holistics

The Analytics Setup Guidebook

How to build scalable analytics & BI stacks in the modern cloud era

holistics

The Analytics Setup Guidebook

We are Holistics. We've been making data analytics tools for over four years, and helped more than a hundred companies build their business intelligence capabilities, sometimes from scratch.

A huge chunk of our time is spent educating and assisting companies as they migrate to this new world of cloud-based business intelligence tools. For the first time ever, we're putting that experience up for the world to read.

www.holistics.io

Besides the guidebook, we also regularly share our thoughts on data analytics, business intelligence and how we build Holistics.

www.holistics.io/blog

Written by: Huy Nguyen, Ha Pham, Cedric Chin

Last edited: July 2nd, 2020

Designed and Published by: My Nguyen, Dung Pham, Son Hoang, Tuan Nguyen, Khai To, Anthony T. Do

Praise for the book

I love that you went into more detail in the later chapters around modeling, transformations, and providing real-world case studies like the Snowplow case which I'd love to hear more about!

– Ian Young, Analytics Engineering Consultant

[...] the book is great and is exactly what someone in my position needs, especially the part about the EL tools, the data modeling layer, and the relationship between the CEO and the Data Analyst.

– Thomas Kane, Head of BI & Data, Unissu

I thought ELT was just another cool kids' jargon [...] Chapter 2 slapped me hard in the face telling me that I was concluding too soon and I know nothing about it.

– Ence Prastama, Data Engineer, Vidio

I love the bits of personality woven in [...] It's definitely getting me excited for my analytics degree and really looking to be versed on the technicalities.

– Niki Torres, Director of Growth, Airalo

Foreword

Have you ever wondered how to build a *contemporary* analytics stack that is useful, scalable, and easy to maintain? And have you looked into building such a stack, only to find yourself quickly drowning in a sea of jargon?

We know how that feels, because we've been there. The truth is that much knowledge of modern data analytics is locked up in the heads of busy practitioners. Very little of it is laid out in a self-contained format.

In this short book, we will give you a practical, high-level understanding of a modern analytics system. We will show you what the components of most modern data stacks are, and how best to put everything together.

This book is suitable for technical team members who are looking into setting up an analytics stack for their company for the very first time.

Book Content

Chapter 1: High-level Overview of an Analytics Setup

1.1 Start here - Introduction

What this book is about, who is it for and who is it not for.

1.2 A simple setup for people just starting out

If you're just starting out and don't need all the bells and whistles, you might be able to get going with this very simple setup.

1.3 A modern analytics stack

Next, what does a full analytics stack look like? We give you a high level overview of a modern analytics system, and lay out the structure of the rest of the book.

1.4 Our biases of a good analytics stack

Everyone has their own biases of how a good analytics setup looks like. Here are ours.

Chapter 2: Centralizing Data

Let's talk about the first step: collecting and storing data in a central place for analytics.

2.1 Consolidating data from source systems

If you have data from many source systems, it is important to consolidate them to a central place. In this section, we talk about the processes and tooling you'll need to do just that.

2.2 Understanding The Data Warehouse

Learn about the data warehouse, the central place to store and process all your data. Understand why the modern data warehouse is at the core of contemporary data analytics — why it's so important, and how to pick one.

2.3 ETL vs. ELT - What's the big deal?

Learn more about ETL, and its more modern cousin, ELT. Learn why we advocate for ELT over ETL in our book.

2.4 Transforming Data in the ELT paradigm

Learn how to turn raw data into clean, reliable and reusable data components, but within the ELT paradigm.

Chapter 3: Data Modeling for Analytics

Once your data sits nicely in your data warehouse, this is where the complex work begins. In this chapter, we talk about the task of processing data, including two intertwined operations: transforming & modeling data for analytics.

3.1 Data Modeling Layer & Concepts

In this chapter, we talk about a core concept in analytics: data modeling. We walk you through the basic concepts of data modeling in an ELT environment, and introduce the idea of a data modeling layer.

3.2 Kimball's Dimensional Data Modeling

We give you a brief overview of Ralph Kimball's ideas on dimensional data modeling, and walk you through a framework for applying them in the modern age.

Chapter 1: High-level Overview of an Analytics Setup

Start here - Introduction

You need analytics.

In today's business world, everyone needs analytics. Analytics powers and informs most decision making in organizations, from sales, marketing, partnership to product and engineering. Running a business without analytics is like driving in a foreign country without GPS.

Yet **most companies fumble when starting to build their analytics stack**. Many of them either spend too much time building a system that's unnecessarily complicated, or spend too little time building a system that doesn't work well.

This is understandable. When starting out, most companies don't have an experienced data engineer or architect to help them build things the right way. So they attempt to do it themselves; when they do their research online, they get lost in a sea of 'Big Data' fads, marketing jargon, and worse.

The questions:

- **How can I set up a simple, scalable analytics stack that serves my business?**
- **How can I start small but still follow best practices that help me scale the system up easily later?**

sound simple, but are actually difficult to answer if you're looking in the wrong places.

Our hope is that this book will help you answer the above questions.

This book aims to:

- Give you a high-level framework and understanding of a proper modern analytics setup, and how each component interacts with each other.
- Go into enough practical detail on each of the components. Explain the best practices, and help you understand the role of each component in the entire pipeline for data delivery: that is, consolidating, transforming, modeling, and using data.
- Show readers how to get started quickly on an analytics setup, yet remain able to scale it as time passes.

This book is not about what metrics to track for your industry. It is about how can you build an adequate system for your business to produce those metrics in a timely manner.

First, who are you and why should I trust you?

We are Holistics. We've been making data analytics tools for over four years, and helped more than a hundred companies build their business intelligence capabilities, sometimes from scratch.

A huge chunk of our time is spent educating and assisting companies as they migrate to this new world of cloud-based business intelligence tools. For the first time ever, we're putting that experience up for the world to read.

Who is this book written for?

This book is written for people who need a map to the world of data analytics.

The field of business intelligence has been around for about 60 years. It is incredibly confusing. There are many vendors, fads, trends, technologies and buzzwords in the market — and it's been this way for most of those six decades. It is impossible to expect new data professionals to be familiar with all that has come before, or to identify new developments as trends that will repeat in the future.

This book will give you the bare minimum you need to orient yourself in the contemporary BI environment. It assumes some technical knowledge, but won't get mired in technical detail.

Our goal is to give you 'just enough so you no longer feel lost'.

Who might find this useful? We can think of a few personas:

- **A junior data analyst (or product manager) with knowledge of SQL.** You have basic facility with data analytics, but do not yet have a full picture of your company's data stack. You find it difficult to talk to data engineers when you need them to help with a particular pipeline problem.
- **A software engineer who is assigned to set up a data stack from scratch.** You *think* you know what tools you need, but you're not sure if the stack you've chosen is the best for your company going forward. This book will give you a lay-of-the-land overview of the entire data analytics world, so you'll be able to pick the right components for your company.

- **An experienced data professional who wants a framework to understand the latest cloud-oriented developments.** You are experienced with business intelligence best practices, and cut your teeth during the heydays of Cognos dominance. You want to know what's up in this crazy new world of cloud data warehouses. You will skim this book where you are familiar, and you intend to slow down when you spot differences between what you know and what we present as the contemporary approach to analytics. You will find Chapters 2 and 3 the most interesting.

Who is this book NOT for?

This book is not written for non-technical business practitioners.

If you are a CEO, a project manager or a business team leader initiating a data analytics project for your company, it is best that you have a technical team member to help you go through the content presented in this book.

This book is also not written for experienced data engineers who manage large-scale analytics systems and want deeper knowledge about one particular problem. If you are familiar with cloud-first environments, you probably already know most, if not all of the content that is covered in this book. That said, you might still find some parts of the book useful as a refresher.

What you won't get from this book

There are a lot of things we won't be covering in the book.

As much as we'd like to, there's an entire topic on the human and organizational aspects of business intelligence that we won't cover in this book, which include questions like:

- How should I engage different stakeholders in the analytics process?
- How should I structure my data team?
- When is the right time to hire a head of analytics? What should I be looking for when I hire?
- How do I hire a data analyst?

These are questions that we hope to cover in another, dedicated book.

Additionally, we also won't cover:

- Industry-specific data knowledge (e.g. what is the standard metrics and best data practices for eCommerce industry?)
- Language-specific knowledge like Python or SQL (how to optimize queries, how to use different Python visualization packages ...)
- Data analysis techniques (how do you identify causal relationships, what different ways are there to verify a hypothesis?)

Let's start

Are you ready to read the book? If so, let's begin.

A simple setup for people just starting out

Before we begin, let's talk about a *minimum viable* analytics stack. We will then move on to a full treatment of a data analytics system in the next section.

The Three Things

Every data analytics system does three basic things.

1. You have to load data into a central repository of data, usually a data warehouse.
2. You have to transform or model your data so that it ends up nice and sparkly and ready for consumption in your data warehouse.
3. You have to get that data to business decision-makers, where it may be used.

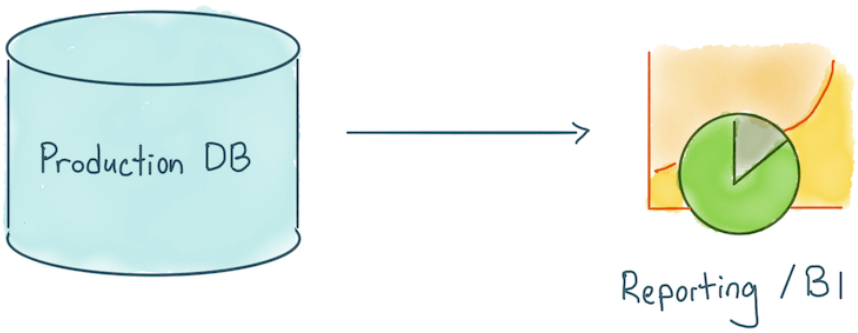
Querying from your production DB

If you are at a very early stage in your company's life and you fit the criteria below, it is likely that you can skip some of the steps above:

- If your data comes from only one source (which is most likely your production database) then you can **skip the data loading process**.
- If you are running a simple, low-traffic website and having an additional analytics workload on your production database will not make a huge dent on your application performance, then you can **skip the data warehouse**.

- If your raw data is simple enough to be visualized out of the box, or your reporting needs are so simple that you need no complex transformations, then you can **skip the data transform and modeling process**.

In short, your initial analytics setup can be very simple: just hook business intelligence tool up to the production database of your application.



When you interact with dashboards in your BI tool, the data will be queried live from your application database. As a result, you also happen to be getting the data in real-time.

Querying from a replica DB

The simple setup above, of course, is not all wonderful.

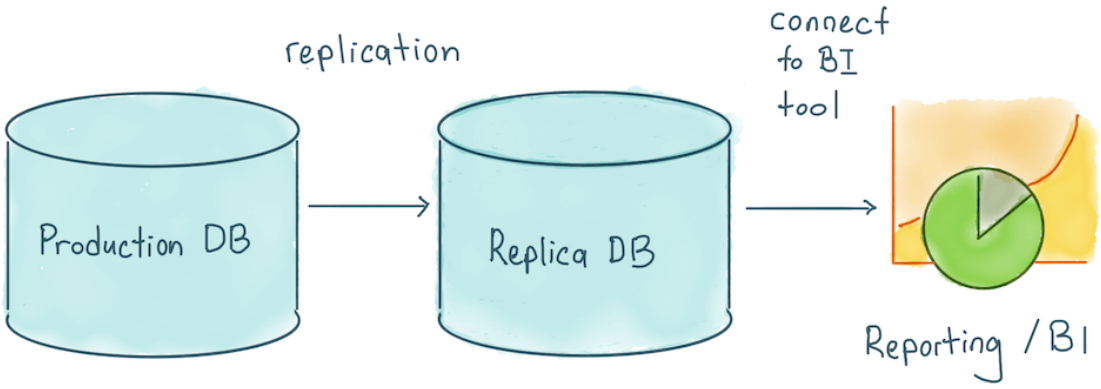
The highest risk you will face with the above setup is performance. Besides your normal production workload, your database now will

also take on an additional analytics workload, and may degrade your users' experience.

The easiest solution here is to set up a replica of your production database.

Check with your dev team to see if they have a replica that you can connect to. There's a good chance that your dev team has already set something up.

Then, instead of connecting to production, point your BI tool to your replica instead.



Of course, you could do some really bizarre things like export a dump, load that into a local database, and then query *that* — but most companies we know outgrow such a workflow in a matter of weeks.

When you connect to a replica, your production database will not be burdened by your analytical workload, while the data you receive

remains relatively fresh (depending, of course, on how your dev team configures the replication interval).

If your app runs on a NoSQL database

If you run on a NoSQL database (for instance, on something sexy like MongoDB or Cassandra), the above setup will not work for two reasons:

1. **Limited choice of reporting tool.**

   Since SQL has become the defacto standard for analytics, most BI tools are designed to work with SQL, so that limits the choice of BI tools you may pick.

   Also, there is **no standardized analytics interface** across different NoSQL databases, so you will end up looking for a specialized solution that is designed specifically for your brand of NoSQL database.

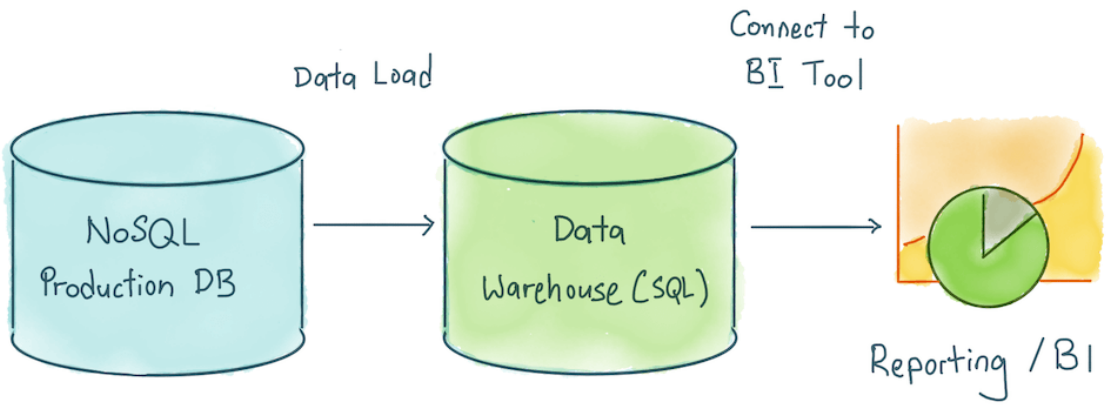
2. **Limited analytics capability.**

   Most NoSQL databases do not have strong support for analytics, both in terms of processing engine and querying interface. Try and write an aggregation query on a big MongoDB collection, and you will quickly understand why we say this. The query will be difficult to write, and the wait for results would be horrific.

   You might even get a surprise visit from your dev team for hogging up their production database.

In this situation, the recommended approach is again getting an SQL data warehouse, and then loading data from your NoSQL app into this

data warehouse. This moves us towards the direction of a proper analytics setup.



Alright, them's the basics. Now let's talk about the vast majority of data analytics stacks we see out there.

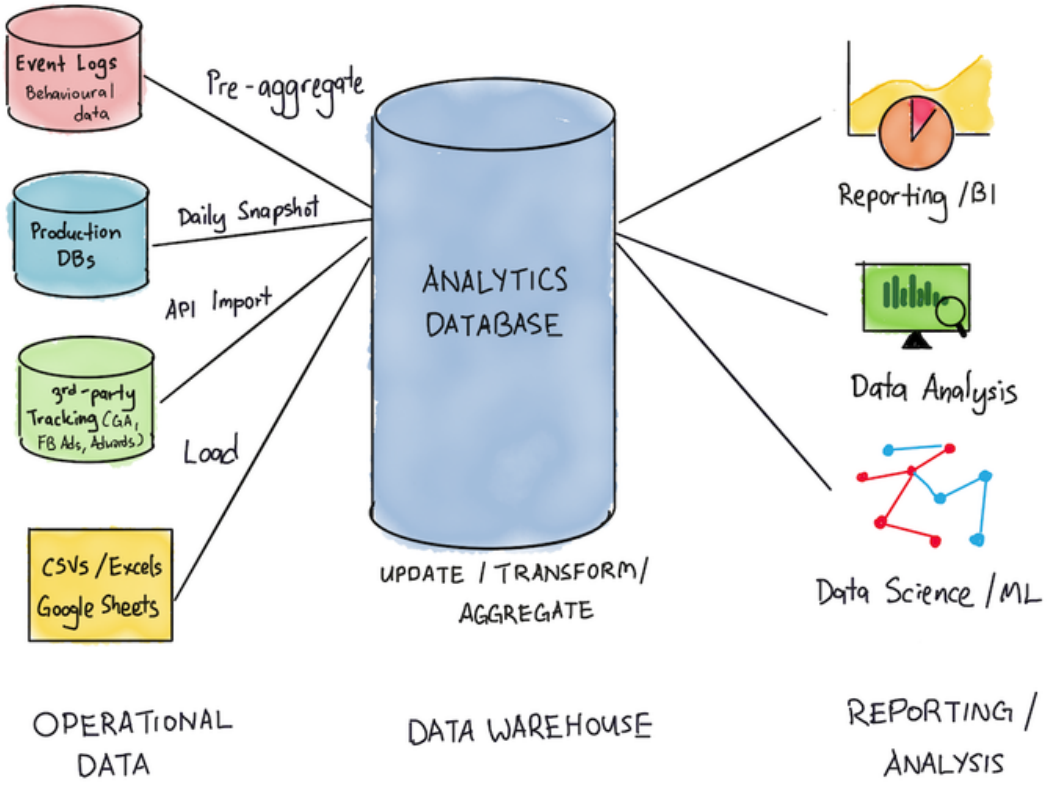
A modern analytics stack

In this chapter, we will talk about the most common setup for an analytics stack. Granted, you may see other data practitioners doing certain parts of this setup differently, but if you take a step back and squint, nearly all data analytics systems boil down to the same basic approach.

Let's get started.

In the previous section on minimum viable analytics, we mentioned that *all* analytical systems must do three basic things. We shall take that idea and elaborate further:

1. You must collect, consolidate and store data in a central data warehouse.
2. You must process data: that is, transform, clean up, aggregate and model the data that has been pushed to a central data warehouse.
3. And you must present data: visualize, extract, or push data to different services or users that need them.



This book is organized around these three steps. We shall examine each step in turn.

Step 1: Collecting, Consolidating and Storing Data

Before you may analyze your organization's data, raw data from multiple sources must be pulled into a central location in your analytics stack.

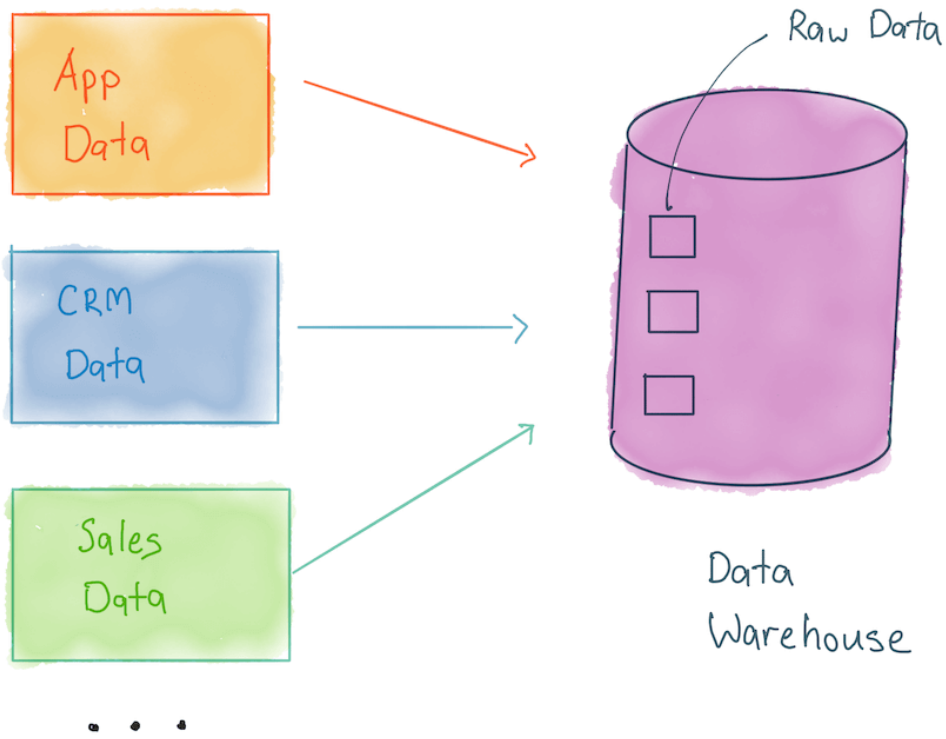
In the past, this may have been a 'staging area' — that is, a random server where everyone dumped their data. A couple of years ago, someone had the bright idea of calling this disorganized staging area a 'data lake'. We believe that the idea is more important than the name (and we also believe that a dump by any other name would smell just

as sweet) and so therefore encourage you to just think of this as a 'centralized location within your analytics stack'.

Why is consolidation important? Consolidation is important because it makes data easier to work with. Today, we encourage you to use an analytical database as your central staging location. Of course, you may choose to work with tools connected to multiple databases, each with different subsets of data, but we do not wish this pain on even our worst enemies, so we do not wish it on you.

Your central analytics database is usually powered by a data warehouse, which is a type of database that is optimized for analytical workloads. The process by which such consolidation happens is commonly called ETL (Extract Transform Load).

Chapter 2 of the book will go into more detail about this step.



Since we're talking about a big picture view in this chapter, there are only two key components you need to understand.

1. The Data Consolidating Process

This is when your raw source data is loaded into a central database.

If you're somewhat familiar with the analytics landscape, you might recall that this process is called ETL (Extract, Transform, Load).

In recent years, there has emerged a more modern approach, known as ELT (Extract-Load-Transform).

To discuss the nuances of our approach, we shall first talk about data consolidation in general, before discussing the pros and cons between

ETL and ELT. Yes, you're probably thinking "Wow! This sounds like a boring, inconsequential discussion!" — but we promise you that it isn't: down one path lies butterflies and sunshine, and down the other is pestilence and death.

In sum, we will use Chapter 2 to explore:

- How do you setup the data consolidation (Extract-Load) process? What ETL/ELT technology should you choose?
- Why the industry is moving from ETL to ELT. How is ELT different from ETL and why should we care?

