OPEN DATA SCIENCE CONFERENCE

Boston | April 30 - May 4, 2019

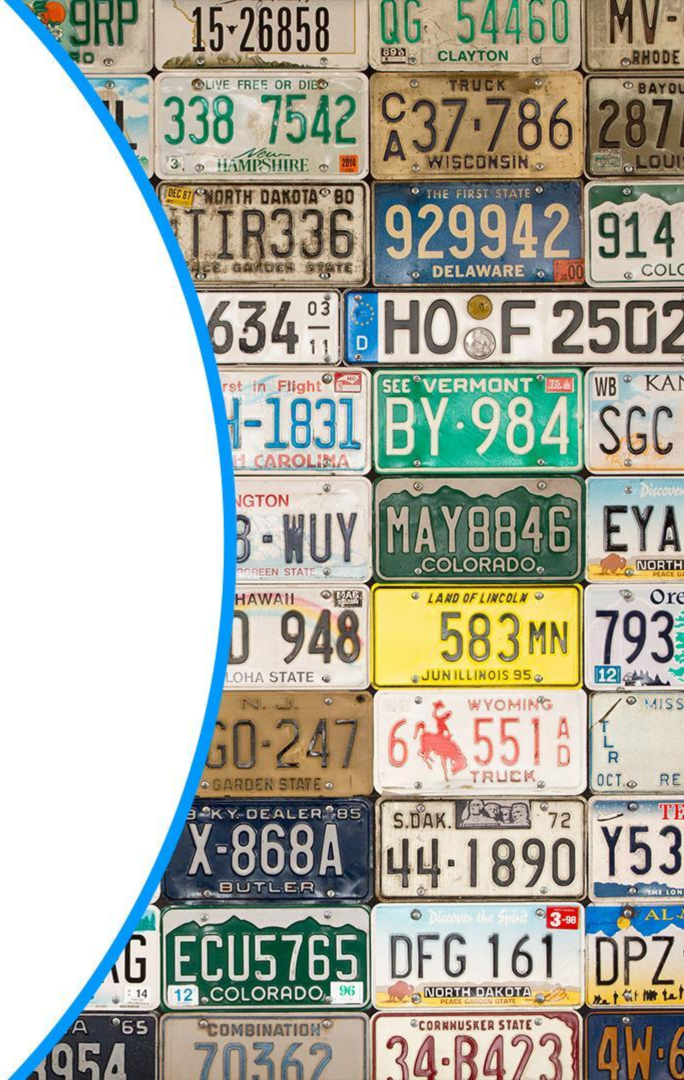
#ODSC

@ODSC

CarGurus®

ETL without pager duty

Greg Joondeph-Breidbart
David Nash



Greg Joondeph-Breidbart

- VP, Analytics & Process Innovation @ CarGurus
- Background in Economics and Computer Science
- Experience building and scaling data engineering teams

David Nash

- Manager, Data Engineering
- Background in Physics and academia

...and these are the people that actually did the work

Jake Thomas, Lead Data Engineer

Seth Woodworth, Senior Data Engineer

Jessie Bleiler, Data Engineer

Dan Rubin, Data Engineer

Who do we support?

At CarGurus, our Analytics Engineering team supports most of the business.

Finance
Publicly facing KPI Reporting

Product
A/B test progress and results

Business Development
Health of partner integrations

Marketing
Campaign performance

Core Engineering
Health of site deployments

Any failure to deliver timely accurate data to the above folks will erode trust.

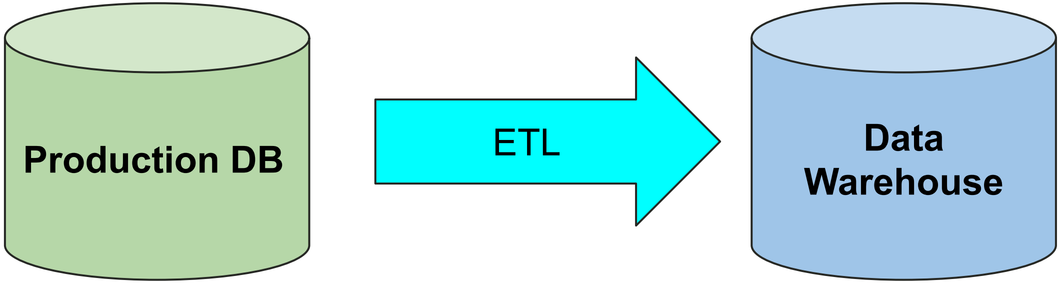
CarGurus

What you will learn today

1. What ETL & ELT are, and what the key differences are between the two
2. The history of ETL and why it is important
3. How the CarGurus data warehouse is structured
4. How we use Airflow to orchestrate and schedule ELT jobs
5. Lessons we've learned in making ELT maintenance easy

ETL! Extract, Transform, Load

A fancy term for moving data from one place to another, typically an application to a data warehouse.



