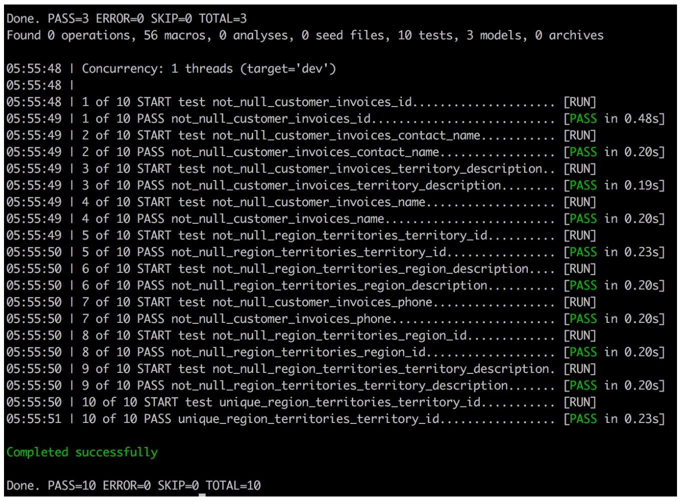

```
root@531ad9df7389:/opt/app# ./run.sh
Found 0 operations, 56 macros, 0 analyses, 0 seed files, 10 tests, 3 models, 0 archives

05:55:44 | Concurrency: 1 threads (target='dev')
05:55:44 |
05:55:44 | 1 of 3 START table model dbt_mihir.region_territories............... [RUN]
05:55:45 | 1 of 3 OK created table model dbt_mihir.region_territories.......... [SELECT 53 in 0.71s]
05:55:45 | 2 of 3 START table model dbt_mihir.employees........................ [RUN]
05:55:45 | 2 of 3 OK created table model dbt_mihir.employees................... [SELECT 9 in 0.46s]
05:55:45 | 3 of 3 START table model dbt_mihir.customer_invoices................ [RUN]
05:55:46 | 3 of 3 OK created table model dbt_mihir.customer_invoices........... [SELECT 832 in 0.37s]
05:55:46 |
05:55:46 | Finished running 3 table models in 2.73s.

Completed successfully

Done. PASS=3 ERROR=0 SKIP=0 TOTAL=3
Found 0 operations, 56 macros, 0 analyses, 0 seed files, 10 tests, 3 models, 0 archives
```

Fig. 6 Output from executing Dbt run



```
Done. PASS=3 ERROR=0 SKIP=0 TOTAL=3
Found 0 operations, 56 macros, 0 analyses, 0 seed files, 10 tests, 3 models, 0 archives

05:55:48 | Concurrency: 1 threads (target='dev')
05:55:48 |
05:55:48 | 1 of 10 START test not_null_customer_invoices_id.................... [RUN]
05:55:49 | 1 of 10 PASS not_null_customer_invoices_id.......................... [PASS in 0.48s]
05:55:49 | 2 of 10 START test not_null_customer_invoices_contact_name.......... [RUN]
05:55:49 | 2 of 10 PASS not_null_customer_invoices_contact_name................ [PASS in 0.20s]
05:55:49 | 3 of 10 START test not_null_customer_invoices_territory_description.. [RUN]
05:55:49 | 3 of 10 PASS not_null_customer_invoices_territory_description........ [PASS in 0.19s]
05:55:49 | 4 of 10 START test not_null_customer_invoices_name.................. [RUN]
05:55:49 | 4 of 10 PASS not_null_customer_invoices_name........................ [PASS in 0.20s]
05:55:49 | 5 of 10 START test not_null_region_territories_territory_id......... [RUN]
05:55:50 | 5 of 10 PASS not_null_region_territories_territory_id............... [PASS in 0.23s]
05:55:50 | 6 of 10 START test not_null_region_territories_region_description.... [RUN]
05:55:50 | 6 of 10 PASS not_null_region_territories_region_description......... [PASS in 0.20s]
05:55:50 | 7 of 10 START test not_null_customer_invoices_phone................. [RUN]
05:55:50 | 7 of 10 PASS not_null_customer_invoices_phone....................... [PASS in 0.20s]
05:55:50 | 8 of 10 START test not_null_region_territories_region_id............ [RUN]
05:55:50 | 8 of 10 PASS not_null_region_territories_region_id.................. [PASS in 0.20s]
05:55:50 | 9 of 10 START test not_null_region_territories_territory_description. [RUN]
05:55:50 | 9 of 10 PASS not_null_region_territories_territory_description...... [PASS in 0.20s]
05:55:50 | 10 of 10 START test unique_region_territories_territory_id.......... [RUN]
05:55:51 | 10 of 10 PASS unique_region_territories_territory_id................ [PASS in 0.23s]

Completed successfully

Done. PASS=10 ERROR=0 SKIP=0 TOTAL=10
```

Fig. 7 Output from running Dbt test

III. AUTOMATING DBT EXECUTION IN THE CLOUD

In order to have newly loaded data included in transformed models, Dbt needs to be run on a specified interval. Deploying to the cloud allows for cron automation on scalable resources.

Since a Docker container was already built, it greatly simplifies the cloud architecture and overall configuration. Most cloud providers have managed solutions for running containers efficiently.

A. *AWS Deployment Strategy*

The high-level deployment strategy involves publishing the Docker image to a private registry and deploying that container on managed resources. To execute the cron script, a serverless task can be created to '*ssh'* into the deployed resource to run the script.

AWS provides Elastic Container Registry (ECR) which provides full management of private Docker images. [8] It works just like the Docker's own registry and can thus be pushed to and pulled from using the AWS CLI. Having pushed my custom Docker image, I was able to use ECR Fargate to deploy my application.