2. The central analytics database, or "data warehouse"

This is the place where most of your analytics activities will take place. In this book we'll talk about:

- Why do you need a data warehouse?
- How do you set one up?
- What data warehouse technologies should you choose?

After going through the above two concepts, what you will get at the end of this step is:

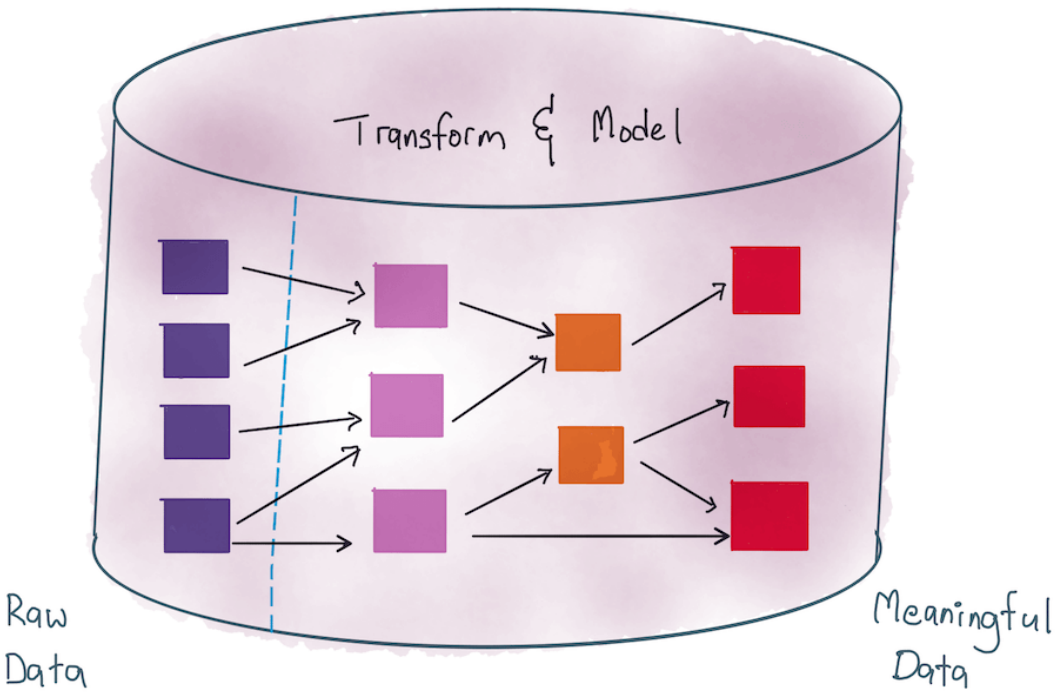
- You will have a data warehouse powerful enough to handle your analytics workload.
- You will have a process in place that syncs all raw data from multiple sources (CRM, app, marketing, etc) into the central data warehouse.

Once you have these two pieces set up, the next step is to turn your raw data into meaningful gold for analytics.

Step 2: Processing Data (Transform & Model Data)

This step is necessary because raw data is not usually ready to be used for reporting. Raw data will often contain extraneous information — for instance, duplicated records, test records, cat pictures, or metadata that is only meaningful to the production system — which is bad for business intelligence.

Therefore, you will usually need to apply a "processing step" to such data. You'll have to clean, transform and shape the data to match the logic necessary for your business's reporting.



This step usually involves two kinds of operations:

- Modeling data: apply business logic and formulae onto the data
- Transforming data: clean up, summarize, and pre-calculate data

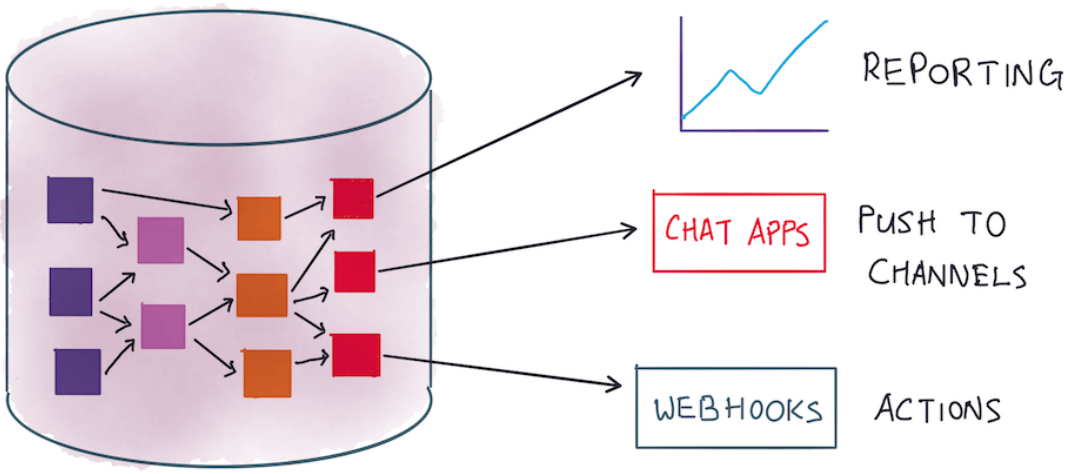
Chapter 3 goes into more detail about these two operations, and compares a modern approach (which we prefer) to a more traditional approach that was developed in the 90s. Beginner readers take note: usually, this is where you'll find most of the fun — and complexity! — of doing data analytics.

At the end of this step, you'll have a small mountain of clean data that's ready for analysis and reporting to end users.

Step 3: Presenting & Using Data

Now that your data is properly transformed for analytics, it's time to make use of the data to help grow your business. This is where people hook up a "reporting/visualization tool" to your data warehouse, and begin making those sweet, sweet charts.

Chapter 4 will focus on this aspect of data analytics.



Most people think of this step as just being about dashboarding and visualization, but it involves quite a bit more than that. In this book we'll touch on a few applications of using data:

- Ad-hoc reporting, which is what happens throughout the lifecycle of the company.
- Data reporting, which we've already covered.
- Data exploration: how letting end users freely explore your data lightens the load on the data department.
- The self-service utopia — or why it's really difficult to have *real* self-service in business intelligence.

Since this step involves the use of a BI/visualization tool, we will also discuss:

- The different types of BI tools.
- A taxonomy to organize what exists in the market.

Alright! You now have an overview of this entire book. Let's take a brief moment to discuss our biases, and then let's dive into data consolidation, in Chapter 2.

Our biases of a good analytics stack

Business intelligence and data analytics are fields that have been around for over 60 years, so clearly there are multiple approaches to building an analytics system.

Everyone will have their own preferences and biases. We do as well. Here are the biases that have shaped how we wrote this book:

- We prefer **ELT over ETL**
- We prefer using a **cloud data warehouse** over an on-premise data warehouse. We also prefer MPP analytics databases over Hadoop-like systems.
- We believe d**ata modeling is essential** in an analytics setup and should not be overlooked.
- We think that **SQL based analytics** will win over non-SQL based analytics.
- We believe that **analytics workflow/operations** is more important than a singular focus on visualizations.

Some of these terms might not be clear to you right now, but we will clarify what each of these terms and statements mean as we go deeper into the book.

For consistency, and for a more enjoyable reading experience, we will assume that the readers are on-board with these biases for the duration of this book.

However, throughout the book we will provide additional materials as well as our own arguments on the choices we have made. We believe that presenting our ideas this way makes it easier for you to evaluate and adapt certain aspects of our approach to your own practice later.

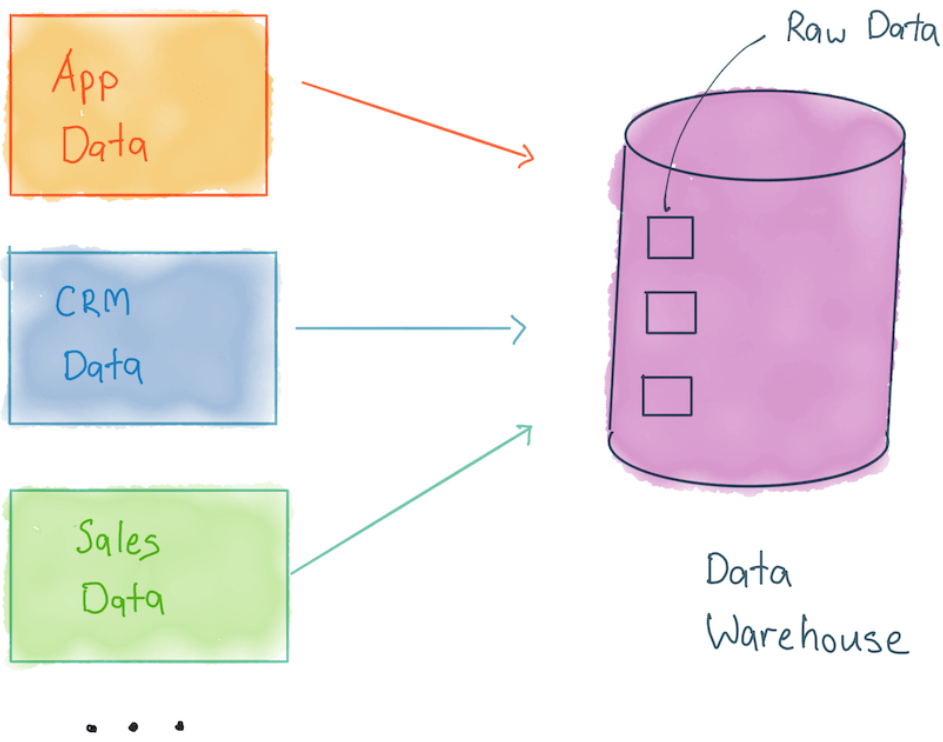
Onward!

Chapter 2:
Centralizing Data

Consolidating data from source systems

In a typical organization, data sits in many different systems. Such fractured data provide only small pieces of the whole puzzle. It is necessary to bring data from different sources to a centralized place in order to get at the big picture.

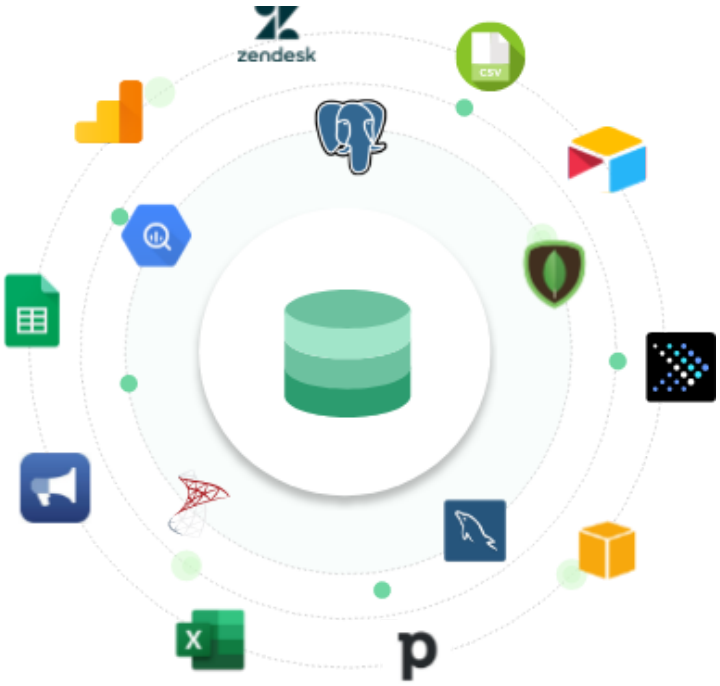
In this chapter, we will walk you through the basics of the data consolidation process.



Different types of source data

Let's start by talking about the different sources of data that a business might generate.

Since this book focuses on the technical aspect of building the analytics stack, the categorization below is meant to explore the difference from the technical point of view.



In a typical organization, it is common to have these three types of data sources:

1- Data coming directly out of your main application (application database)

If you are a company that does most of its business through a website or application (like Amazon, for instance), this is the data that exists within your main app. Such application databases typically contain all the transactional data that is critical to your business operations.

For example:

- An eCommerce store must have a database containing customers, products, orders and so on.
- A ride-hailing app must have a database containing customers, drivers, cars, and booking information

This data is usually stored in an SQL (or NoSQL!) database that is directly connected to your application. To access this data, you usually connect to it via a Database Query Tool.

2- Data coming from Customer Relationship Management (CRM), Content Management System (CMS), Enterprise Resource Planning (ERP), or marketing systems

These systems are third-party applications that are used in your organization's operations. For instance:

- CRM tools: Pipedrive, Salesforce, Zendesk
- Analytics: Google Analytics, Firebase, Amplitude
- Marketing: Facebook Ads, Google Ads, DoubleClick

These tools are often managed by a third-party. Usually, you **do not have direct access to the data stored within it**. You need to get the data via a company-provided API and store that data in your own data store.

3- Manual data created by employees and other systems

The third and last category is data that is created manually by employees in your organization.

This typically includes formats like Excel spreadsheets and Google Sheets, but may sometimes come in the form of CSV files, text documents, or (god-forbid) PDF reports.

Since this data is created by humans without enforced structures and rules, they are usually the most prone to error.

The central datastore for analytics

The above systems are places where you **store & transact data**, **not where you run analysis**. For that purpose, you will need a **data warehouse**.

A data warehouse is a central database to **store & process a large amount of data for analytical purposes**. You may think of it as the place where you dump a copy of all your data from the other source systems.

A data warehouse is nothing more than a database that is optimized for analytical processing. We'll dive deep on the data warehouse in the next section, but for now, let's stay focused on the data loading process.

The data loading process is the work of extracting data from multiple sources and loading them onto your data warehouse. This process is

also called **Extract & Load**. Let's look at that now.

The Extract & Load process

Extract & Load (EL) is quite a straightforward concept: a program is written to extract raw data from a data source, and that same data will be copied (loaded) over to a destination.

For example, the following pseudo-code loads booking data from source (a MySQL application) and copies them into an SQL data warehouse. It involves three steps:

- Run queries to extract all data from source table `bookings`
- Create the destination database table based on a specific structure.
- Load the data into the destination data warehouse.

```
source_db = connect_to_source_db();
dest_db = connect_to_dest_datawarehouse();


# Extract step
source_records = source_db.query(
  "SELECT id, email, user_id, listing_id, created_at FROM bookings"
);


# create the database table if not exists.
dest_db.query("
  CREATE TABLE IF NOT EXISTS dw.bookings (
    id integer,
    email varchar,
    user_id integer,
    listing_id integer,
    created_at datetime
  );
");
```

```
# Load step

for record in source_records {
  dest_db.query("
    INSERT INTO dw.bookings (id, email, user_id, listing_id, created_at)
    VALUES ( $1, $2, $3, $4, $5 )
  ", record )
}
```

(This is a crude implementation to illustrate a naive loading process)

The above script is relatively simple since it queries directly from an SQL database, and it doesn't handle things like failure, smart retries, and so on. This script can then be set up to run every day in the early morning time.

In actual production, your script will be a lot more complicated, since there are also other considerations like performance optimization, failure handling, source systems interface and incremental loading.

Over time, the human cost to maintain your scripts will far out-weight the actual value it brings you.

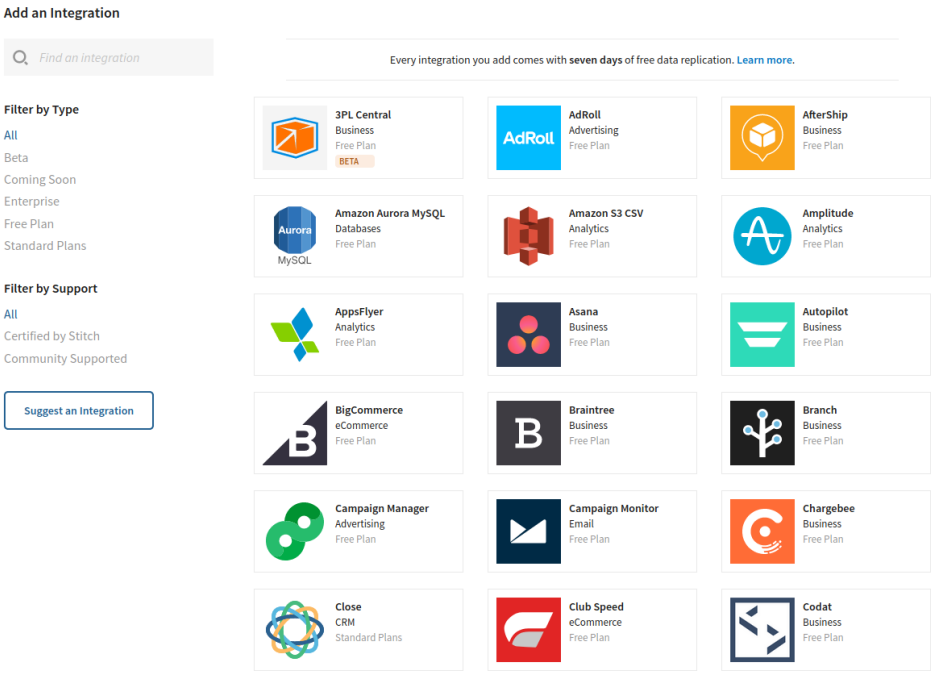
That's why it is usually better to consider adopting an existing data load tool instead. The only exception here is if your business has specialized needs, or if you operate at the scale of big tech companies like Netflix, Google and Facebook.

Using a Data Loading Tool

The good news is that there are a large number of free and paid data loading tools in the market. These tools often behave in a plug-and-play manner. They provide a user-friendly interface to connect to your data sources and data storage, set loading intervals and load modes, and they likely also deliver reports about all your data loading jobs.

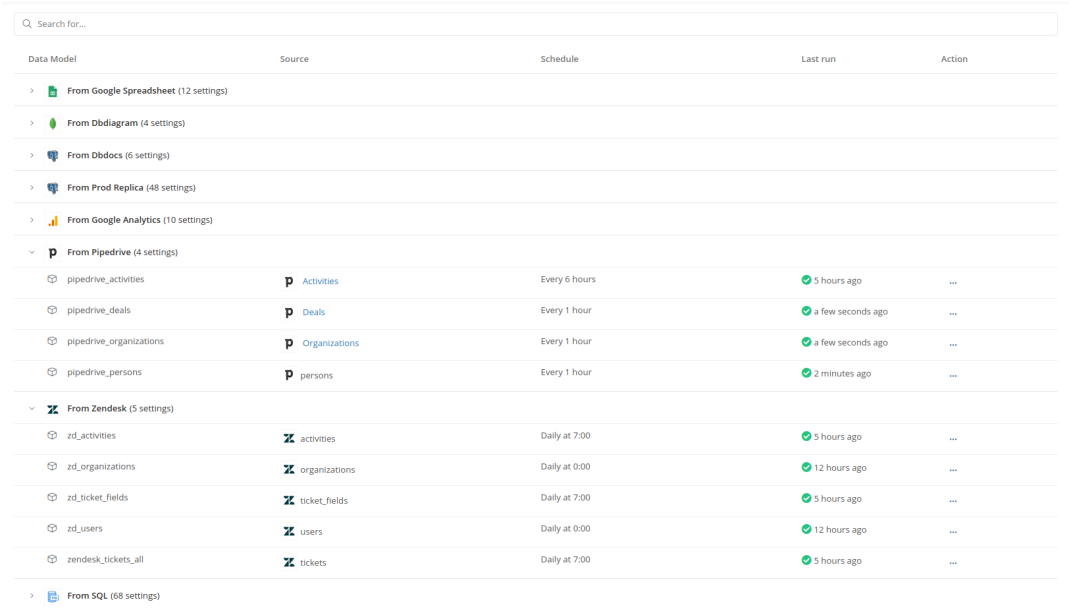
These data load tools are commonly known as **ETL tools.** This might cause confusion to beginner readers though, since most of the modern tools we look at don't do the Transform step, and ask you to use a dedicated Transformation tool.

Paid tools like StitchData or Talend often boast a large selection of data integrations, from web analytics platforms, Software-as-a-Service web applications, and NoSQL and SQL databases alike.



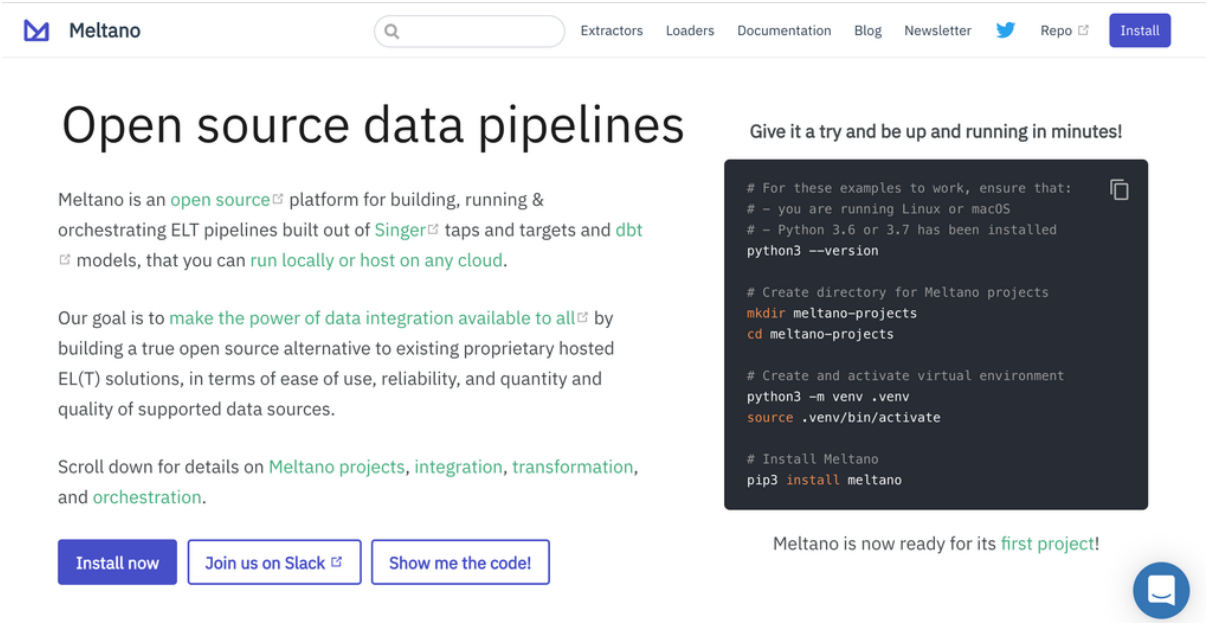
That isn't to say that you need such sophisticated tools all the time, though. Basic data loading capabilities are also often bundled in data

analytics platforms like Holistics, Google Data Studio, and Tableau. Though the number of integrations are often not as extensive as dedicated tools, they are sometimes enough for basic reporting needs.