The goal is to get all of your business data into one place where it can be queried

History of ETL

A brief history of ETL:

- Relational DB based data warehouses couldn’t handle all of the data from production systems
- The answer was to perform transformation on extraction, creating “fact” and “dimension” tables. This helped to reduce data through filtering and aggregation.

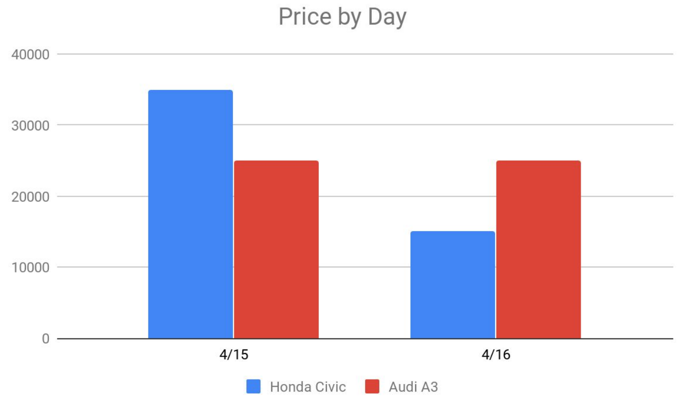
vehicle_sales				
ID	Purchase_ Time	Vehicle	Color	Price
1	4/15 12:05	Honda Civic	Red	$15K
2	4/15 13:01	Honda Civic	Blue	$20K
3	4/16 08:30	Audi A3	Gray	$25K
4	4/16 10:11	Honda Civic	Red	$15K

becomes

vehicle_sales_rollup			
Day	Vehicle	Total_Count	Total_Price
4/15	Honda Civic	2	$35K
4/15	Audi A3	1	$25K
4/16	Audi A3	1	$25K
4/16	Honda Civic	1	$15K

CarGurus

Example: Answering business questions with ETL



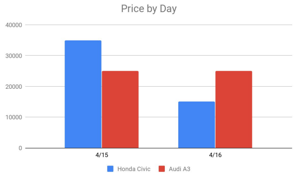
vehicle_sales_rollup			
Day	Vehicle	Total_Count	Total_Price
4/15	Honda Civic	2	$35K
4/15	Audi A3	1	$25K
4/16	Audi A3	1	$25K
4/16	Honda Civic	1	$15K

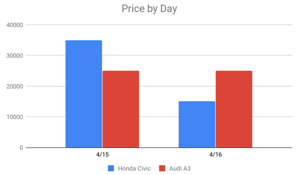
What happens when you want to report on sales by vehicle color?

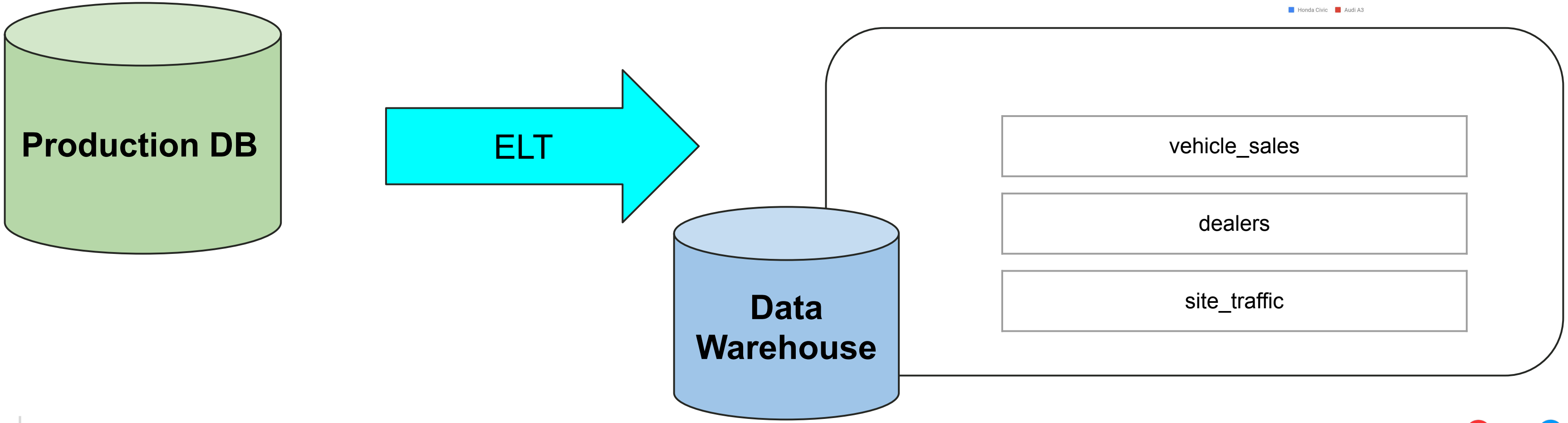
You need to either create a new data transformation that includes VEHICLE_COLOR, or add the VEHICLE_COLOR column. Both options are difficult to implement, and prone to error.