ECR Fargate allows you to start up an application from a custom image without having to provision, configure, and scale clusters of virtual machines. ECR takes about ten minutes to spin up basic resources after which a service task is kicked off to scale and check the health of the cluster. If a task does not positively respond, another task is kicked off to spin up more resources. Behind the scenes, Fargate creates an EC2 instance, however it is not displayed in the EC2 console. Since Fargate is managing the resources, it does not provide options to more finely configure instances. Fargate does present more configurable options where you can specify your EC2 instances but requires more overhead when it comes to managing the cluster. Figure 8 summarizes the functionality of Fargate. In addition to providing basic resources, it takes advantage of Cloudwatch to manage logs and in our case a record of whether models were run and tested successfully. [9]

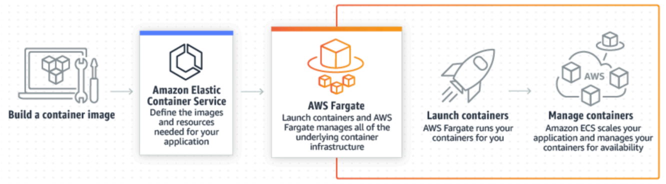

Build a container image
Amazon Elastic Container Service
Define the images and resources needed for your application
AWS Fargate
Launch containers and AWS Fargate manages all of the underlying container infrastructure
Launch containers
AWS Fargate runs your containers for you
AWS
Manage containers
Amazon ECS scales your application and manages your containers for availability

Fig. 8 Functionality of Amazon ECS Fargate [9]

B. *CI/CD*

Architecting a CI/CD pipeline with a containerized solution is fairly straight forward. The addition of a staging ECR registry and a staging ECS Fargate cluster would allow for an intermediate state between development and production. When new code gets merged into the master branch, an AWS CodePipeline webhook would get triggered to kick-off the pipeline.

The first step would be to push the updated container to the staging ECR repository. Which would in turn trigger an AWS CodeDeploy to the staging cluster. If running and testing is successful, then CodeDeploy would trigger a push of the staging ECR image to the production image. This would trigger a deploy to the production ECS cluster. It is important to note that the same production warehouse is used throughout the pipeline. However, different schemas are configured for each step and can be deleted at various stages.

IV. RESULTS AND ANALYSIS

The sample deployed Dbt project provides some insight as to whether Dbt is a viable tool for performing data transformations in a mature software development cycle.

Easily getting a Dbt project running in both development and production (cloud) environments alleviates concerns in regard to managing external dependencies and versions. The development infrastructure Dbt provides is probably the most valuable asset of the tool. Having a clear convention for where logic belongs makes it easier for teams to write more organized and maintainable code. Since logic is written in SQL itself, it also breaks the silo of developers being the sole owners of data transformation. Analysts and non-engineers can work comfortably within a language they already know. Finally, Dbt's testing framework ensures that transformation logic and data loaded into the warehouse meet expectations. This can help fix issues at points where data is collected and improve overall data integrity.

Where Dbt is lacking is that it is not a catch all tool for ETL. The focus is solely on transformation and thus still requires extraction and loading logic to be specified using custom scripts or other third-party services. It also requires greater RDS resources for the warehouse itself since transformed data lives within different schemas. Thus, a warehouse may need to be large enough to accommodate developers to have their own development schemas.

V. Future Considerations and Conclusions

Future considerations of implementing Dbt infrastructure in the cloud revolve around serverless solutions. Amazon Lambda functions provide a way to run code as needed by paying only for compute time. Running Docker in a Lambda function could provide the means to reduce the cost of ECS clusters. [11] However, challenges of implementing this include limited file size of Lambda layers, integrating with pipelines, and wrapping Dbt commands in languages that work with Lambda runtimes. There are opensource projects in the works that expose Lambda runtime Docker images, however I was not able to make any of them work with Dbt.

In conclusion, Dbt provides the tooling necessary to centralize transformation logic in an organized code repository. This logic represents business value and should utilize strong software development practices. Since Dbt is opensource, it moves business value from third-party services to internal code repositories which decouple analytics value from BI service providers.

*All code can be found at: https://github.com/mihir787/mihir-dbt/

References

[1] Fowler, Martin, and Matthew Foemmel. "Continuous integration." *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122 (2006): 14.

[2] Fowler, Martin. "Is design dead?." *SOFTWARE DEVELOPMENT-SAN FRANCISCO-* 9.4 (2001): 42-47.

[3] Poppendieck, Mary, and Michael A. Cusumano. "Lean software development: A tutorial." *IEEE software* 29.5 (2012): 26-32.

[4] Devlin, Barry, and Lynne Doran Cote. *Data warehouse: from architecture to implementation*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[5] "About." *Transform Data in Your Warehouse*, www.getdbt.com/about/.

[6] Bernstein, David. "Containers and cloud: From lxc to docker to kubernetes." *IEEE Cloud Computing* 1.3 (2014): 81-84.

[7] Suarez, Anthony Joseph, et al. "Software container registry container image deployment." U.S. Patent Application No. 14/975,627.

[8] "Amazon ECR | Amazon Web Services." *Amazon*, Amazon, aws.amazon.com/ecr/.

[9] "AWS Fargate - Run Containers without Having to Manage Servers or Clusters." *Amazon*, Amazon, aws.amazon.com/fargate/.

[10] "AWS CodePipeline | Continuous Integration & Continuous Delivery." *Amazon*, Amazon, aws.amazon.com/codepipeline/.

[11] Villamizar, Mario, et al. "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures." *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016.

[12] Berner, Stefan, Roland Weber, and Rudolf K. Keller. "Observations and lessons learned from automated testing." *Proceedings of the 27th international conference on Software engineering*. ACM, 2005.