If you don't want to go for one of the more comprehensive, paid options, you may also choose to go for open-source software. Particularly famous packages include Airflow and Prefect. As with all open-source software, you will likely need to spend some time setting things up and integrating such tools into your systems.

A recent example of this category that we find particularly interesting is Meltano — a platform by GitLab that focuses on providing open-source data pipelines.



It will be interesting to see if Meltano can galvanize an open-source community around it. If they succeed, we would have a large set of data loaders for as many services as possible under the sun.

Anyway, let's wrap up. These are some proprietary data loading tools on the market for you to consider:

- Alooma
- HevoData
- StitchData
- Talend
- Pentaho

And here are a couple of great open source options (though — as with all things open source, caveat emptor):

- Prefect
- Airflow

- Meltano
- Singer (on which Meltano, above, is built on)

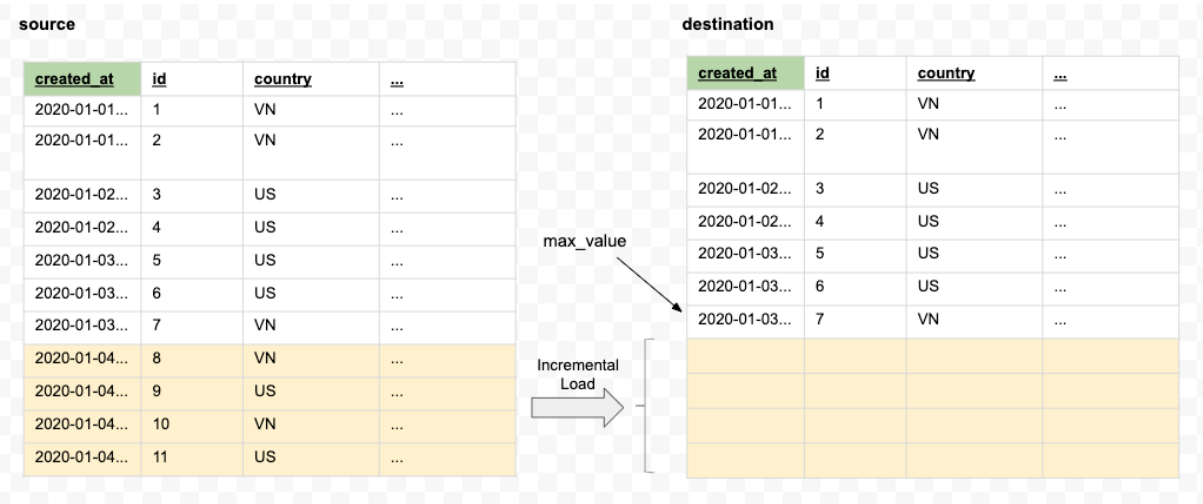
If you still need to be convinced that writing your own data load & ETL scripts is a bad idea, check out this great article by Jeff Magnusson, who wrote it in his capacity as the VP of Data Platform at Stitch Fix.

Common Concepts

How do I load data incrementally?

One common concept worth mentioning is the concept of incremental load, which is the notion of loading data (you guessed it!) incrementally to your data warehouse. As your source systems grow in volume, this incremental load concept will become more important to ensure your system runs smoothly.

Let's revisit the earlier example of loading bookings data, but this time, let's look at how to run this transformation incrementally.



We can see that when an incremental loading job runs, only data for `2020-01-04` will be queried and copied over to the new table.

To reflect this in the above pseudo-code, there's an additional step we have to write in order to grab the most recently created bookings.

```
source_db = connect_to_source_db();
dest_db = connect_to_dest_datawarehouse();

max_value = dest_db.query("SELECT max(created_at) FROM bookings")

# Extract step - this time we only extract recent records
source_records = source_db.query("
  SELECT id, email, user_id, listing_id, created_at
  FROM bookings
  WHERE created_at > $1", max_value
");

# the rest of the load step happens normally.
...
```

Most industry-standard data load tools should have support for incremental load.

How much performance gain does incremental load get you? A lot.

Imagine that you have 100M booking records and that those records are growing at a pace of 10,000 records a day:

- With incremental load: you copy *10,000 records* each day.
- Without incremental load: you process *100M records (and more as time goes by!)* each day.

This is a 10,000 times difference in load.

How often should I perform data loading?

Based on our own experience, most analytics use cases in most businesses just need a daily refresh of data.

It's also important to note that unless your use case absolutely requires it, it is not very important to get real-time data in most business analytics. To understand why, think about this: If you want to view sales data over the last seven weeks, is it necessary for the data to account up to the minute you're requesting it?

> Most business use cases just need a daily refresh of analytics data.

A common setup that we see is that the organization has a pipeline that runs after midnight and finishes before people get to work. This is so that when business users login in the morning, all of their analytics reports are refreshed with yesterday's data.

Summary

- There are three different types of source data systems: application data, third-party data and manual data.
- To store and process data for analytics, you need a thing called data warehouse.
- The process to move data from source to destination is called Extract & Load.
- We went over how the Extract & Load process looks like in practice, and recommend that you use off-the-shelf tools. We also

talk about how incremental load can help you increase the performance of your EL process.

In the next section, we talk about the next logical piece of the puzzle: the Understanding The Data Warehouse.

Understanding The Data Warehouse

In the previous section we spoke about the process of consolidating (Extract & Load) data from multiple source systems into your analytics database. In this post, we'll talk specifically about your analytics database, i.e your data warehouse.

What is a data warehouse?

A data warehouse is a type of analytics database that stores and processes your data for the purpose of analytics. Your data warehouse will handle two main functions of your analytics: **store your analytical data & process your analytical data**.

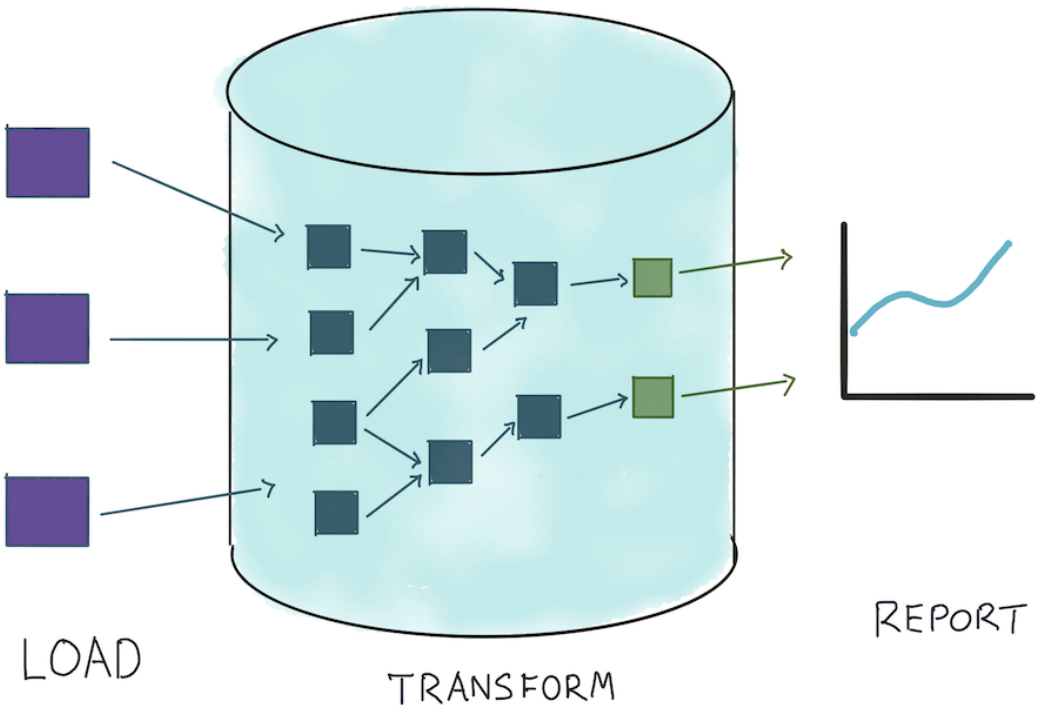
Why do you need one? You will need a data warehouse for two main purposes:

1. First, you can't combine data from multiple business functions easily if they sit in different sources.
2. Second, your source systems are not designed to run heavy analytics, and doing so might jeopardize your business operations as it increases the load on those systems.

Your data warehouse is the centerpiece of every step of your analytics pipeline process, and it serves three main purposes:

- **Storage:** In the consolidate (Extract & Load) step, your data warehouse will **receive** and **store data** coming from multiple sources.

- **Process:** In the process (Transform & Model) step, your data warehouse will handle most (if not all) of the **intensive processing** generated from the transform step.
- **Access:** In the reporting (Visualize & Delivery) step, reports are being **gathered within the data-warehouse** first, then visualized and delivered to end-users.



At the moment, most data warehouses use SQL as their primary querying language.

When is the right time to get a data warehouse?

The TL;DR answer is that *it depends.* It depends on the stage of your company, the amount of data you have, your budget, and so on.

At an early stage, you can probably get by without a data warehouse, and connect a business intelligence (BI) tool directly to your production database (As we've mentioned in A simple setup for people just starting out).

However, if you are still not sure if a data warehouse is the right thing for your company, consider the below pointers:

- **First, do you need to analyze data from different sources?**

  At some point in your company's life, you would need to combine data from different internal tools in order to make better, more informed business decisions.

  For instance, if you're a restaurant and want to analyze orders/waitress ratio efficiency (which hour of the week the staff is most busy vs most free), you need to combine your sales data (from POS system) with your staff duty data (from HR system).

  For those analyses, it is a lot easier to do if your data is located in one central location.

- **Second, do you need to separate your analytical data from your transactional data?**

  As mentioned, your transactional systems are not designed for analytical purposes. So if you collect activity logs or other

potentially useful pieces of information in your app, it's probably not a good idea to store this data in your app's database and have your analysts work on the production database directly.

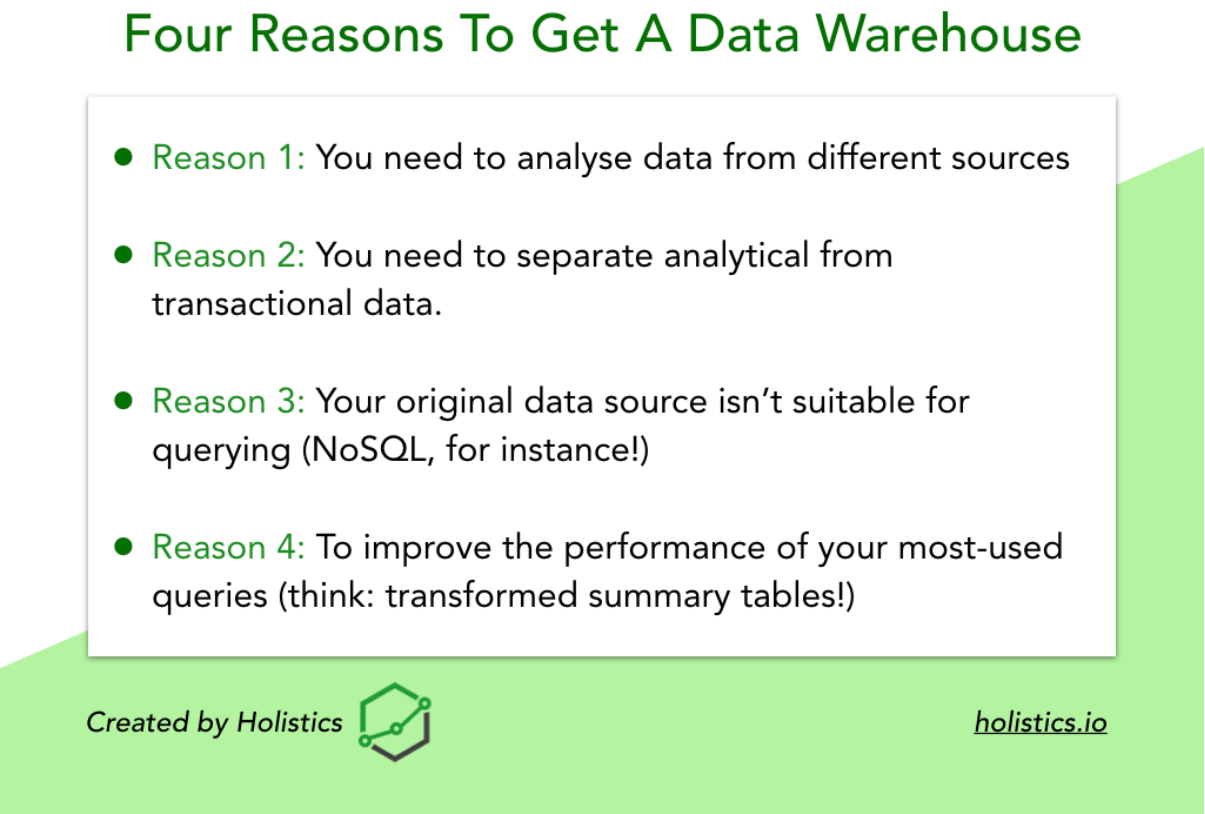
Instead, it's a much better idea to purchase a data warehouse — one that's *designed* for complex querying — and transfer the analytical data there instead. That way, the performance of your app isn't affected by your analytics work.

- **Third, is your original data source not suitable for querying?**

  For example, the vast majority of BI tools do not work well with NoSQL data stores like MongoDB. This means that applications that use MongoDB on the backend *need* their analytical data to be transferred to a data warehouse, in order for data analysts to work effectively with it.

- **Fourth, do you want to increase the performance of your analytical queries?**

  If your transactional data consists of hundreds of thousands of rows, it's probably a good idea to create summary tables that aggregate that data into a more queryable form. Not doing so will cause queries to be *incredibly* slow — not to mention having them being an unnecessary burden on your database.



If you answered yes to any of the above questions, then chances are good that you should just get a data warehouse.

That said, in our opinion, it's usually a good idea to just go get a data warehouse, as data warehouses are not expensive in the cloud era.

Which Data Warehouse Should I Pick?

Here are some common data warehouses that you may pick from:

- Amazon Redshift
- Google BigQuery
- Snowflake

- ClickHouse (self-hosted)
- Presto (self-hosted)

If you're just getting started and don't have a strong preference, we suggest that you go with Google BigQuery for the following reasons:

- **BigQuery is free for the first 10GB storage and first 1TB of queries**. After that it's pay-per-usage.
- **BigQuery is fully managed (serverless)**: There is no physical (or virtual) server to spin up or manage.
- **As a result of its architecture, BigQuery auto-scales:** BigQuery will automatically determine the right amount of computing resources to allocate to each query, depending on the query's complexity and the amount of data you scan, without you having to manually fine-tune it.

(Note: we don't have any affiliation with Google, and we don't get paid to promote BigQuery).

However, if you have a rapidly increasing volume of data, or if you have complex/special use cases, you will need to carefully evaluate your options.

Below, we present a table of the most popular data warehouses. Our intention here is to give you a high-level understanding of the most common choices in the data warehouse space. This is by no means comprehensive, nor is it sufficient to help you make an informed decision.

But it is, we think, a good start:

Data Warehouse Brief Comparison

Aa Name	Developer	Pricing & Delivery
Amazon Redshift	Amazon, as part of AWS offering	Pay per instance. Starts at $0.25/hr (~$180/month)
Google BigQuery	Google, as part of Google Cloud offering	Pay per data queried & stored. Starts at $0, free first 10GB storage & 1TB queried.
ClickHouse	Developed in-house at Yandex, later open-sourced.	Free & Open-source. Deploy on your own server.
Snowflake	Snowflake (company)	Pay per usage. Check website for more info. Cloud-based (on AWS, GCP or Azure).
Presto	Developed in-house at Facebook, later open-sourced. Now managed by Presto Foundation (part of Linux Foundation).	Free & open source. Deploy on your own server.

What makes a data warehouse different from normal SQL database?

At this point some of you might be asking:

"Hey isn't a data warehouse just like a relational database that stores data for analytics? Can't I just use something like MySQL, PostgreSQL, MSSQL or Oracle as my data warehouse?"

The short answer is: yes you can.

The long answer is: it depends. First, we need to understand a few concepts.

Transactional Workloads vs Analytical Workloads

It is important to understand the difference between two kinds of database workloads: transactional workloads and analytical workloads.

A **transactional workload** is the querying workload that serves normal business applications. When a visitor loads a product page in a web app, a query is sent to the database to fetch this product, and return the result to the application for processing.

```
SELECT * FROM products WHERE id = 123
```

(the query above retrieves information for a single product with ID 123)

Here are several common attributes of transactional workloads:

- Each query **usually retrieves a single record, or a small amount** of records (e.g. get the first 10 blog posts in a category)
- Transactional workloads typically involve simple queries that **take a very short time to run** (less than 1 second)
- **Lots of concurrent queries** at any point in time, limited by the number of concurrent visitors of the application. For big websites this can go to the thousands or hundreds of thousands.
- Usually interested in the **whole data record** (e.g. every column in the product table).

Analytical workloads, on the other hand, refer to workload for analytical purposes, the kind of workload that this book talks about. When a data report is run, a query will be sent to DB to calculate the results, and then displayed to end-users.

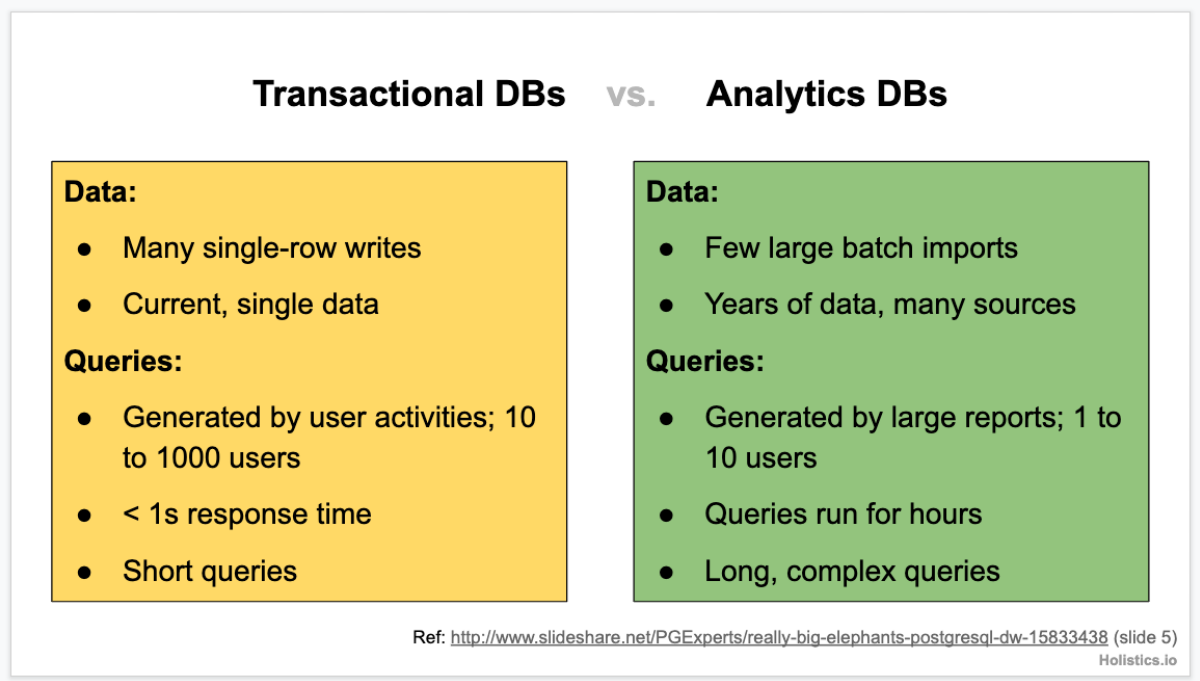
```
SELECT
  category_name,
  count(*) as num_products
FROM products
GROUP BY 1
```

(The above query scans the entire products table to count how many products are there in each category)

Analytical workloads, on the other hand, have the following attributes:

- Each query typically **scans a large number of rows** in the table.
- Each query is **heavy and takes a long time** (minutes, or even hours) to finish
- **Not a lot of concurrent queries** happen, limited by the amount of reports or internal staff members using the analytics system.
- Usually interested in **just a few columns** of data.

Below is a comparison table between transactional vs analytical workload/databases.



Transactional workloads have many simple queries, whereas analytical workloads have few heavy queries.

The Backend for Analytics Databases is Different

Because of the drastic difference between the two workloads above, the underlying backend design of the database for the two workloads are very different. Transactional databases are optimized for fast, short queries with high concurrent volume, while analytical databases are optimized for long-running, resource-intensive queries.

What are the differences in architecture you ask? This will take a dedicated section to explain, but the gist of it is that analytical databases use the following techniques to guarantee superior performance:

- **Columnar storage engine:** Instead of storing data row by row on disk, analytical databases group columns of data together and store them.
- **Compression of columnar data:** Data within each column is compressed for smaller storage and faster retrieval.
- **Parallelization of query executions:** Modern analytical databases are typically run on top of thousands of machines. Each analytical query can thus be split into multiple smaller queries to be executed in parallel amongst those machines (divide and conquer strategy)

As you can probably guess by now, MySQL, PostgreSQL, MSSQL, and Oracle databases are designed to handle transactional workloads, whereas data warehouses are designed to handle analytical workloads.

So, can I use a normal SQL database as my data warehouse?

Like we've said earlier, yes you can, but it depends.

If you're just starting out with small set of data and few analytical use cases, it's perfectly fine to pick a normal SQL database as your data warehouse (most popular ones are MySQL, PostgreSQL, MSSQL or Oracle). If you're relatively big with lots of data, you still can, but it will require proper tuning and configuring.

That said, with the advent of low-cost data warehouse like BigQuery, Redshift above, we would recommend you go ahead with a data warehouse.

However, if you *must* choose a normal SQL-based database (for example your business only allows you to host it on-premise, within

your own network) we **recommend going with PostgreSQL** as it has the most features supported for analytics. We've also written a detailed blog post discussing this topic here: Why you should use PostgreSQL over MySQL for analytics purpose.