Enter ELT

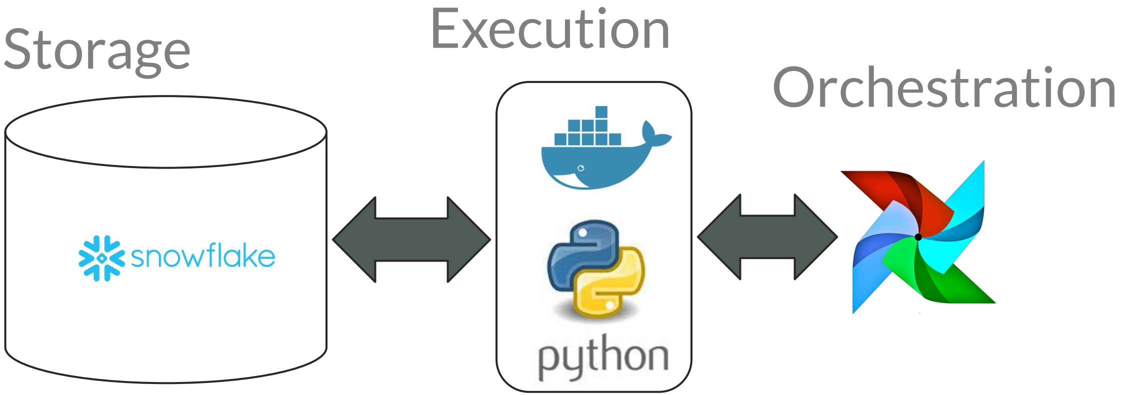
- With column store DBs, parallel architecture, and cheaper than ever storage costs, we can get around a number of these issues by moving raw data directly into our data warehouse.
- Then, we transform the data later







Broad Architecture and Technology Overview

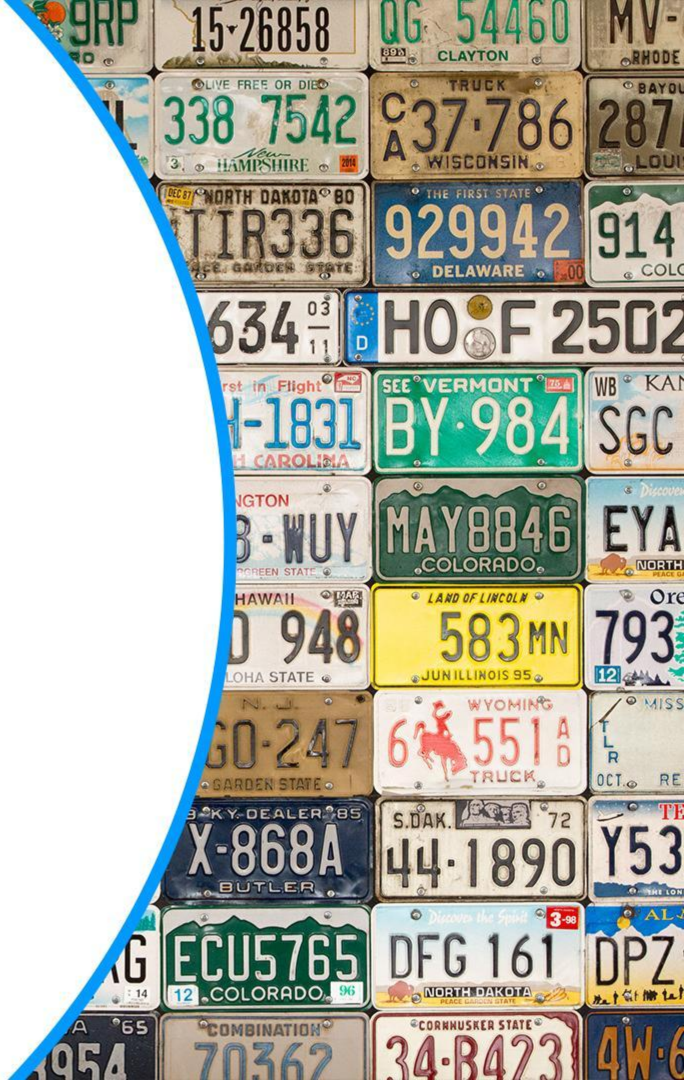


Let's discuss how we built our data warehouse, and what issues we've encountered that we learned from in creating our homegrown ELT system. We'll discuss the following two themes:

- Job orchestration
- Making debugging easy

CarGurus

Job orchestration

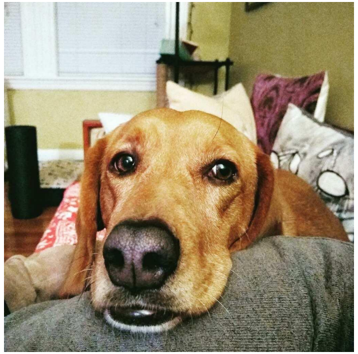


We use Airflow to orchestrate and schedule our jobs

Airflow is a popular tool used by data engineers to schedule and manage their work

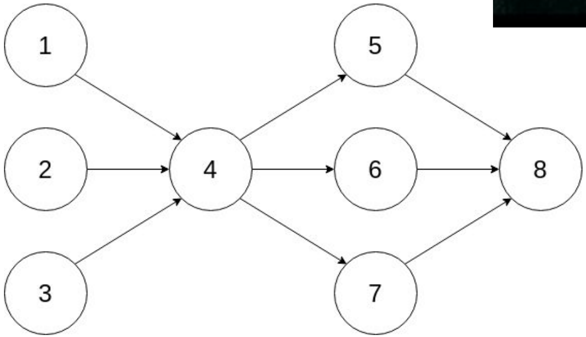
Not an ETL tool, but a DAG management framework

What's a DAG?



← not this

“A finite directed graph with no directed cycles.” --Brad Pitt



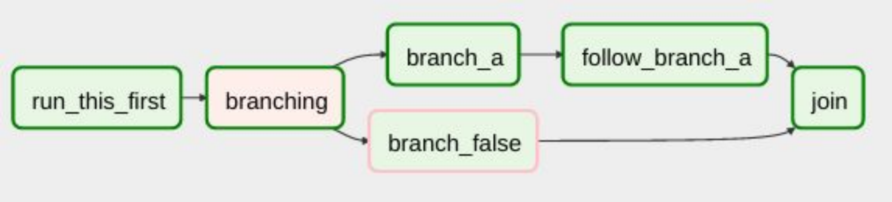
+

=

CarGurus

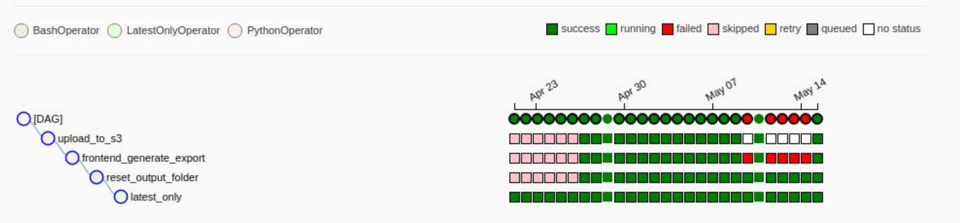
Airflow: in practice

An airflow dag looks like this:



Airflow jobs get inputs from their schedule, like this...

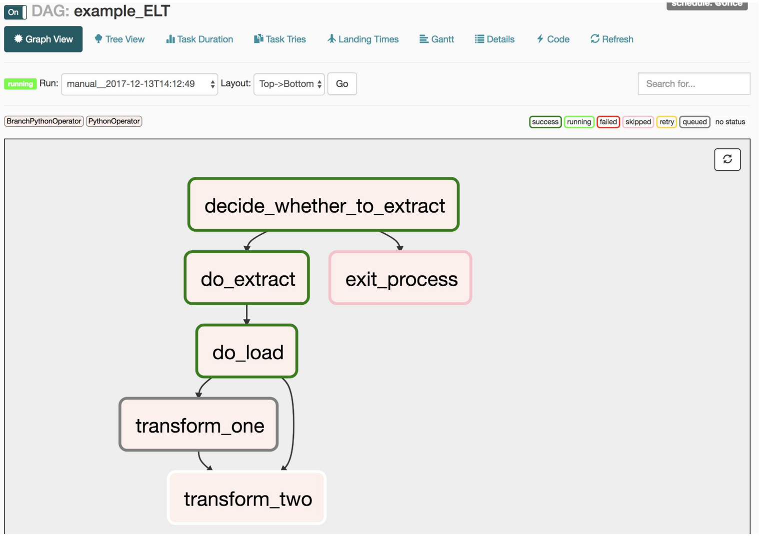
```
return {
    'END_DATE': ds,
    'conf': configuration,
    'dag': task.dag,
    'dag_run': dag_run,
    'ds': ds,
    'ds_nodash': ds_nodash,
    'end_date': ds,
    'execution_date': self.execution_date,
```