Summary

In this section, we zoomed in into data warehouse and spoke about:

- Data warehouse is the central analytics database that stores & processes your data for analytics
- The 4 trigger points when you should get a data warehouse
- A simple list of data warehouse technologies you can choose from
- How a data warehouse is optimized for analytical workload vs traditional database for transactional workload.

ETL vs. ELT - What's the big deal?

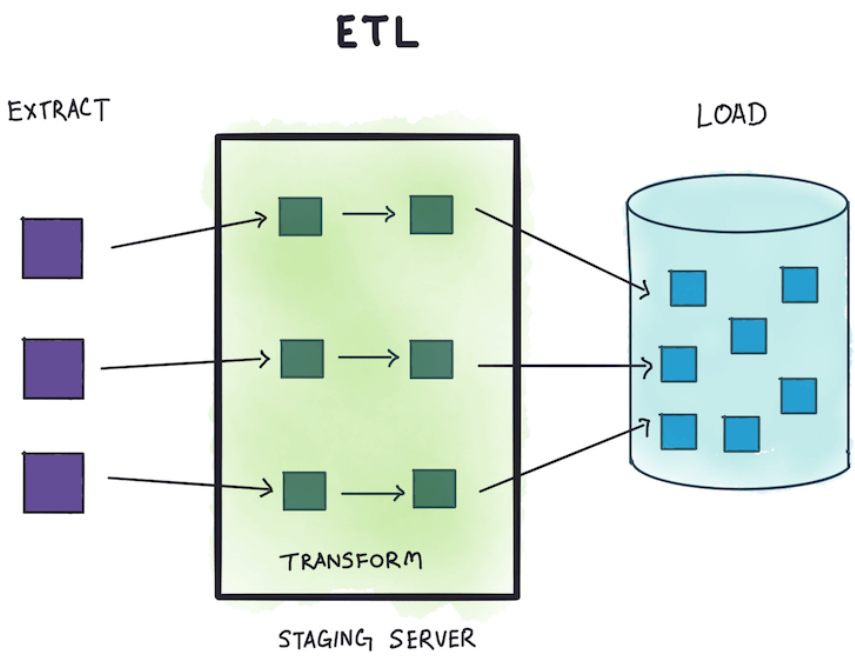
In the previous sections we have mentioned two terms repeatedly: ETL, and ELT. In this section, we will dive into details of these two processes, examine their histories, and explain why it is important to understand the implications of adopting one versus the other.

Do note that some of this will make more sense after you read Transforming Data in the ELT paradigm (in chapter 3).

The ETL process

In any organization's analytics workflow, the most intensive step usually lies in the data preparation step: that is, combining, cleaning, and creating data sets that are ready for business consumption and decision making.

This function is commonly known as ETL (Extract, Transform, and Load), which identifies the three distinct stages involved.



In this process, an ETL tool extracts the data from different data source systems, transforms the data by applying calculations, concatenations, and the like, and finally loads the data into the data warehouse.

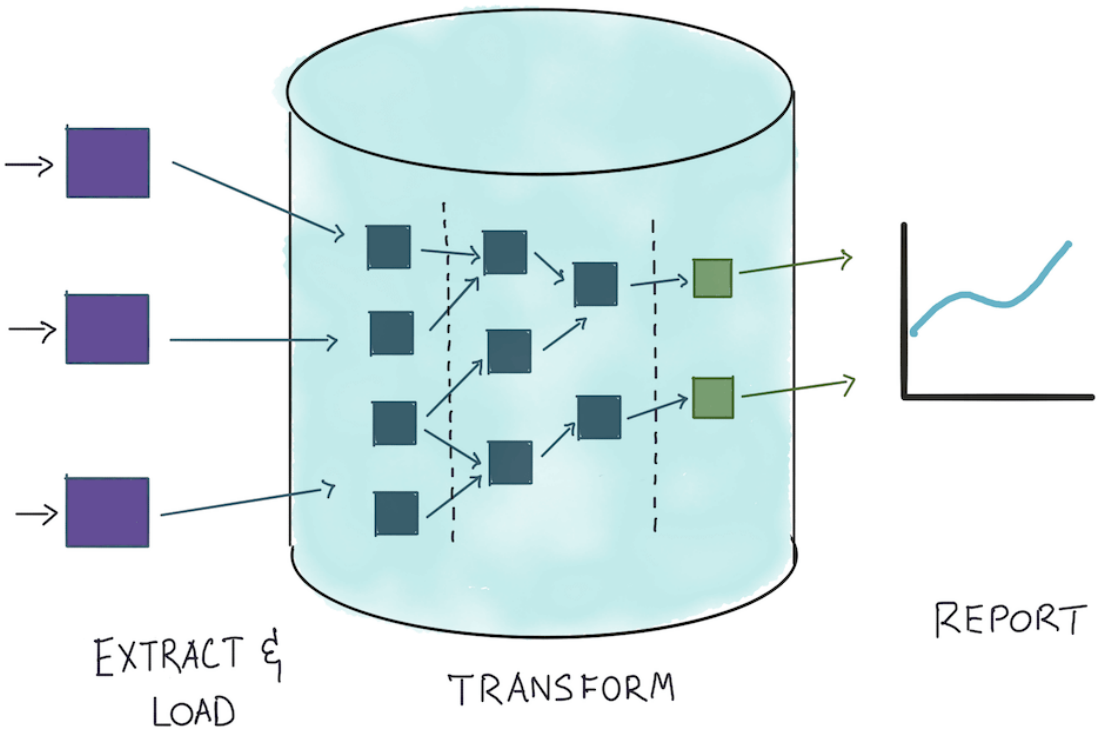
What happens in this approach:

1. You extract data from sources (write queries on your SQL databases, or send data extract requests to an application via its API).
2. Extracted data will be transformed in the ETL tool's memory.
3. Transformed data is then loaded in the final data storage/warehouse.

The key things to note here is that raw data is transformed **outside of the data warehouse**, usually with the help of a dedicated "staging server"; and that only *transformed* data is loaded into the warehouse.

The ELT process

ELT is a different way of looking at this problem. Instead of transforming the data *before* it is loaded into the database, ELT does the transformation within the data warehouse. Your data will be loaded into the data warehouse *first*, and then transformed in place.



What happens in this approach:

1. You extract data from sources.
2. Instead of transforming in-memory, using a pipelining tool, you load the raw, extracted data straight into the destination data storage/warehouse.
3. Finally, you perform any necessary transformations within your data warehouse

The key things to note here are that raw data is transformed inside the data warehouse *without* the need of a staging server; your data warehouse now contains both raw data and transformed data.

The shift from ETL to ELT

Historically, building a data warehouse was a very expensive undertaking, both on the hardware side and on the software side. The server costs, implementation costs and software licenses for a data warehousing project 20 to 30 years ago could easily go up to the millions of dollars and take months to implement.

- Since data warehouses were so expensive, to save on cost, people would only want to load clean, properly transformed and aggregated data into the data warehouse.
- Practitioners were still following waterfall development models back then, so it was acceptable to take the time to plan out and perform proper transformations.

In this context, the ETL model made perfect sense: raw data was properly transformed in a staging server (or ETL pipelining tool) before being loaded into your ridiculously expensive data warehouse. The volume of data that was handled by such tools back then was relatively small, and thus manageable for most staging servers to handle.

But the ETL approach has a number of drawbacks when viewed through a more contemporary lens:

- Every new set of transformations would require involvement from IT or from data engineering, in order to code up new

transformations. The ETL tools used in the old paradigm were hardly accessible to data analysts after all, who would traditionally come from an SQL background. As a result, data analysts relied on data engineering for access to new transformed data, and would often have to wait for *days* before they could get to implement new reports.

- As data sizes increased, the ETL approach became more and more problematic. Specifically, the staging server — that is, the machine that orchestrated all the loading and transforming of data — began to be a bottleneck for the rest of the stack.

So what changed? Well, a couple of things emerged in the 2010s that made an alternative approach possible:

- First, we saw the commoditization of the cloud data warehouse. Modern data warehouses today can store and process a *very* large amount of data at very little cost.
- We also saw an explosion in the amount and in the variety of data being collected. Some of us have heard of this change as the 'big data revolution' — which was a fad in the mid 2010s. The end result of that fad, however, was good for all of us: it pushed the development of new tools and new approaches to data, all of which were built around the assumption of needing to deal with terabyte-level data volumes at a minimum.
- And finally, we saw the rise of lean and agile software development practices. Such practices meant that people began to expect more from their data departments, the same way that they were used to quick execution speeds in their software development teams.

And so at some point, people began to realize: **the cost of storing and processing data had become so cheap, it was now a better idea to just**

dump all your data into a central location, before applying any transformations.

And thus lay the seed that grew into the ELT approach.

In contrast to ETL, an ELT approach has a number of advantages:

- It removes the performance bottleneck at the staging server/ETL pipelining tool. This is significant because data warehouses had increased in processing power at a level far beyond the most advanced ETL pipelining tool. The ELT approach assumes a powerful data warehouse at its core.
- It does not demand detailed planning on what data to transform beforehand. Data practitioners began to take a more agile approach to analytics, aka "dump first, transform later".
- With proper transform and modeling tools, ELT did not require data engineers to be on standby to support any transformation request from the analytics team. This empowered data analysts, and increased execution speed.

As stated in Our biases of a good analytics stack , we favor ELT over ETL, and we believe that all modern analytics stacks should be set up this way.

Below is a short table to summarize the differences between ETL and ELT.

Name	ETL	ELT
History	- Data warehouse cost is very expensive (millions of dollars) - Data volume is still manageable. - People are forced to practice waterfall development.	- Cloud data warehouse drives the cost of storing and processing data down significantly (hundreds/thousands of dollars only) - Data volume explode. - Agile practices are possible.
Process	Raw data is transformed in a staging server. Only transformed data is loaded into the data warehouse. Transformations rely on the server's processing power.	Raw data is loaded into the data warehouse. Transformations are done within the data warehouse. Results are also stored within the data warehouse. Transformations rely on data warehouse processing power.
Pros/Cons	Data warehouse only contains cleaned, transformed data ⇒ maximize utilization of data warehouse. Doesn't work well when data volume increase ⇒ bottlenecks on the staging server. Usually take weeks/months to change process due to waterfall approach.	All data is stored in the cloud data warehouse ⇒ very easy to change up new data warehouse. Doesn't need additional staging servers. Assuming a modern data warehouse, works well when data volume increases. Takes only days to transform/introduce new data.

What About Data Lakes?

At this point, it's worth asking: what about data lakes? How does that fit into the ELT vs ETL paradigm that's we've just discussed?

Let's back up a moment.

A data lake is a fancy term for a central staging area for raw data. The idea is to have everything in your organization dumped into a central lake, before loading it into your data warehouse. Unlike data warehouses (which we have talked about extensively in our discussion about ELT, above) lakes are often object buckets in which you may

upload all manner of unstructured data: examples of buckets are services like AWS S3 or Google Cloud Storage; examples of unstructured data are CSV dumps or even text files, exported from various source systems.

It is important to understand that a data lake is not a new idea. Since the 80s, business intelligence projects have usually included a staging area for data. ETL systems would then take data from that staging area and transform it within the tool, before loading it into data warehouses. The only thing that is new here is the term itself — and that term harkens back to a 2010 blog post by Pentaho CTO James Dixon.

We have no strong feelings about data lakes. The point of ELT is to load the unstructured data into your data warehouse *first*, and then transform *within*, rather than transforming data in-flight through a pipelining tool. Whether this raw data sits in an object bucket before loading is of little concern to us.

Summary

To conclude, when you are picking analytics tools, ask yourself: does this tool assume an ETL approach, or does it assume an ELT approach? Anything that requires a data transformation step *outside* the data warehouse should set off alarm bells in your head; it means that it was built for the past, not the future.

Pick ELT. As we will soon see in Chapter 3, ELT unlocks a **lot more** than just the operational advantages we've talked about above.

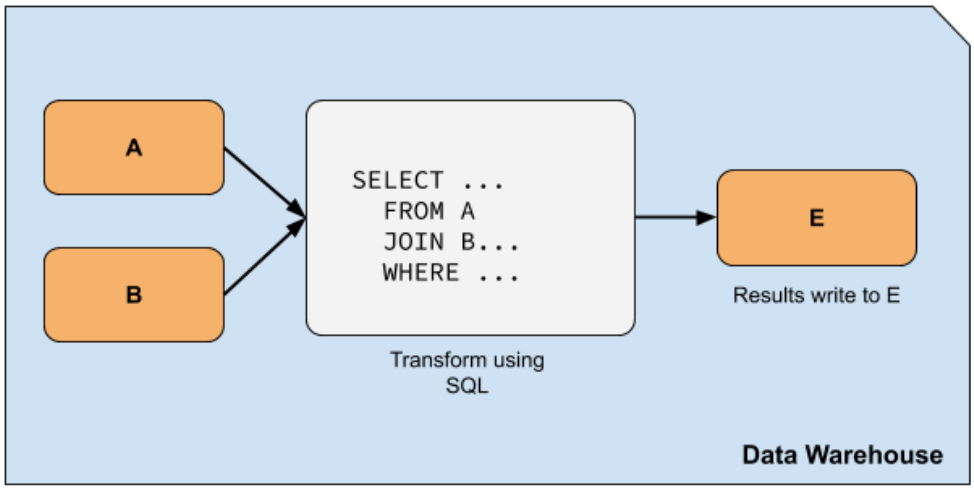
Transforming Data in the ELT paradigm

What is data transformation?

Data transformation is a process that changes the structure of the existing data into some other structures (thus '*transform'*).

Common use cases of data transformations include:

- **Data cleaning:** You correct, remove or update inaccurate records from a recordset.
- **Aggregate data:** You aggregate data into a more summarized version of itself. For example, calculating transaction counts by different region and category, and storing that into a new table.
- **Pre-computation:** You calculate new numbers according to a business formula during a transformation step, turning raw data into a business measure (e.g. You calculate a ratio, or a trend).



Example of transformation that happens inside a data warehouse using SQL

Why do I need to transform data?

Raw data that is piped into your data warehouse is usually in a format designed for storing transactional information. To analyze this data effectively, a transformation step is needed to make it easier to work with. Some people call this 'data modeling'. We will talk about modeling in the next chapter, but note here that data transformation includes more than just modeling.

Applying good data transformations will yield the following benefits:

- **Reusability:** Think of each data transform as a data component that expresses some business logic, and that this data component may be reused multiple times in different reports and analyses.
- **Your reports and analyses will be more consistent**: Because of the above reusability property, when it comes to reporting, instead of rewriting your SQL logic to multiple reports, the logic is written in

just one transform and is reused in multiple reports. This helps you avoid the scenario where two different data reports produce two different numbers for the same metric.

- **Improve overall runtime performance:** If data is transformed and aggregated, the amount of computation you will need to do down the road only happens once at the time of running. This reduces report processing time and improves performance significantly.
- **Cost effectiveness:** Less repeated computation will lead to lower processing and server costs overall.

Implementing data transformation in the ELT paradigm

In the past, data transformation was often done by an ETL tool, before the loading process into the data warehouse. This meant significant data engineering involvement, as it was the engineers who created and maintained such transformation pipelines. The pipelines were also often implemented in some programming or scripting language.

It is no accident that we introduced the concept of ELT instead, which has gained traction alongside the growth of more powerful data warehouses. In this paradigm, data is loaded into a data warehouse *before* it is transformed. This allows us to do two things.

First, it allows us to write transformations using SQL alone. A transformation is thus the creation of a new table that will be created within the same data warehouse, storing the results of that transform step.

Second, it allows analysts to write transformations. This removes the dependency on data engineering, and frees us from building and maintaining pipelines in external tools. (We've already covered this in ETL vs. ELT - What's the big deal?, so we recommend that you read that if you haven't done so already.)

Let's talk about what this actually looks like.

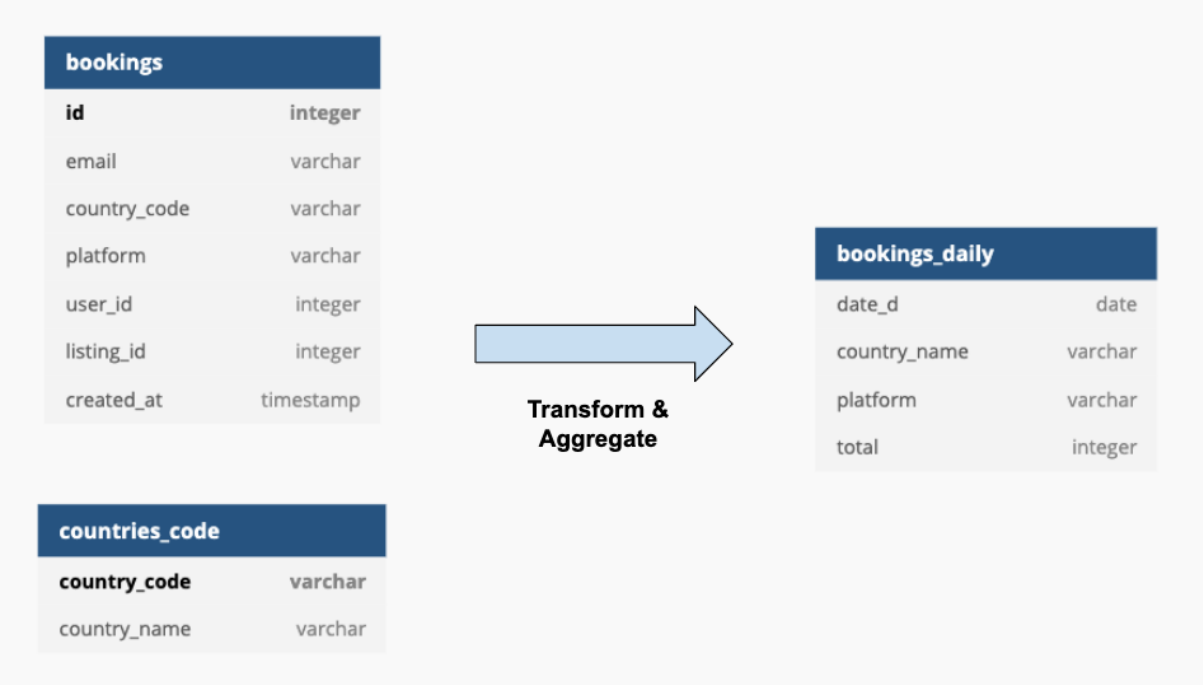
Imagine that we're running a hotel booking website, and want to create a summarization of daily bookings according to a few dimensions. In this case, we want to look at dimensions like country, as well as the platform on which the booking was made.

Our raw bookings data would look like this after being loaded into the data warehouse:

```
Table bookings {
  id integer [pk]
  email varchar
  country_code varchar
  platform varchar
  user_id integer
  listing_id integer
  created_at timestamp
}

Table countries_code {
  country_code varchar [pk]
  country_name varchar
}
```

(syntax written using DBML)

We want to transform them into a `bookings_daily` table like so.



Source

So to implement this, the code for the transform job will be:

```
-- Transform: Summarize bookings by country and platform
BEGIN;
DROP TABLE IF EXISTS bookings_daily;
CREATE TABLE bookings_daily (
  date_d date,
  country_name varchar,
  platform varchar,
  total integer
);
INSERT INTO bookings_daily (
  date_d, country_name, platform, total
)
SELECT
  ts::date as date_d,
  C.country_name,
  platform,
  count(*) as total
FROM bookings B
LEFT JOIN countries C ON B.country_code = C.country_code
GROUP BY 1
COMMIT;
```

The above code:

- Creates a table named `bookings_daily` (or recreates it if the table already exists)
- Runs an SQL transform query to calculate the aggregation and load the results into the newly created table. In the process, the code also turns country code into proper country name by joining with a countries table.
- All of this is done inside a database transaction, so that if things fail half-way, the prior state is restored.

To deploy the above SQL code to production, we set up a daily cron job that runs the SQL file. This is the most basic method possible, and the contents of our cron job will look like so:

```
$ psql transforms/bookings_daily.sql
```

In the above example, the main transform logic is only within the SELECT statement at the end of the code block. The rest is considered metadata and operational boilerplate.

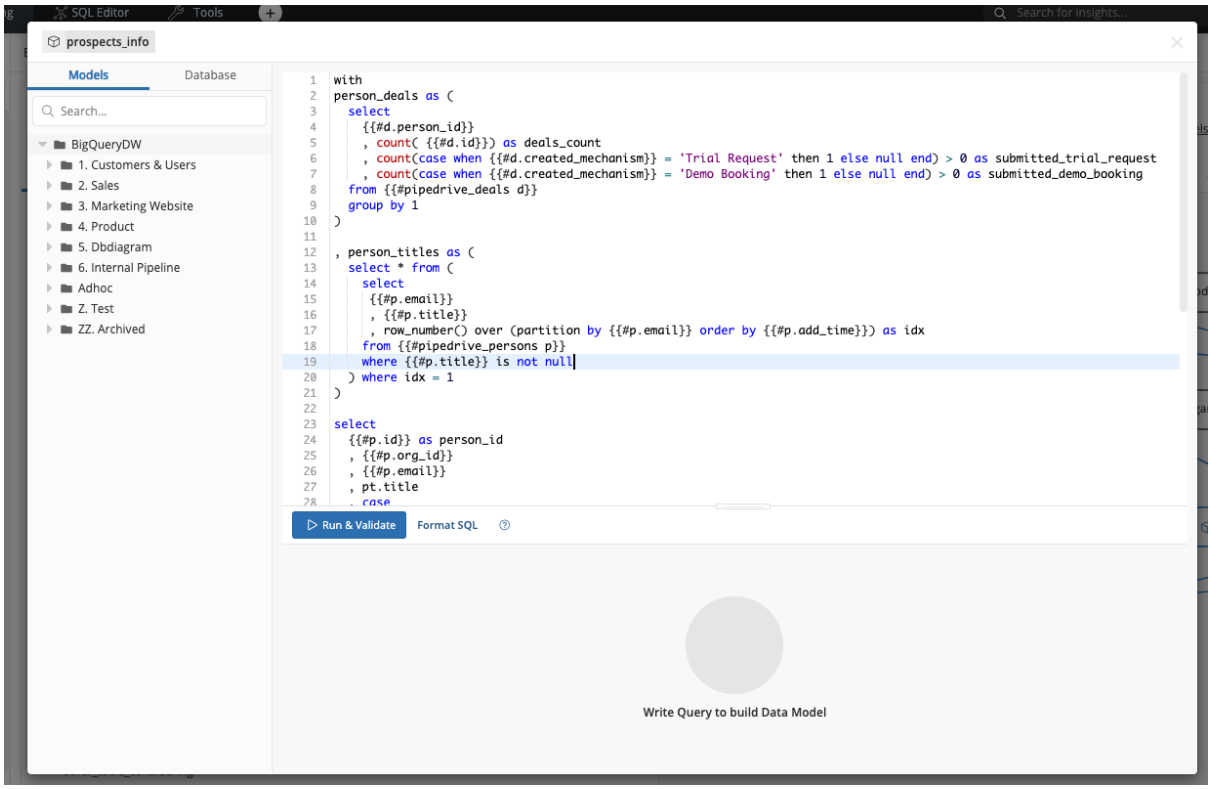
Besides using an SQL table to store the transform results, we may also opt to create a database view (which means we store the definition only), or we can create a materialized view for it.