...and you can examine past runs like this, called the "tree view"

Our first mistake made deploying new code difficult

At the beginning, airflow was running inside docker and airflow inputs were part of control flow



...but it was a deployment nightmare

- Every time we changed a schedule or a dependency we had to rebuild containers
- Heavy airflow dependencies inside containers
- Worst of all: **Every time we redeployed we had to stop all jobs - that's not how software should work!**

CarGurus

The answer: treat our ELT system as a running application

Developer pushes change to loader process

Jenkins picks up change, runs tests, and pushes new image to box

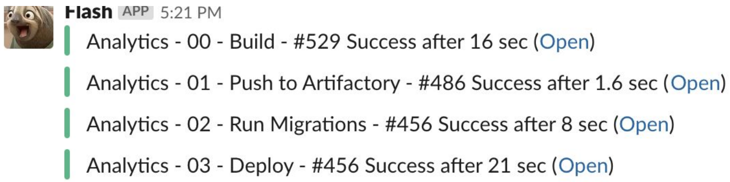
Running containers are not interrupted, the next time a job runs it picks up the new image

Developer pushes change to DAG dependencies or schedule

Airflow services are restarted

Running containers are not interrupted by scheduler reboot, following runs are processed according to new schedule or dependencies

...and these steps send notifications to us

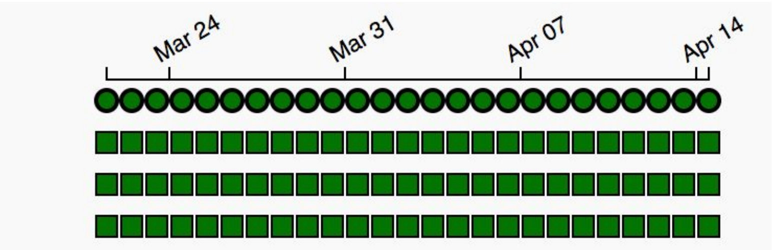


Deployment is fixed, but our job definitions are brittle

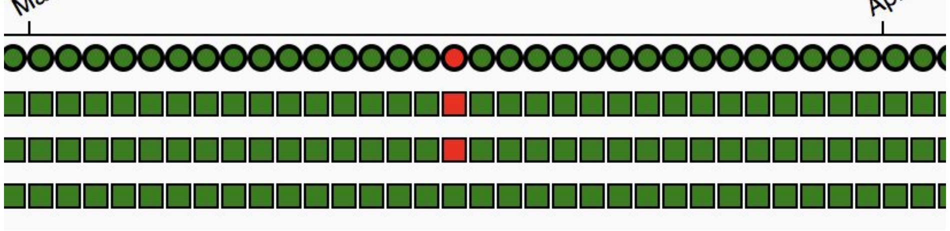
```
return {
    'END_DATE': ds,
    'conf': configuration,
    'dag': task.dag,
    'dag_run': dag_run,
    'ds': ds,
    'ds_nodash': ds_nodash,
    'end_date': ds,
    'execution_date': self.execution_date,
```

We were grabbing time ranges from the jobs themselves

Bootstrapping a new data source:



When things get spotty:
Hold up a backfill, or continue and hunt the bad run down?



CarGurus

The answer: Trust the database

Rather than letting airflow decide what to do, each job now scans for changed records, iterating until the data warehouse is up to date with the source

Warehouse at time as of time t2

Name	Status	LastUpdated
Bob	Paying	t1
Sue	Paying	t2
Joe	Not Paying	t2

Select * from source where LastUpdated > t2

Name	Status	LastUpdated
Bob	Paying	t1
Sue	Paying	t2
Joe	Paying	t3
Mary	Paying	t3

Carry over the change to Joe's status, and the new customer Mary

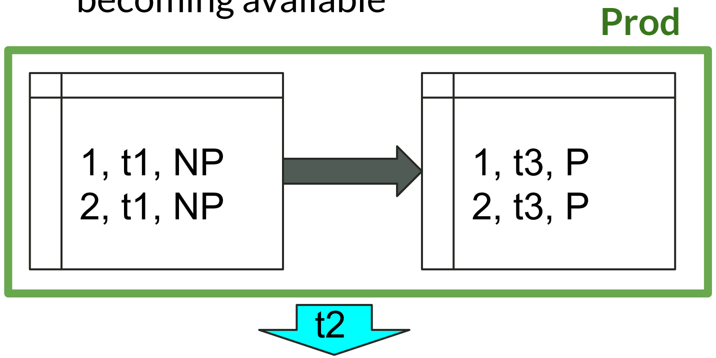
Batch this in ways the source and database can handle, and loop until the timestamp is up to date

Debugging and maintenance, made easy

Warehousing changes in flight: transactionless sources

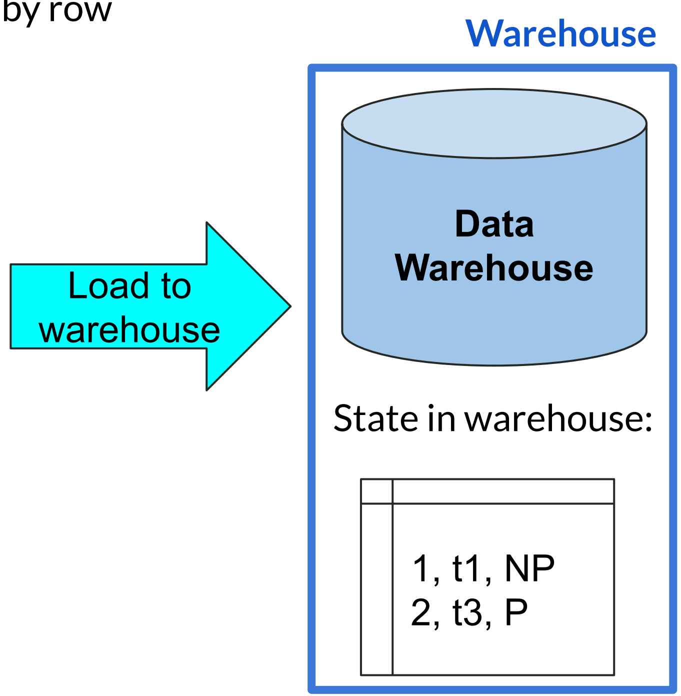
What about sources with no transactions? Or with no read committed isolation?

Set status='Paying' during the read, row by row becoming available



Read at time t2, during a process update on rows 1 and then 2

The read sees the version of record 1 at time t1, but by the time it reads record 2 it is already time t3, and it does a "dirty read" of its state

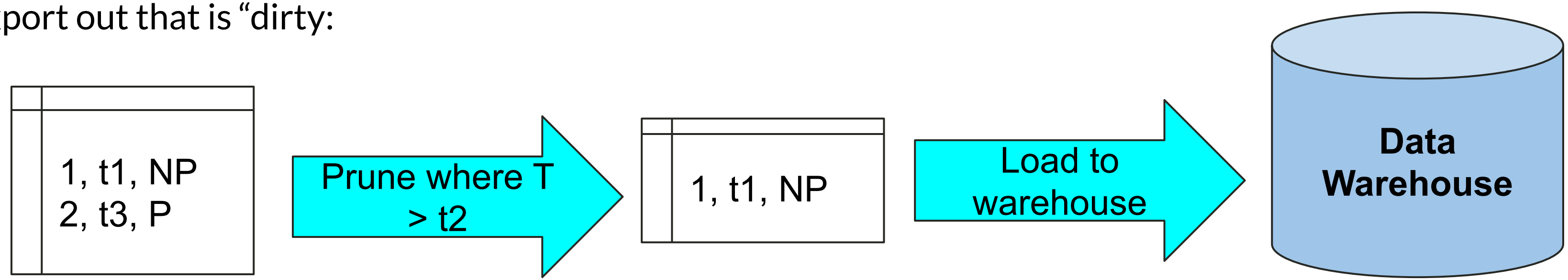


At the end of this process, we know that record 2 is now 'Paying' but record 1 is still recorded as not paying!

If we high watermark on the time, we'll never get record 1 right unless it changes again!

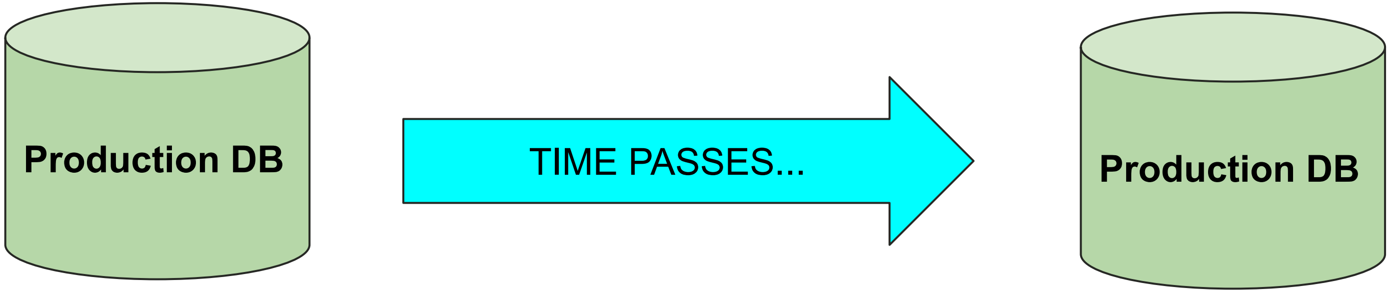
Don’t load dirt!