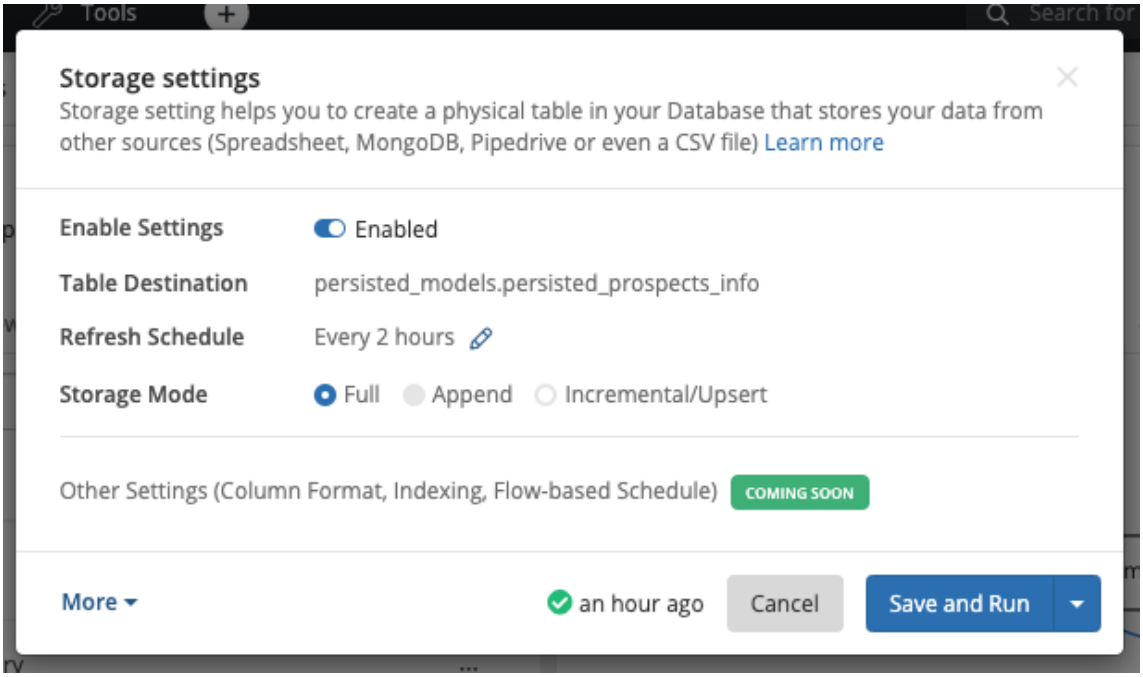
Using data transform tools

In practice, using dedicated transformation tools (like Holistics, dbt, dataform and so on) will handle the SQL boilerplate and let you focus on just the core transformation logic.

For example, the below screenshots show how this is done using Holistics:

- The user focuses on writing SQL to transform data, and the software handles the creation of the model.
- Holistics allows you to choose to save it just as a "view" or as a "materialized view" (persisted to the data warehouse as a new table).



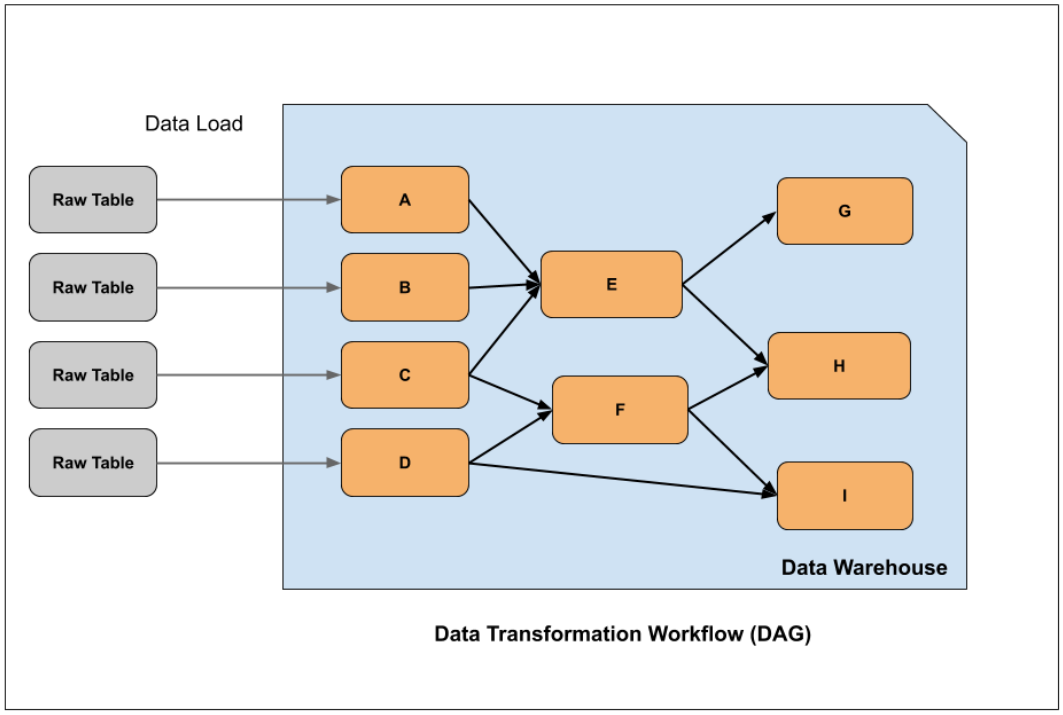


The transformation workflow (or the DAG)

The above section talks about a single transform step. But when you have multiple transforms that depend on each other, you will run into a problem of "dependency management".

For example, let's say that you have two transform jobs: one to calculate sales revenue, and the other to calculate sales commissions based on revenue. You will want the commissions job to be run only after the revenue calculation is done.

This concept is called a DAG (directed acyclic graph) workflow. It is best explained by the diagram below.



Source

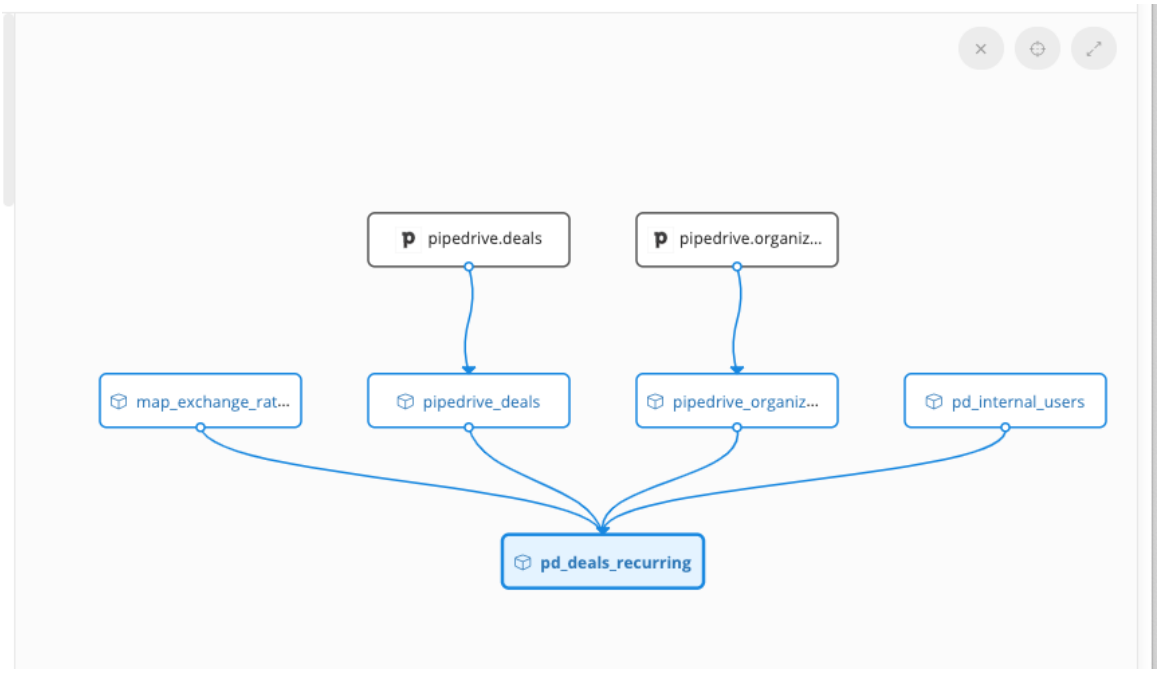
In the above diagram:

- Each node inside the data warehouse represents a table, with the left column (A, B, C, D) being tables loaded from source systems into the data warehouse.
- The arrows represent dependency. That means that in order to create table E, we need data from table A, B and C.
- Tables E to I are transformed tables, created by running corresponding transformation jobs.
- You can clearly see that job E should run after job A, B, C have finished, while job I should run only after both D and F finish. This is the dependency property of a DAG workflow in action.

In practice, most data transformation tools will have support for DAG workflows. This is especially true for classical ETL tools, in the older

paradigm. Regardless of which paradigm you're in, your job will be to focus on managing each transform's logic and their dependencies.

To continue the earlier example using Holistics, once you define a transform job using SQL, Holistics will automatically read the transformation query and then calculate a dependency workflow for it.



A note on traditional Data Transform using ETL

The examples we have been discussing so far is done in the context of the ELT model, which means the transformation happens directly inside the data warehouse.

We shall talk a little about the drawbacks of the more classical approach now. As we've mentioned before, data transformation used to take place in a programming or scripting language, *before* the data

is loaded into the data warehouse. Below is an example of a transform step done using Ruby programming language.

```
# Extract: users is an array of users loaded from the Extract phase
users = load_users_from_crm()

# Transform: select only active users
filtered_users = users.select { |u| u['active'] == true }

# Load: load into data warehouse table
write_to_table(filtered_users, 'reporting.active_users')
```

The main drawback of this approach is that the majority of the load is now on the single computer that runs the script (which has to process millions of data records).

This worked well when data warehouses were slow and expensive. It also worked well at a time when data volumes were comparatively low. Given these restrictions, data professionals would look for ways to offload all processing outside of the data warehouse, so that they may only store cleaned, modeled data in the warehouse to cut down on costs.

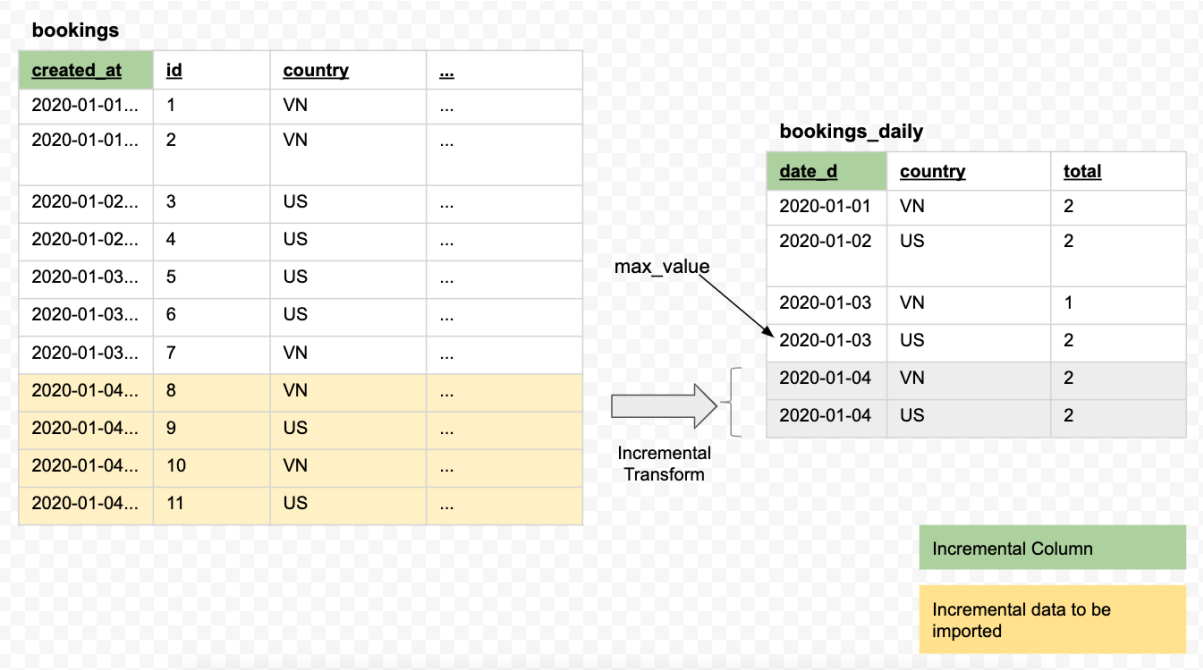
However in recent years, data warehouse costs have gone down significantly, while performance has drastically improved. Today, running transformations in the data warehouse is typically more cost efficient than executing the transformation in code running on a normal machine. Thus we see the trend of moving all processing into the data warehouse itself (ELT).

You may read more about this in the Consolidating data from source systems section of our book.

Advanced Topic: Incremental transform to optimize for performance

In our section on "data loading" earlier, we spoke about Incremental Load, where only the differences in data are loaded from source to destination. A similar concept exists with data transformation.

Let's revisit the earlier example of `bookings → bookings_daily`. This time, let's look at how to run this transformation incrementally.



Source

In the above diagram, you can see that when the transform job runs, only data for `2020-01-04` will be processed, and only two new records will be appended to the `bookings_daily` table.

How much cost does this save us? Quite a bit, as it turns out.

Imagine that your bookings have 100M records and are growing at a pace of 10,000 records a day:

- With incremental transformation: you only process **10,000 records** a day
- Without incremental transformation: you process **100M (and growing) records** a day.

Incremental Transform in Practice

In practice, an incremental transform in SQL should look something like this:

```
destination: bookings_daily
incremental:
  enabled: true
  column: date_d
---

SELECT
  ts::date as date_d,
  C.country_name,
  platform,
  count(*) as total
FROM bookings B
LEFT JOIN countries C ON B.country_code = C.country_code
WHERE [[ ts::date > {{max_value}} ]] --this is added to the code.
GROUP BY 1
```

The `[[ ts::date > {{max_value}} ]]` is added so that the tool will pull the latest value of the incremental column from the destination table and substitute it within the SQL query. With this, only newer data are materialized into a destination table.

When can you *not* run incremental transform?

If you look at the above example, it is clear that sometimes you cannot run incremental transform:

- When your old transformed data keeps changing, and would need to be reloaded
- A quick observation with incremental transforms is that it usually only works if the transform/load step prior to that is also 'incremental-ble' (i.e you may transform the `bookings` table in an incremental manner), though this might not necessarily be true all the time.

Chapter 3:
Data Modeling for Analytics

Data Modeling Layer & Concepts

A contemporary look at data modeling

In this section we're going to introduce data modeling from scratch. We shall approach this in a contemporary manner, which means that our presentation here is going to seem rather unusual to you if you've had prior experience with more classical techniques.

More specifically: if you have lots of experience with Kimball, Inmon or Data Vault-style data modeling, skim this section to familiarise yourself with the terms we introduce. We will tie the concepts back to the more classical approaches once we get to the next two sections.

If you *don't* have any experience with data modeling, then buckle in. We're going to take you on a ride. Let's get started.

What is data modeling and why is it needed?

To best understand what data modeling is, let's imagine that your company runs a homestay booking site, and that it has data stored in a production database.

When the CEO has a question to ask about data, she goes to the data analyst and asks: "Hey, Daniel can you help me get the sales commissions numbers for bookings in this region?"

Daniel listens to the CEO's request, goes to his computer, and comes back with the data, sometimes in the form of a short written note,

other times with the data presented in an Excel file.

So here's our question: why can't the CEO do it themselves? Wouldn't the ideal situation be that the CEO opens up some exploration tool that's linked directly to the production database and helps herself to the data? Why does she have to go through the data analyst to get it for her?

"She doesn't know how to do it", you might say, or "This is serious technical stuff". These are common responses that you might get when you pose this question.

But there's something else that's going on here.

The CEO (or any other business user, for that matter) thinks in business terms, using business logic. Your actual data, on the other hand, is stored in a different format. It follows different rules — often rules imposed by the implementation of the application.

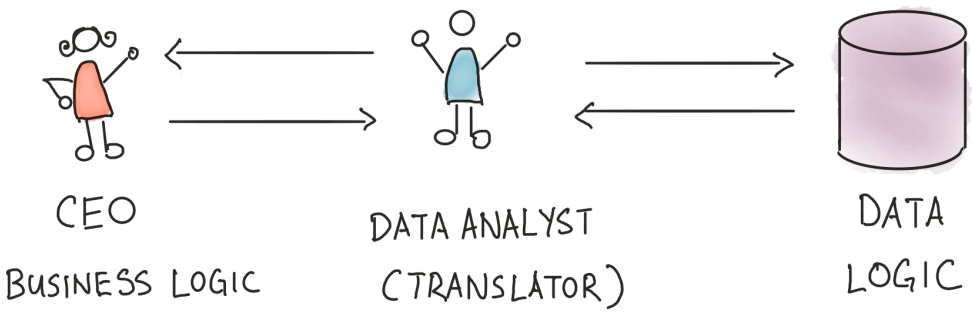
The CEO can't translate her mental model of the business into code in order to run the numbers. She doesn't know how the data is organized. But Daniel does.

For example: when asking about sales commissions, the CEO will think *"sales commissions is 5% of closed deals"*. However, the data analyst will think *"closed deals are stored in table* `closed_deals`*, so I need to take the* `amount` *column and multiply that with* `months` *to figure out the final amount; oh, and I need to check the* `payment_received` *column to make sure that only the received payment is counted"*.

Here, Daniel the data analyst simply serves as a **"data translator"**:

- He receives a data question in business English.
- He figures out where the corresponding data lies in his data warehouse.
- **He then translates the business question into corresponding data logic**, and expresses this logic in the form of a data query (usually SQL).
- He runs the query, get the results to Excel, formats it, and then sends it over to the CEO.

Essentially, the data analyst knows **the mapping between business logic to data logic**. That's why he is able to help the CEO with her data questions.



This process works fine for some companies. But it will not scale up beyond a few people, and is an incredibly inefficient way to do things. Why? Well:

- Your data analyst Daniel is now a bottleneck. Every small change needs to go through him.
- What if Daniel goes on leave? What if Daniel leaves the company? What if Daniel forgets how a certain piece of business logic is implemented?
- Every time the CEO wants something, she needs to wait hours (or even days) for Daniel to crunch the numbers and get back to her.
- At one point, Daniel might be too busy crunching out numbers for different business stakeholders, instead of focusing his time on more valuable, long-term impact work.

So what do we need to do here?

We need to offload the mapping knowledge inside Daniel's head into some system, so that anyone can understand it. We need to externalize it, so that it doesn't just live in Daniel's head.

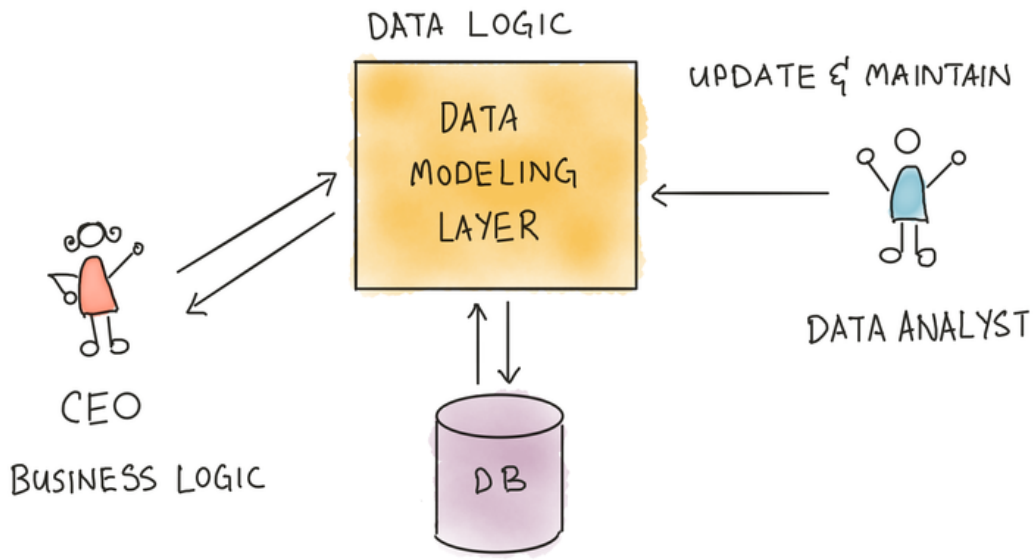
The Data Modeling Layer

Earlier in this book, we introduced you to the idea that we should extract data into a data warehouse first, before doing transformations. We mentioned that this is commonly known as the 'ELT' paradigm.

What we want to do now is to perform some series of transformations to offload the mapping in Daniel's head to something that is persisted in a data warehouse. Again, all of these transformations are to be performed *within* the data warehouse, as per the ELT paradigm.

This process of mapping raw data to a format that can be easily understood by business users is known as 'data modeling'. There are other reasons to do data modeling, of course. Performance is one of them, as is explorability. But at its most basic level, data modeling is about taking raw data and transforming it into a form that is useful for business measurement.

The contemporary approach to doing data modeling is to **orchestrate transformations within the data warehouse, via a tool that sits on top of the data warehouse**. This stands in contrast to ETL tools in the past, which usually exist as pipelines external to the data warehouse.



These tools include such tools like Holistics, dbt, dataform and Looker. These tools share a couple of similar characteristics:

- They connect to your data warehouse.
- They treat the modeling process as the act of transforming data from old tables to new ones within the data warehouse.

- They generate SQL behind the scenes to execute such transformations.
- They allow users to annotate, manage, and track changes to data models over time.
- They allow users to trace the lineage of data transformations within a single tool.

There isn't a good name for such tools right now. For the sake of convenience, we will call them 'data modeling layer' tools. Conceptually, they present a 'data modeling layer' to the analytics department.

> A data modeling layer is a system that contains the mapping between business logic and underlying data storage rules of your business. It exists primarily in the ELT paradigm, where data is loaded into the data warehouse first before being transformed.
>
> In this context, data modeling is the process of building and maintaining this layer.

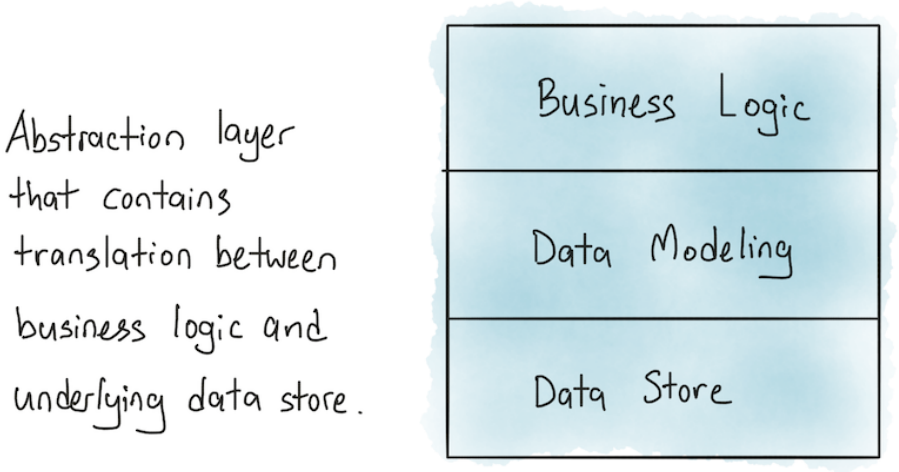
Usually, the data modeling layer will later be connected to some visualization tool or business intelligence layer. Non-technical users should be able to log in, interact with some user interface, and get the analytics they need, *without* the requirement to talk to anyone technical.

With a proper, well-maintained data modeling layer, everyone is happy:

- The CEO can just log in to the BI application, ask questions and get the right numbers that she needs, without waiting for the data

analyst. In other words, **business users can now do self-service analytics.**

- The data analyst's job is now focused on maintaining the data pipeline and modeling layer, without being bombarded by adhoc data requests.
- The entire company has a well-documented layer of data knowledge. Even if the data analyst is busy or leaves the company, this knowledge is properly annotated and organized, and not at risk of being lost.

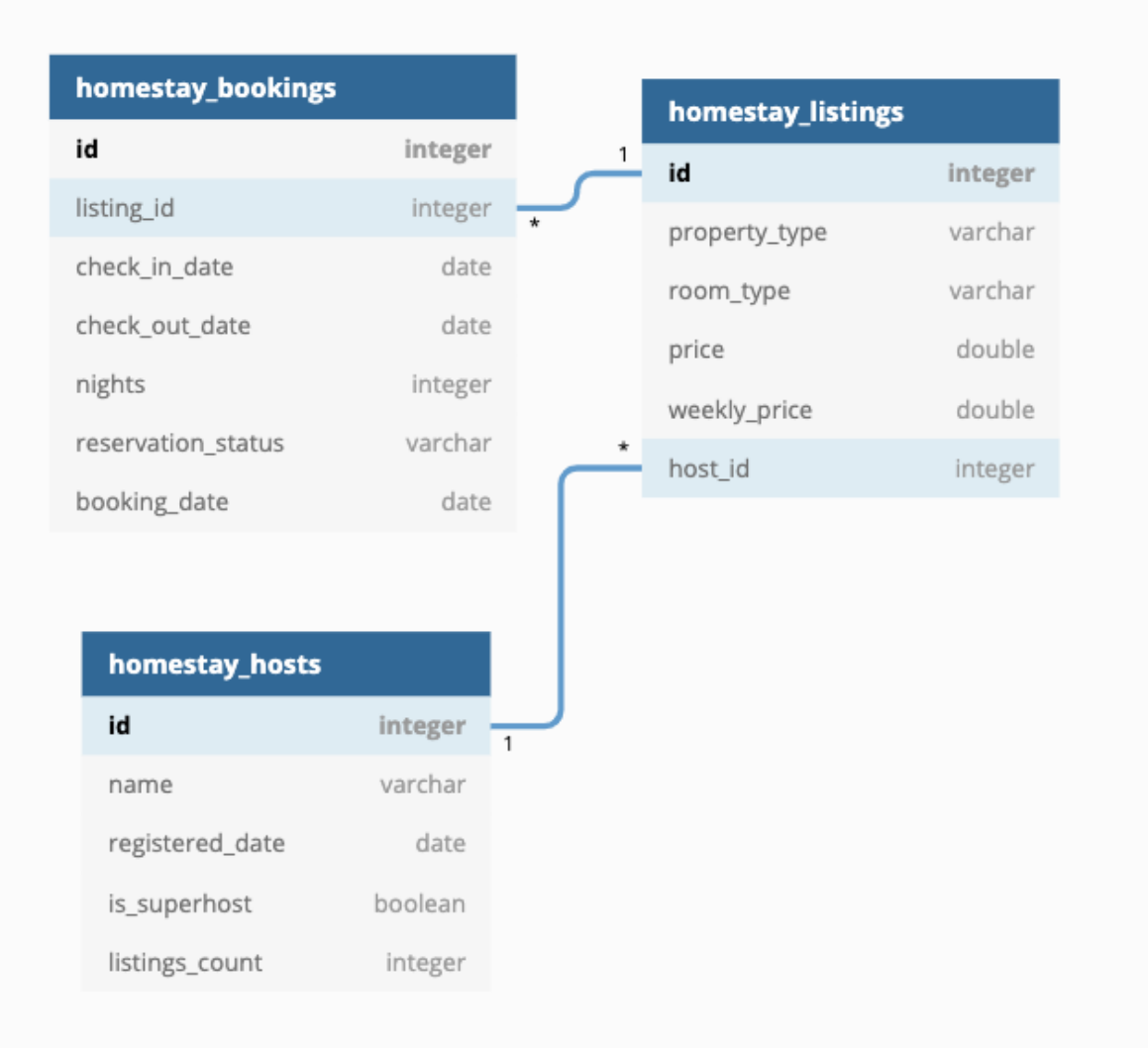


Now that we know what data modeling is at a high level and why it's important, let's talk about specific concepts that exist in the data modeling layer paradigm.

Data Modeling Layer Concepts

Let's return to the homestay booking example above. We shall use this as an overarching example to introduce data modeling layer concepts to you.

These are the basic database tables that you pulled into your data warehouse.

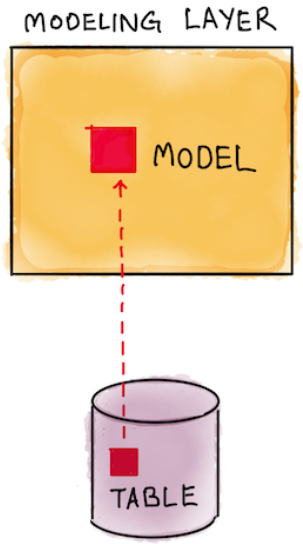


Now, let's look at a few modeling operations that you can apply to the above data.

We'll be using Holistics for the examples below. That said, these concepts map pretty well across any data modeling layer-type tool in the market. (We'll tell you if we're introducing an idea that is specific to Holistics, with no clear external analogs).

Concept: Data Model

When manipulating data in a data modeling layer, it's common *not* to deal with the underlying data table directly, but to instead create an abstract object above it for ease of manipulation.



This is called a data model.

A data model is an abstract view on top of a physical database table that you may manipulate without directly affecting the underlying data. Most data modeling layers allow you to store additional metadata that may enrich the underlying data in the data table.

The most common types of metadata at the level of a data model are textual descriptions, calculated dimensions that capture some business logic, and relationship mappings between a model and some other model (or table). Data modeling layers often also include housekeeping metadata alongside the data model, such as a full history of user accounts who have modified the model, when that model was first created and last modified, when the underlying data was last refreshed, and so on.

> *While database tables hold data, data models often contain metadata to provide extra context for that data.*

A data table is managed by the database, and a data model is managed by the data modeling layer that you use.

Here's a quick comparison between the two:

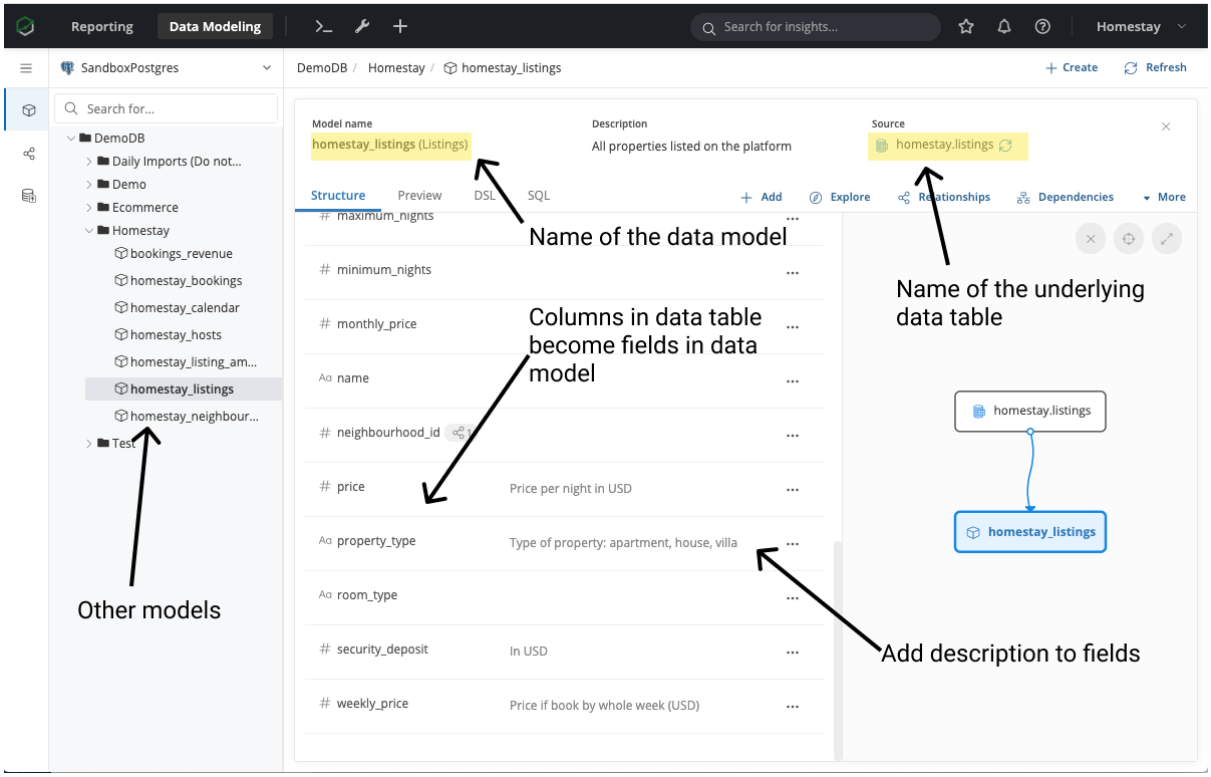
Data Table:

- Is 'physical', lives in the analytical database
- Store actual data records

Data Model:

- Is an abstract object, managed by the data modeling layer
- Stores metadata (description, business logic, relationship mapping)
- Usually sits above a data table
- Is usually created via a `SELECT` statement from the data table.

By connecting a modeling layer to your analytics database, you can "model up" your database tables so that you can add textual descriptions, or metadata. This is what it looks like in Holistics:



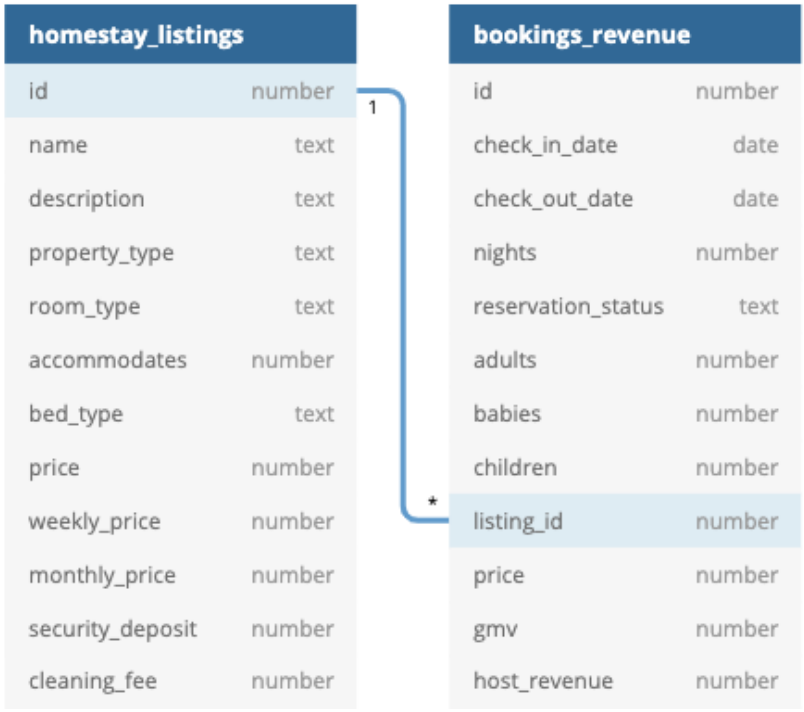
In the Holistics interface above, we take a table in our data warehouse named `homestay.listings` and create a data model named `homestay_listings`. This model is 'abstract', in the sense that it lives only within the data modeling layer.

Similar flows exist in dbt, dataform and Looker.

Concept: Relationship mapping

A relationship mapping is a foreign relationship between two data models.

This relationship is analogous to the foreign key concept in relational databases. In this context, however, we are creating relationships between two data models instead of tables — that is, relationships between two abstract 'views' of data that don't actually exist in the data warehouse below.



Defining a model's relationships is like defining JOINs between the data tables.

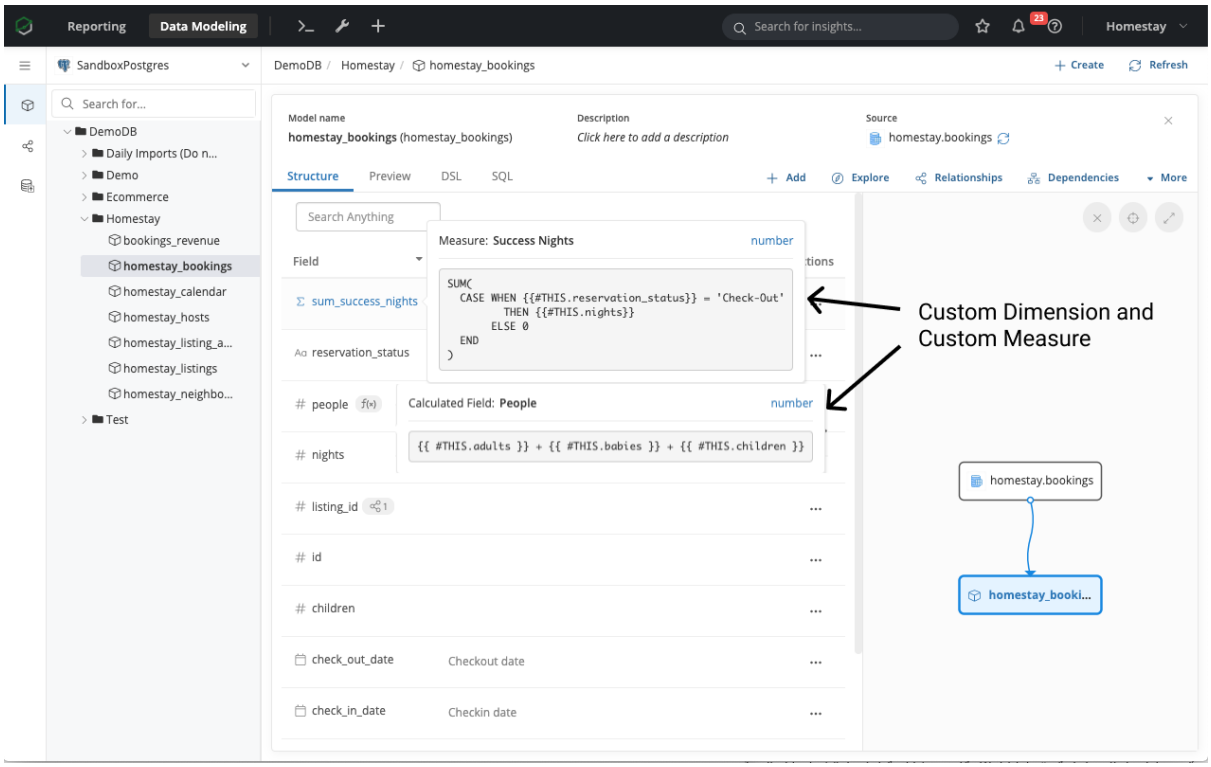
This is useful because you may want to create models that derive from other models, instead of deriving from just underlying database tables.

Concept: Custom Field Logic

Remember earlier that we had a few pieces of business logic that the CEO might want to check? For instance, one metric the CEO might be particularly interested in is the number of guests per booking. This way, she can have a good idea of the inventory matchup between

available rooms and groups of guests. Notice how a measure of the number of guests might not exist in the underlying table. Instead, we're going to define it as a *combination* of table dimensions, but within the data modeling layer.

In Holistics, we open up the `homestay_bookings` model and define our custom dimensions and measures, like so:



We have defined two custom fields here. The first is the sum of nights stayed from a successful booking (that is, a booking that has been seen to check out). If the booking has reached the 'checked out' state, then the nights are counted. Otherwise, it returns 0.

The second is a calculated field that returns the total number of guests per booking. This latter field is a simple sum, because the `homestay.bookings` table stores the number of children, babies, and

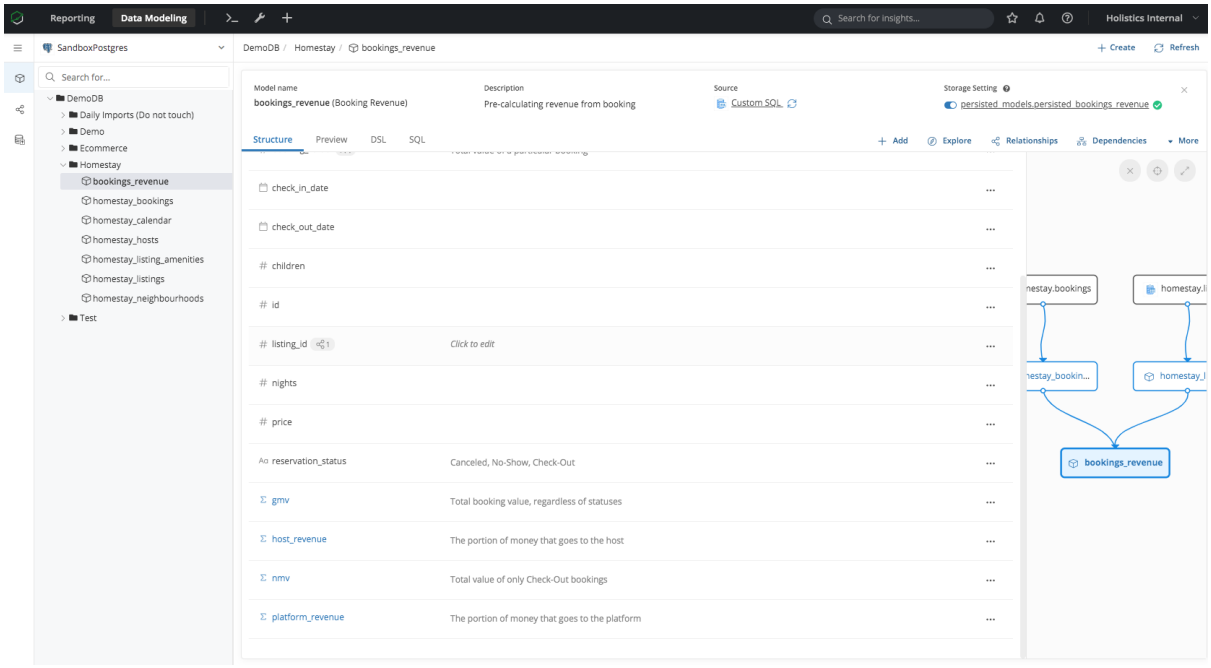
adults as separate numbers. In our `homestay_bookings` model, we simply define the total number of guests as the sum of all three numbers.

We call these measures 'custom fields' in Holistics; though the idea is more important than the specific name we've used. The idea is this: data modeling layers allow you to create calculated fields that are combinations of other fields within the model. Similar concepts exist in other modeling BI tools — the point is that you want to be able to augment existing dimensions in the underlying tables.

Concept: Models Built On Top of Other Models

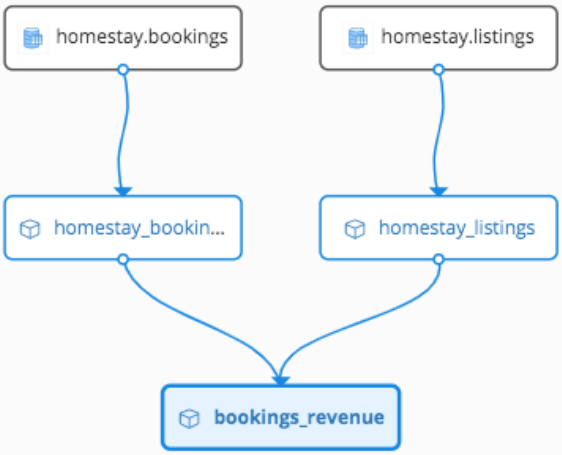
At this stage, we've merely demonstrated that data models allow you to annotate and enrich existing tables in your data warehouse. But what if you want to *transform* your data into a new set of tables? Imagine, for instance, that you want to take the data within `homestay.listings` and turn it into a star schema. In the past, you might have done this by asking a data engineer to set up a new ETL pipeline. With a data modeling layer tool, however, you may do this within the same user interface.