Instead, we can record the job time of t2, and just throw anything from our export out that is “dirty:



Now, next time our job spins up, at time t4 perhaps, we don’t have to worry about missing records, we know that customer 2 will be picked up

My product engineers keep deploying features!




```
cargurus> desc consumer_leads
+------------------------------+--------------------+------
| Field                        | Type               | Null
+------------------------------+--------------------+------
| id                           | int(10) unsigned   | NO
| type_id                      | char(2)            | NO
| submission_date              | timestamp          | NO
```

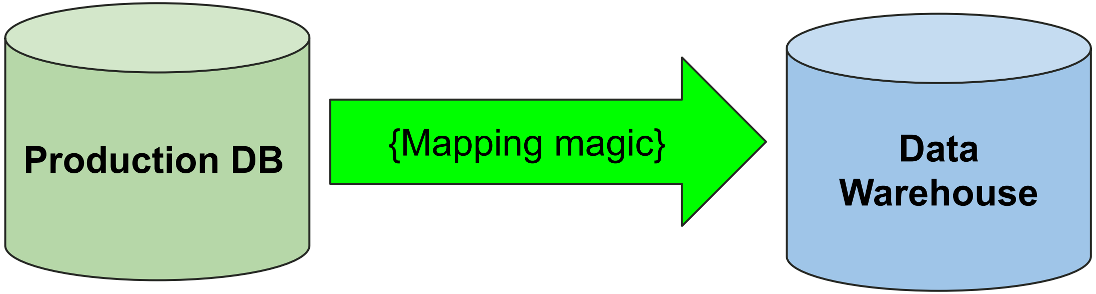
```
cargurus> desc consumer_leads
+--------------------------------+--------------------+------
| Field                          | Type               | Null
+--------------------------------+--------------------+------
| id                             | int(10) unsigned   | NO
| type_id                        | char(2)            | NO
| submission_date                | timestamp          | NO
| success                        | tinyint(1)         | NO
```

...and what's worse, my analysts want that data!

Read schema...write schema

Use the database, leverage information_schema

```
cargurus> SELECT
       ->     COLUMN_NAME,
       ->     DATA_TYPE,
       ->     NUMERIC_PRECISION,
       ->     NUMERIC_SCALE,
       ->     DATETIME_PRECISION,
       ->     COLUMN_COMMENT
       -> FROM information_schema.columns
       -> WHERE TABLE_NAME = 'consumer_leads'
       -> ORDER BY ORDINAL_POSITION ASC;
+--------------------------------+-----------+
| COLUMN_NAME                    | DATA_TYPE |
+--------------------------------+-----------+
| id                             | int       |
| type_id                        | char      |
| submission_date                | timestamp |
| success                        | tinyint   |
```



Parity has been restored on WAREHOUSE.AWS_COST.BILLING.
Columns added:
"PRODUCT_STEPS"
Love without rules.

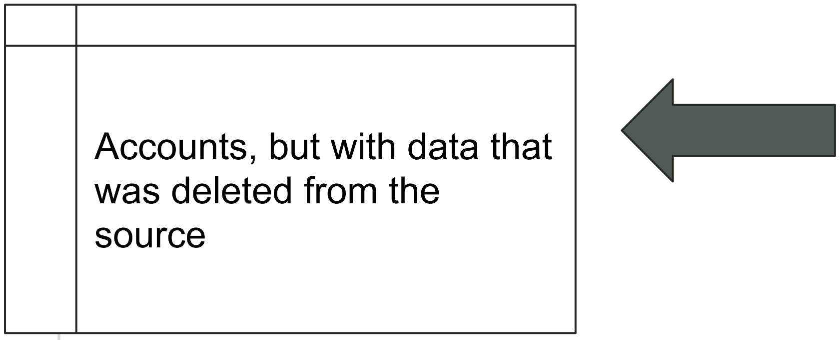
Alerting here builds trust, and exposes possible issues as well

Productionizing ops - reloading from the source

We learned that reloading a existing table from scratch was not a terribly rare operation

Various problems could come up that make a table usable by one stakeholder group but not by another

E.g. Deletes at a source causing one user to experience data duplication, but another user needs aggregated reports to continue to run on to of it