Below, we've created a new model that is derived from the `homestay_listings` data model (not the table!). This model is called `bookings_revenue`, and it combines fields from two different models — the `homestay_listings` model and the `homestay_bookings` models — to calculate things like `gmv` and `host_revenue`.

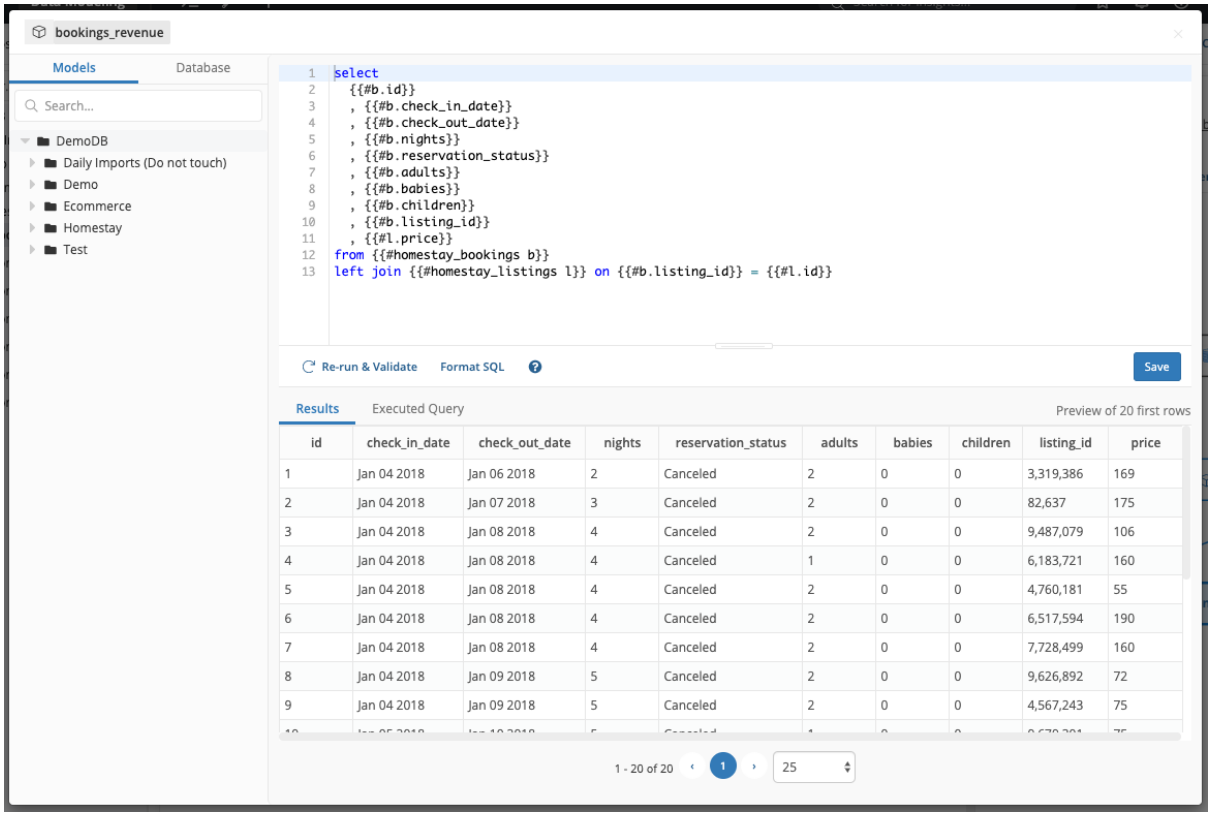


Notice the conceptual leap that's just happened. `homestay.bookings` and `homestay.listings` are two tables in our data warehouse. We've created two models respectively above them: `homestay_bookings` and `homestay_listings`. As we've mentioned previously, these models allow us to annotate or create new derived fields *without* touching the tables themselves.

We have then taken the two *models*, and created a new *transformed* model on top of them.



This is powerful for two reasons. First, all of this has happened via SQL. We do not need to wait for data engineering to set up a new transformation pipeline; we can simply ask an analyst to model the data within the Holistics data modeling layer, by writing the following SQL:



As you can see, our new `bookings_revenue` model is simply a `SELECT` statement that `JOINS` two **models**. (Emphasis added).

Second, the fact that our model exists as a simple join means that our `bookings_revenue` model will be updated whenever the two underlying models (or their underlying tables) are updated!

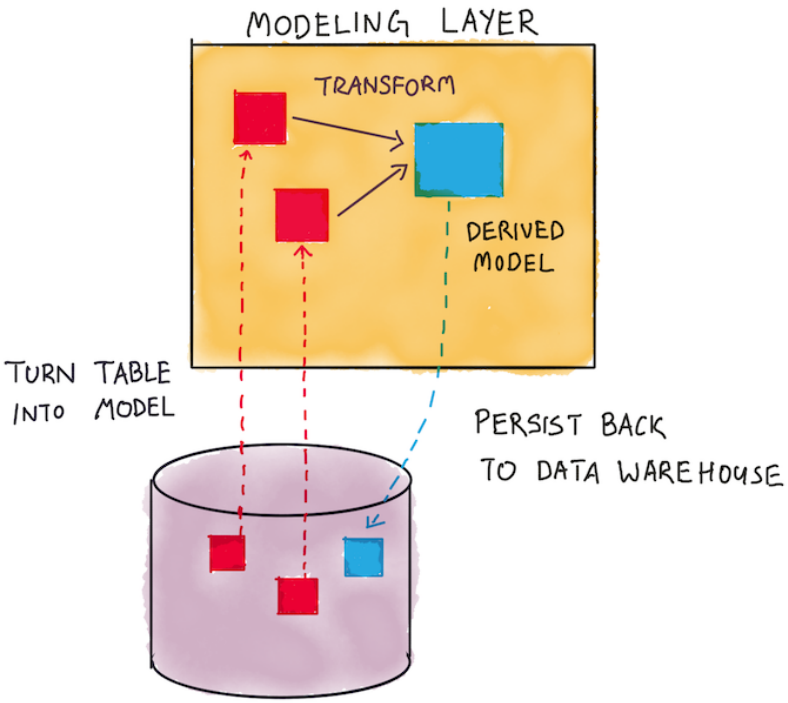
Again, the *idea* here is more important than the particular implementation we've shown you. You will find similar flows in other data modeling layer tools (like Dataform, for instance).

Concept: Model Persistence (equivalent to materialized view)

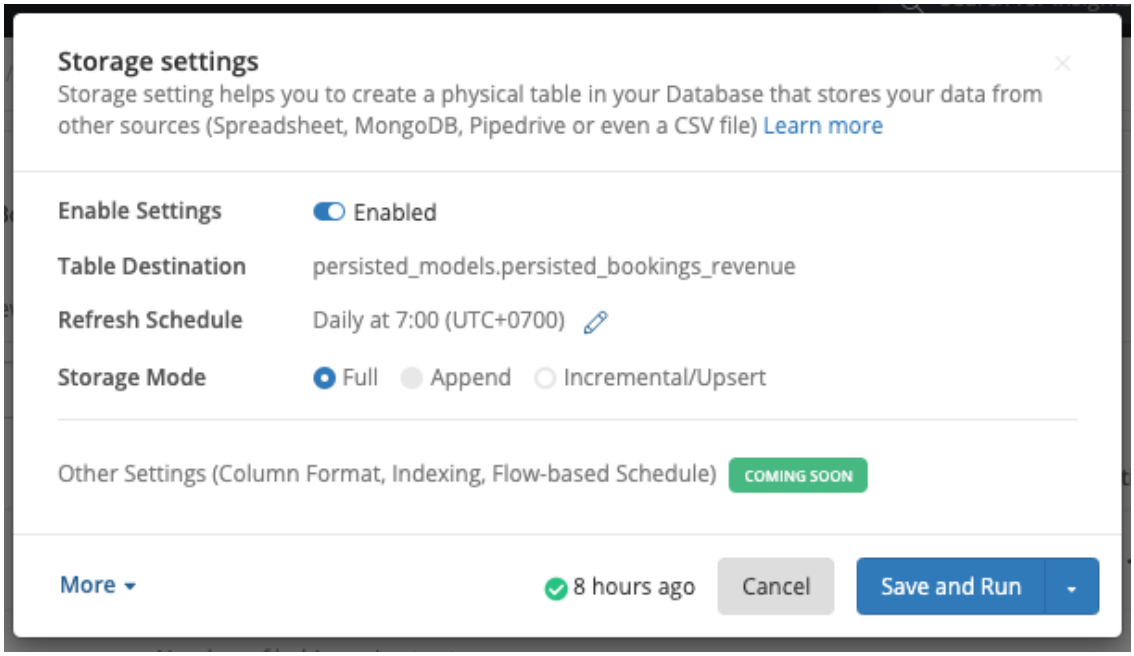
Once you understand that it's possible to create new models from other models by writing SQL, the next step is to ask: is it possible to

persist such transformations?

The answer is yes, of course!



While data models are normally 'views' (or `SELECT` statements) on top of physical tables, nearly all data modeling layers allow you to persist these as new tables within the data warehouse. This is what the Holistics persistence menu looks like, for instance:

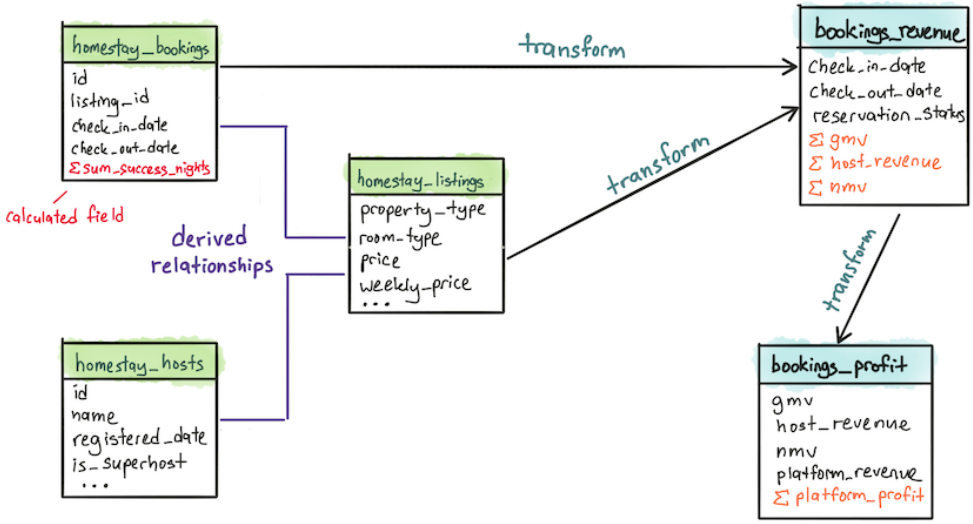


Here, Holistics offers to update these models at periodic times. In our example above, we can choose to persist `bookings_revenue` to our data warehouse, and let Holistics update the model every day at 7am (or whatever other time intervals we wish).

Persistence is useful for performance reasons; in this case, our revenue reporting may run on top of the `bookings_revenue` persisted table, instead of the abstract model itself.

Putting things together

Let's put the above concepts together in a single diagram, as a way to wrap things up.



Notice three things:

- `homestay_listings` is a base model that is derived from `homestay_bookings` and `homestay_hosts`.
- `bookings_revenue` is a 'transformed' model that is drawn from `homestay_bookings` and `homestay_listings`. Note how it contains custom fields that are basically calculations of multiple fields that exist within its underlying models.
- Similarly, `bookings_profit` is a transformed model that is taken from `bookings_revenue`.

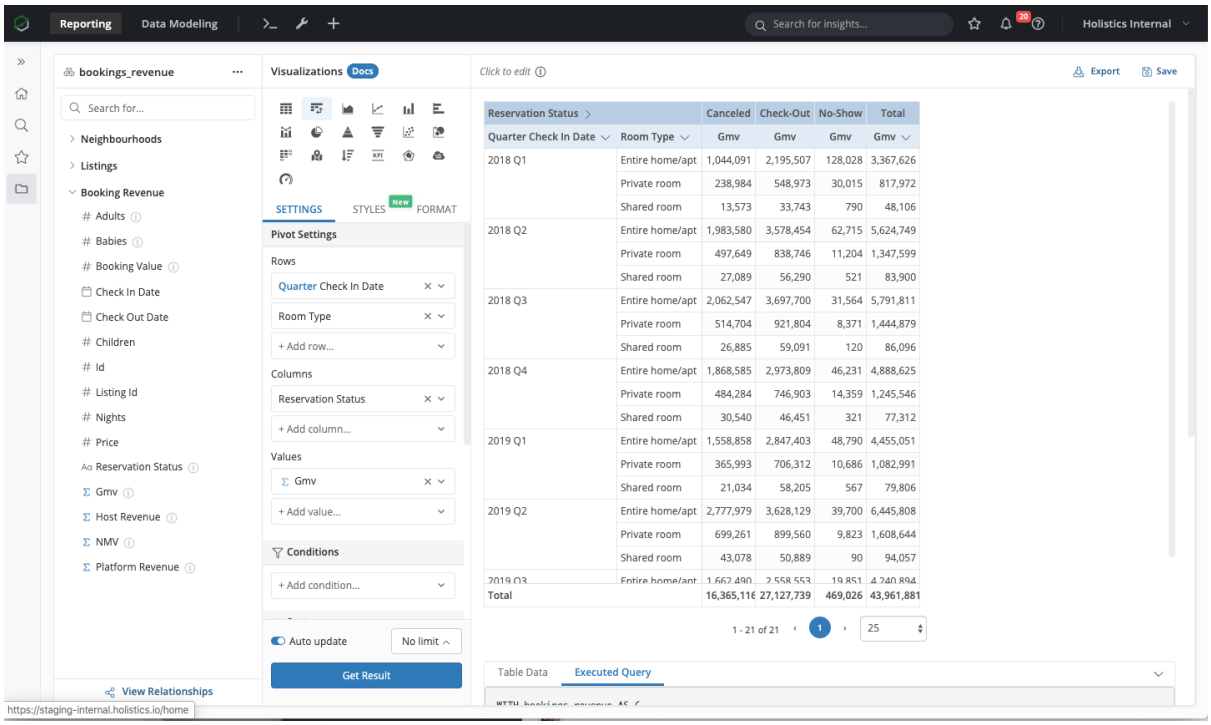
This dependency diagram of models is pretty representative of the sort of work analysts at Holistics do on a day-to-day basis. The job of data analytics is essentially a process of modeling raw tables into base models, and then modeling base models into transformed models.

Our analysts do this until they have a tapestry of data models that represent everything our business people would ever want to know about our company. Then, creating a report simply becomes an issue of picking the right models to present to the business.

What do business users get in the end?

Remember at the beginning of this section, we said that with a data modeling layer, the CEO can extract meaningful numbers by herself without having to bother the data analyst? Well, we weren't lying.

With a SQL-based data modeling BI tool (like Holistics or Looker), the CEO can now use a UI to help herself to the data she needs, based on the metrics, descriptions, and relationships we have defined above.



Example of self-service data exploration UI that CEO can play around.

What happens behind the scenes is this: the CEO's selection will be translated into a corresponding SQL query (thanks to the modeling

layer), and this query will be sent to the analytical database. The retrieved results will then be displayed for the CEO to view.

Table Data | Executed Query

```
WITH bookings_revenue AS (
  SELECT
    T0."check_in_date" AS "check_in_date",
    T0."nights" AS "nights",
    T0."reservation_status" AS "reservation_status",
    T0."listing_id" AS "listing_id",
    T0."price" AS "price"
  FROM
    "persisted_models"."persisted_bookings_revenue" T0
), homestay_listings AS (
  SELECT
    T0."id" AS "id",
    T0."room_type" AS "room_type"
  FROM
    "homestay"."listings" T0
), holistics__explore_fb4d4c AS (
  SELECT DISTINCT
    DATE_TRUNC ( 'QUARTER', T0."check_in_date" ) AS "datetrunc_quarter_bookings_revenue_check_in_date",
    T1."room_type" AS "homestay_listings_room_type",
    T0."reservation_status" AS "bookings_revenue_reservation_status",
    sum( ( T0."price" * T0."nights" )) AS "custom_bookings_revenue_gmv"
  FROM
    bookings_revenue T0
    LEFT JOIN homestay_listings T1 ON T0."listing_id" = T1."id"
  GROUP BY 1, 2, 3
  ORDER BY 4 DESC
)
SELECT
  T0."datetrunc_quarter_bookings_revenue_check_in_date" AS "datetrunc_quarter_bookings_revenue_check_in_date",
  T0."homestay_listings_room_type" AS "homestay_listings_room_type",
  T0."bookings_revenue_reservation_status" AS "bookings_revenue_reservation_status",
  T0."custom_bookings_revenue_gmv" AS "custom_bookings_revenue_gmv"
FROM
  holistics__explore_fb4d4c T0
LIMIT 100000
```

Summary

So what have we covered? We've looked at data modeling in the ELT paradigm. The modern approach to data modeling in this paradigm is to use what we call a 'data modeling layer', though this is a name that we've adopted out of convenience. Such tools include Dataform, dbt, Looker, and Holistics itself.

We then discussed several ideas that exist within this approach to data modeling:

- We talked about how data models are 'abstract views' on top of your data, within the context of a 'data modeling layer'.

- We talked about how such data modeling layer-type tools allowed users to enrich models with metadata, and how most modeling layers added useful housekeeping data like user modifications, freshness of the underlying data, and so on.
- We discussed two useful features of data models: custom field logic and relationship mapping.
- We then talked about how data models may be built on top of other models.
- Finally, we talked about how such derived models may be persisted, just like materialized views in a more traditional relational database system.

In our next section, we shall talk about how this approach to data modeling works when it is combined with the classical method of dimensional data modeling.

Kimball's Dimensional Data Modeling

This section covers the ideas of Ralph Kimball and his peers, who developed them in the 90s, published *The Data Warehouse Toolkit* in 1996, and through it introduced the world to dimensional data modeling.

In this section, we will present a broad-based overview of dimensional data modeling, explore why the approach has become so dominant, and then examine what bits of it we think should be brought into the modern cloud data warehousing era.

Why Kimball?

There are many approaches to data modeling. We have chosen to focus on Kimball's because we think his ideas are the most widespread, and therefore the most resonant amongst data professionals. If you hire a data analyst today, it is likely that they will be familiar with the ideas of dimensional data modeling. So you will need to have a handle on the approach to work effectively with them.

But we *should* note that there is another approach to data modeling that is commonly mentioned in the same breath. This approach is known as Inmon data modeling, named after data warehouse pioneer Bill Inmon. Inmon's approach was published in 1990, six years before Kimball's. It focused on normalized schemas, instead of Kimball's more denormalized approach.

A third data modeling approach, named Data Vault, was released in the early 2000s.

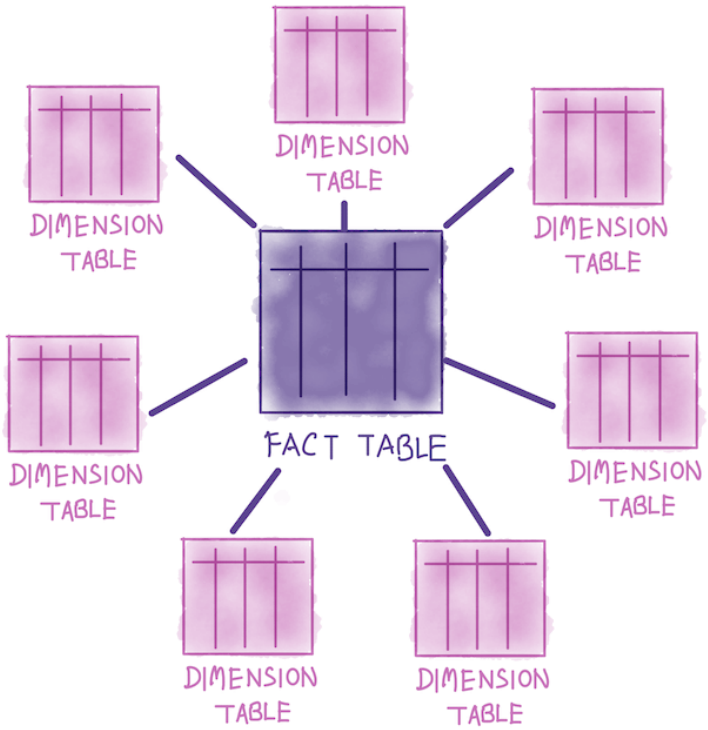
We think that many of these approaches are valuable, but that all of them are in need of updates given the rapid progress in data warehousing technology.

The Star Schema

To understand Kimball's approach to data modeling, we should begin by talking about the star schema. The star schema is a particular way of organizing data for analytical purposes. It consists of two types of tables:

- A fact table, which acts as the primary table for the schema. A fact table contains the primary measurements, metrics, or 'facts' of a business process.
- Many dimension tables associated with the fact table. Each dimension table contains 'dimensions' — that is, descriptive attributes of the fact table.

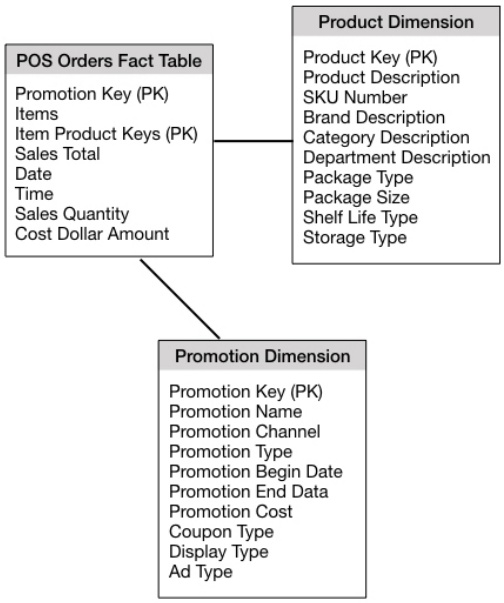
These dimensional tables are said to 'surround' the fact table, which is where the name 'star schema' comes from.



This is all a little abstract, so let's go through an example to make this concrete.