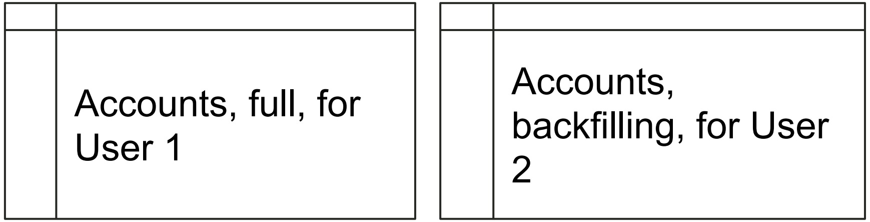


User 1 is happy with an aggregate immune to the dupes

User 2 needs a dupe-less table

...enter, the sideload:

```
+    "ACCOUNTS_SIDELOAD_DEL": {
+        SOURCE_OBJECT: "Account",
+        DESTINATION_TABLE: "ACCOUNTS_SIDELOAD_DEL",
+        BATCH_SIZE: 10000,
+    },
```



User 1 queries main table

User 2 waits for second table to finish

Same **source**, configurable **targets** for simultaneous loading to multiple places

Building loaders as well as transformers that have this abstraction saves time, and both data engineers and stakeholders trust that maintenance will be done with production quality

What do you do when something goes wrong?

Let's look at how we debugged a missing day of data

The characters: Our team, and a product engineering team

The set: A custom system where the second engineering team wanted to be able to trigger our jobs with their own exports

The scene: No data for a day where they thought they exported

1. Source, Derived, and Process fields

We load our tomcat logs, parse them out, parse URLs…

But always always save the raw line

```
| MKT_CAMPAIGN          | [Derived] The request's utm_campaign query parameter, if existent.
| RAW_LINE              | [Source] The raw, untouched log line as represented at the source.
| SOURCE_PATH           | [Process] The full path of the source log file in S3.
```

Also, we keep track of the path to wherever we grab files from, so we can chase down problems at the source

CarGurus

2. Timestamp everything

Clearly, source timestamps are critical…

But if we keep track of changes in the source, why not at the destination?

COLUMN_NAME	COMMENT
CREATEDDATE	Created Date
LASTMODIFIEDDATE	Last Modified Date
_EXTRACTED_AT_	[Process] The timestamp associated with pulling a particular record from an upstream source
_LOADED_AT_	[Process] The load timestamp of a particular record
_UPDATED_AT_	[Process] The most recent time a record was updated

For example, while loading Salesforce data, we know, **for every record**, when it was pulled from its source, when the record was first created, and when it was updated, **also in our data warehouse.**

3. Tie through to full job execution-level metadata

Sometimes the table looks like it's up to date based on those timestamps, but still looks weird

Maybe a user expected to see changes where there weren't any, but _LOADED_AT_ looks fine… enter the META tables:

Here we have **one record per job**, and we know what was executed in aggregate, but also in the internals of the job

COLUMN_NAME
CREATED_AT
SOURCE_HOST
SOURCE_TABLE
DESTINATION_SCHEMA
DESTINATION_TABLE
STAGE_PATH
EXTRACT_DURATION
LOAD_TO_STAGE_DURATION
LOAD_FROM_STAGE_DURATION
RECORDS_WRITTEN
RECORDS_INSERTED
RECORDS_UPDATED

4. Connecting the dots between artifacts

Job times, query results, code base, and job outputs all connect - and they all do so with the minimum of tooling

Do what you can to tie through to queries, too:

COLUMN_NAME	COMMENT
CREATED_AT	NULL
QUERY_TAG	The associated query tag of the instrumented load
DESTINATION_TABLE	The destination table
DESTINATION_SCHEMA	The destination schema

That way it's easy to reconstruct what your **code** did in the **warehouse**

The investigation

1. Looking at the **table** we saw we ran a job recently, what gives??
2. In the **meta** schema, we noticed that we had no records from the corresponding job
3. Back to the **table**, we checked the **source** file from their export
4. Oops no records there either!

After action report: what did we learn?

Trust: During the exercise, we showed that we:

1. Keep track of everything we do
2. We never never never do anything to the data itself
3. Our process is reliable

Self service: Not only was trust built, but self service was born - next time this user group could debug their own situation, and escalate problems to the right location

What did we learn?

- Extracting data as-is helps with not only analytics insights but with debugging and maintenance
- The ELT service may incorporate a job scheduler, but ELT code itself should not **incorporate** that tool as a dependency
- Automate database maintenance but track and alert on what your software does
- When moving data, change nothing and track everything
- Put debugging information in easy-to-reach systems and empower stakeholders to do their own digging

More reading

New CarGurus Engineering blog, content upcoming: Revved

Team member (Jake Thomas) writes his own blog too