Let's say that you're running a store, and you want to model the data from your Point of Sales system. A naive approach to this is to use your order transaction data as your fact table. You then place several dimension tables around your order table — most notably products and promotions. These three tables are linked by foreign keys — that is, each order may reference several products or promotions stored in their respective tables.

This basic star schema would thus look something like this:



Notice how our fact table will grow very quickly over time, as we may see hundreds of orders per day. By way of comparison, our products table and promotions table would contain far fewer entries, and would be updated at a frequency much lower than the fact table.

Kimball's Four Step Process

The star schema is useful because it gives us a standardized, time-tested way to think about shaping your data for analytical purposes.

The star schema is:

- **Flexible** — it allows your data to be easily sliced and diced any which way your business users want to.
- **Extensible** — you may evolve your star schema in response to business changes.

- **Performant** — Kimball's dimensional modeling approach was developed when the majority of analytical systems were run on relational database management systems (RDBMSes). The star schema is particularly performant on RDBMSes, as most queries end up being executed using the 'star join', which is a Cartesian product of all the dimensional tables.

But the star schema is only useful if it is easily applicable within your company. So how do you come up with a star schema for your particular business?

Kimball's answer to that is the Four Step Process to dimensional data modeling. These four steps are as follows:

1. **Pick a business process to model.** Kimball's approach begins with a business process, since ultimately, business users would want to ask questions about processes. This stands in contrast to earlier modeling methodologies, like Bill Inmon's, which started with the business entities in mind (e.g. the customer model, product model, etc).

2. **Decide on the grain**. The grain here means the level of data to store as the primary fact table. It should be the most *atomic* level possible — that is, a level of data that cannot be split further. For instance, in our Point of Sales example earlier, the grain should actually be the *line items* inside each order, instead of the order itself. This is because in the future, business users may want to ask questions like "what are the products that sold the best during the day in our stores?" — and you would need to drop down to the line-item level in order to query for that question effectively. In Kimball's day, if you had modeled your data at the order level, such a question would take a huge amount of work to get at the data, because you would run the query on slow database systems. You

might even need to do ETL again, if the data is not currently in a queryable form in your warehouse! So it is best to model at the lowest level possible from the beginning.

3. **Chose the dimensions that apply to each fact table row.** This is usually quite easy to answer if you have 'picked the grain' properly. Dimensions fall out of the question "how do business people describe the data that results from the business process?" You will decorate fact tables with a robust set of dimensions representing all possible descriptions.

4. **Identify the numeric facts that will populate each fact table row.** The numeric data in the fact table falls out of the question "what are we answering?" Business people will ask certain obvious business questions (e.g. what's the average profit margin per product category?), and so you will need to decide on what are the most important numeric measures to store at the fact table layer, in order to be recombined later to answer their queries. Facts should be true to the grain defined in step 2; if a fact belongs to a different grain, it should live in a separate fact table.

In the case of a retail POS, if we go through the four steps, above, we would model line items, and would end up with something like this:

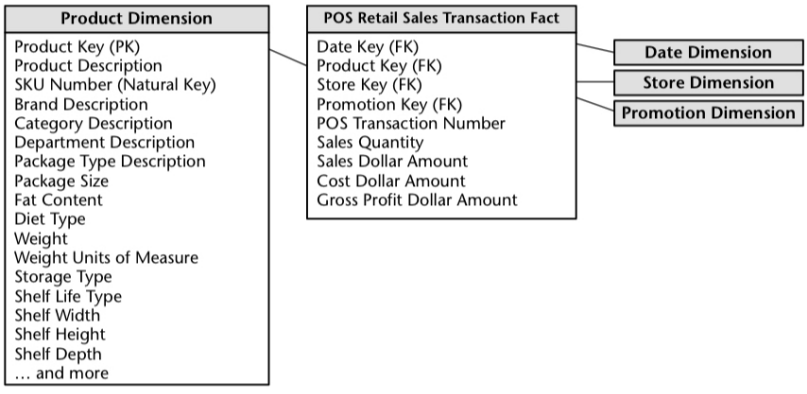


Figure 2.7 Product dimension in the retail sales schema.

Notice how the dimension tables are oriented out from around the fact table. Note also how fact tables consist of foreign keys to the dimensional tables, and also how 'numeric facts' — fields that can be aggregated for business metric purposes — are carefully chosen at the line item fact table.

Now notice that we have a *date dimension* as well:

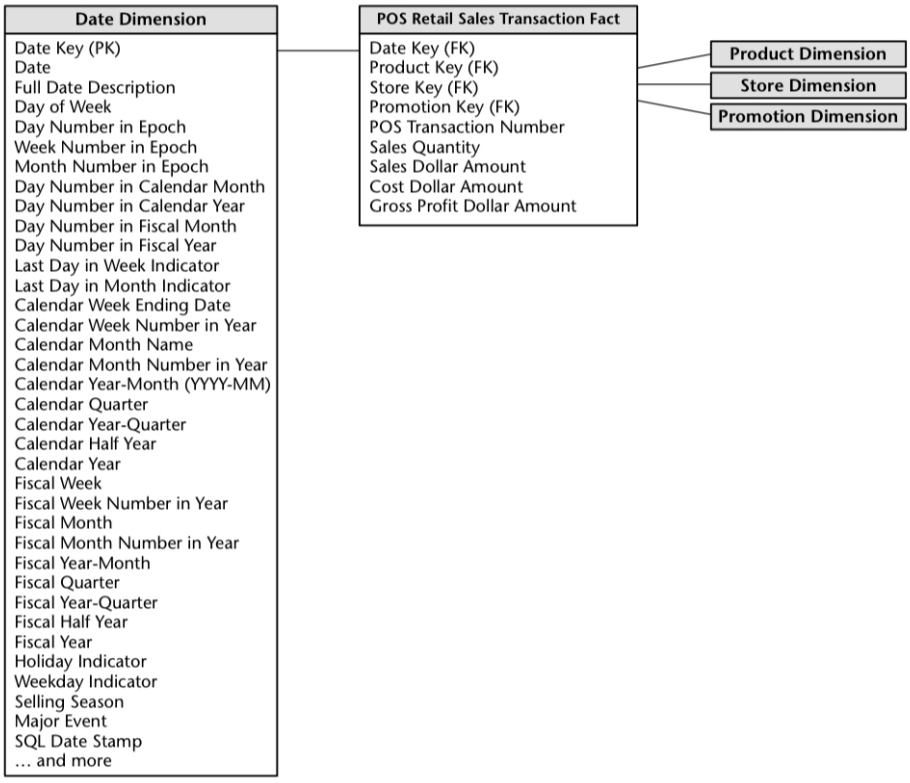


Figure 2.4 Date dimension in the retail sales schema.

This might be surprising to you. Why would you have something like a date dimension, of all things? The answer is to make things easier to query for the business user. Business users might like to query in terms of fiscal year, special holidays, or selling seasons like Thanksgiving and Christmas. Since these concepts aren't captured in the date field of an RDBMS system, we need to model date as an explicit dimension.

This captures a core philosophy of Kimball's approach, which is to **do the hard work now, to make it easy to query later**.

This short example gives you all the flavor of dimensional data modeling. We can see that:

1. The fact and dimension tables give us a standardized way to think about shaping your analytical data. This makes your work as a data analyst a lot easier, since you are guided by a certain structure.
2. Kimball's four steps can be applied to *any* business process (and he proves this, because *every chapter in* The Data Warehouse Toolkit *covers a different business process!*)
3. The star schema that falls out of this results in flexibility, extensibility, and performance.
4. The star schema works well given the performance constraints that Kimball worked with. Remember that memory was relatively expensive during Kimball's time, and that analytical queries were either run on top of RDBMSes, or exported into OLAP cubes. Both approaches benefited from a well-structured dimensional data model.

Why Was Kimball's Approach Needed?

Before we discuss if these techniques are applicable today, we must ask: why were these data modeling techniques introduced in the first place? Answering this question helps us because we may now evaluate if the underlying reasons have changed.

The dimensional data modeling approach gained traction when it was first introduced in the 90s because:

1. **It gave us speed to business value.** Back in the day, data warehouse projects were costly affairs, and needed to show business value as quickly as possible. Data warehouse designers before the Kimball era would often come up with normalized schemas. This made query writing very complicated, and made it more difficult for business intelligence teams to deliver value to the business quickly and reliably. Kimball was amongst the first to formally realize that denormalized data worked better for analytical workloads compared to normalized data. His notion of the star schema, alongside the 'four steps' we discussed earlier in this section, turned his approach into a repeatable and easily applicable process.
2. **Performance reasons.** As we've mentioned earlier, in Kimball's time, the majority of analytical workloads were still run on RDBMSes (as Kimball asserts himself, in *The Data Warehouse Toolkit*). Scattered throughout the book are performance considerations you needed to keep in mind, even as they expanded on variations of schema design — chief amongst them is the idea that star schemas allowed RDBMSes to perform highly efficient 'star joins'. In a sentence: dimensional modeling had *very real benefits* when it came to running business analysis — so large, in fact, that you simply couldn't ignore it. Many of these benefits applied even when people were exporting data out from data warehouses to run in more efficient data structures such as OLAP cubes.

We think that Kimball's ideas are so useful and so influential that we would be unwise to ignore them today. But now that we've examined the reasons that it rose in prominence in the first place, we must ask: how relevant are these ideas in an age of cloud-first, incredibly powerful data warehouses?

Kimball-Style Data Modeling, Then And Now

The biggest thing that has changed today is the difference in costs between data labor versus data infrastructure.

Kimball data modeling demanded that you:

- Spent time up front designing the schema.
- Spent time building and maintaining data pipelines to execute such schemas (using ETL tools, for the most part).
- Keep a dedicated team around that is trained in Kimball's methodologies, so that you may evaluate, extend, and modify existing star schemas in response to business process changes.

When data infrastructure was underpowered and expensive, this investment made sense. Today, cloud data warehouses are many times more powerful than old data warehouses, and come at a fraction of the cost.

Perhaps we can make that more concrete. In *The Data Warehouse Toolkit*, Kimball described a typical data warehouse implementation project with the following illustration:

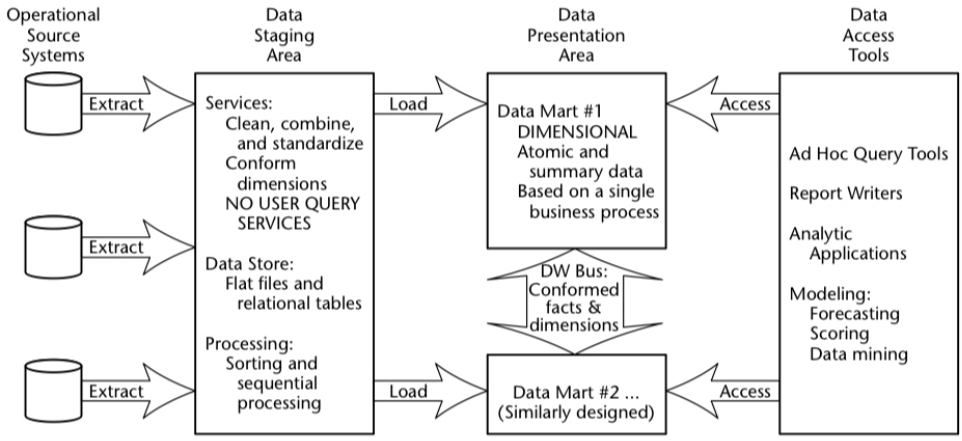


Figure 1.1 Basic elements of the data warehouse.

A typical project would go like this: you would write ETL to consolidate data sources from different source systems, accumulate data into a staging area, then use an ETL tool (again!) to model data into a data presentation area. This data presentation area consists of multiple data marts. In turn, these 'marts' may be implemented on top of RDBMSes, or on top of an OLAP cube, but the point is that the marts must contain dimensionally modeled data, and that data must be *conformed* across the entire data warehouse project.

Finally, those data marts are consumed by data presentation tools.

You will notice that this setup is vastly more complicated than our approach. Why is this the case?

Again, the answer lies in the technology that was available at the time. Databases were slow, computer storage was expensive, and BI tools needed to run on top of OLAP cubes in order to be fast. This demanded that the data warehouse project be composed of a number of separate data processing steps.

Today, things are much better. Our approach assumes that you can do away with many elements of Kimball's approach.

We shall give two examples of this, before we generalize to a handful of principles that you may apply to your own practice.

Example 1: Inventory Management

In *The Data Warehouse Toolkit*, Ralph Kimball describes how keeping track of inventory movements is a common business activity for many types of businesses. He also notes that a fact table consisting of *every single inventory move is too large to do good analysis on.*

Therefore, he dedicates an entire chapter to discuss various techniques to get around this problem. The main solution Kimball proposes is to use ETL tools to create 'snapshot' fact tables, that are basically aggregated inventory moves for a certain time period. This snapshotting action is meant to occur on a regular basis.

Kimball then demonstrates that data analysis can happen using the aggregated snapshot tables, and only go down to the inventory fact table for a minority of queries. This helps the business user because running such queries on the full inventory table is often a performance nightmare.

Today, modern cloud data warehouses have a number of properties to make this 'snapshotting' less of a hard requirement:

1. Modern cloud data warehouses are usually backed by a columnar data architecture. These columnar data stores are able to chew through *millions* of rows in seconds. The upshot here is that you

can throw out the entire chapter on snapshot techniques and still get relatively good results.

2. Nearly all modern cloud data warehouses run on massively parallel processing (MPP) architectures, meaning that the data warehouse can dynamically spin up or down as many servers as is required to run your query.
3. Finally, cloud data warehouses charge by usage, so you pay a low upfront cost, and only pay for what you use.

These three requirements mean that it is often more expensive to hire, train and retain a data engineering team necessary to maintain such complex snapshotting workflows. It is thus often a better idea to run all such processes directly on inventory data within a modern columnar data warehouse.

(Yes, we can hear you saying "but snapshotting is still a best practice!" — the point here is that it's now an *optional* one, not a hard must.)

Example 2: Slowly Changing Dimensions

What happens if the dimensions in your dimension tables change over time? Say, for instance, that you have a product in the education department:

Product Key	Product Description	Department	SKU Number (Natural Key)
12345	IntelliKidz 1.0	Education	ABC922-Z

And you want to change IntelliKidz 1.0's department to 'Strategy'.

Product Key	Product Description	Department	SKU Number (Natural Key)
12345	IntelliKidz 1.0	Strategy	ABC922-Z

The simplest strategy you may adopt is what Kimball calls a 'Type 1' response: you update the dimension naively. This is what has happened above. The good news is that this response is simple. The bad news is that updating your dimension tables this way will mess up your old reports.

For instance, if management were to run the old revenue reports again, the same queries that were used to calculate revenue attributed to the Education department would now return different results — because IntelliKidz 1.0 is now registered under a different department! So the question becomes: how do you register a change in one or more of your dimensions, while still retaining the report data?

This is known as the 'slowly changing dimension' problem, or 'dealing with SCDs'.

Kimball proposed three solutions:

The first, 'Type 1', is to update the dimension column naively. This approach has problems, as we've just seen.

The second, 'Type 2', is to add a new row to your product table, with a new product key. This looks as follows:

Product Key	Product Description	Department	SKU Number (Natural Key)
12345	IntelliKidz 1.0	Education	ABC922-Z
25984	IntelliKidz 1.0	Strategy	ABC922-Z

With this approach, all new orders in the fact table will refer to the product key 25984, not 12345. This allows old reports to return the same numbers.

The final approach, 'Type 3', is to add a new column to the dimension table to capture the previous department. This setup supports the ability to view an 'alternate reality' of the same data. The setup thus looks like this:

Product Key	Product Description	Department	Prior Department	SKU Number (Natural Key)
12345	IntelliKidz 1.0	Strategy	Education	ABC922-Z

Kimball's three approaches require some effort when executing. As a side effect, such approaches make querying and writing reports rather complicated affairs.

So how do you handle SCDs today?

In a 2018 talk at Data Council, senior Lyft data engineer Maxime Beauchemin describes an approach that is currently used in Facebook, Airbnb, and Lyft.

The approach is simple: many modern data warehouses support a table partitioning feature. Beauchemin's idea is to use an ETL tool to create and copy new table partitions as a 'snapshot' of *all* the dimensional data, on a daily or weekly basis.

This approach has a number of benefits:

1. As Beauchemin puts it: “Compute is cheap. Storage is cheap. Engineering time is expensive.” This approach is as pure a tradeoff between computational resources and engineering time.
2. Dimensional data is small and simple when compared to fact data. This means that even a couple thousand rows, snapshotted going back ten years, is a drop in the bucket for modern data warehouses.
3. Finally, snapshots give analysts an easy mental model to reason with, compared to the queries that you might have to write for a Type 2 or Type 3 response.

As an example of the third benefit, Beauchemin presents a sample query to demonstrate the simplicity of the mental model required for this approach:

```
--- With current attribute
select *
FROM fact a
JOIN dimension b ON
  a.dim_id = b.dim_id AND
  date_partition = `{{ latest_partition('dimension') }}`

--- With historical attribute
select *
FROM fact a
JOIN dimension b ON
  a.dim_id = b.dim_id AND
  a.date_partition = b.date_partition
```

Really simple stuff.

The key insight here is that *storage is really cheap today*. When storage is cheap, you can get away with 'silly' things like partitioning every dimension table every day, in order to get a full history of slowly changing dimensions.

As Beauchemin mentions at the end of his talk: "the next time someone talks to you about SCD, you can show them this approach and tell them it's solved."

Applying Kimball Style Dimensional Modeling to the Data Infrastructure of Today

So how do we blend traditional Kimball-style dimensional modeling with modern techniques?

We've built Holistics with a focus on data modeling, so naturally we think there *is* value to the approach. Here are some ideas from our practice, that we think can apply generally to your work in analytics:

Kimball-style dimensional modeling is effective

Let's give credit where credit is due: Kimball's ideas around the star schema, his approach of using denormalized data, and the notion of dimension and fact tables are powerful, time-tested ways to model data for analytical workloads. We use it internally at Holistics, and we recommend you do the same.

We think that the question isn't: 'is Kimball relevant today?' It's clear to us that the approach remains useful. The question we think *is* worth

asking is: 'is it possible to get the benefits of dimensional modeling without *all* the busy work associated with it?'

And we think the answer to that is an unambiguous yes.

Model As And When You Need To

We think that the biggest benefit of having gobsmacking amounts of raw computing power today is the fact that such power allows us increased flexibility with our modeling practices.

By this we mean that you should *model when you have to.*

Start with generating reports from the raw data tables from your source systems — especially if the reports aren't too difficult to create, or the queries not too difficult to write. If they are, model your tables to match the business metrics that are most important to your users — without too much thought for future flexibility.

Then, when reporting requirements become more painful to satisfy — and *only when they become painful to satisfy* — you may redo your models in a more formal dimensional modeling manner.

Why does this approach work? It works because transformations are comparatively easy when done within the same data warehouse. It is here that the power of the ELT paradigm truly shows itself. When you have everything stored in a modern data warehouse, you are able to change up your modeling approach as and when you wish.

This seems like a ridiculous statement to make — and can be! — especially if you read it within the context where Kimball originally

developed his ideas. *The Data Warehouse Toolkit* was written at a time when one had to create new ETL pipelines in order to change the shape of one's data models. This was expensive and time consuming. This is not the case with our approach: because we recommend that you centralize your raw data within a data warehouse first, you are able to transform them into new tables within the same warehouse, using the power of that warehouse.

This is even easier when coupled with tools that are designed for this paradigm.

What are some of these tools? Well, we've introduced these tools in the previous section of the book. We called these tools 'data modeling layer tools', and they are things like Holistics, dbt, and Looker.

The common characteristic among these tools is that they provide helpful structure and administrative assistance when creating, updating, and maintaining new data models. For instance, with Holistics, you can visualize the lineage of your models. With dbt and Looker, you can track changes to your models over time. Most tools in this segment allow you to do incremental updating of your models.

These tools then generate the SQL required to create new data models and persist them into new tables within the same warehouse. Note how there is no need to request data engineering to get involved to set up (and maintain!) external transformation pipelines. Everything happens in one tool, leveraging the power of the underlying data warehouse.

The upshot: it is no longer necessary to treat data modeling as a big, momentous undertaking to be done at the start of a data warehousing

project. With 'data modeling layer tools', you no longer need data engineering to get involved — you may simply give the task of modeling to anyone on your team with SQL experience. So: do it 'just-in-time', when you are sure you're going to need it.

Use Technology To Replace Labor Whenever Possible

A more general principle is to use technology to replace labor whenever possible.

We have given you two examples of this: inventory modeling, and dealing with slowly changing dimensions. In both, Kimball's approach demanded a level of manual engineering. The contemporary approach is to simply rely on the power of modern data infrastructure to render such manual activities irrelevant.

With inventory modeling, we argued that the power of MPP columnar data warehouses made it possible to skip aggregation tables ... unless they were absolutely necessary. Your usage should drive your modeling requirements, and not the other way around.

With SCDs, we presented an approach that has been adopted at some of the largest tech companies: that is, recognize that storage is incredibly cheap today, and use table partitions to snapshot dimensional data over time. This sidesteps the need to implement one of the three responses Kimball details in his approach.

In both cases, the idea is to **critically evaluate the balance between computing cost and labor cost**. Many of Kimball's techniques should not be adopted if you can find some way to sidestep it using contemporary cloud data warehousing functionality.

Data architects trained in the old paradigm are likely to balk at this approach. They look at potential cloud DW costs, and gasp at the extra thousands of dollars you might have to pay if you push the heavy-lifting to the data warehouse. But remember this: it is usually far more costly to hire an extra data engineer than it is to pay for the marginal cost of DW functionality. Pushing BigQuery to aggregate terabytes of data might cost you an extra 1000 dollars of query time a month. But hiring an extra data engineer to set up and maintain a pipeline for you is going to cost many times more than that, especially if you include the full cost of employee benefits.

Conclusion

Think holistically about your data infrastructure. The best companies we work with do more with fewer people. They use the power of their data warehouses to increase the impact of the people they have, and choose to hire data analysts (who create reusable models) over data engineers (who create extra infra).

You should consider doing the same.

Modeling Example: A Real-world Use Case

In this section we are going to walk through a real world data modeling effort that we executed in Holistics, so that you may gain a better understanding of the ideas we've presented in the previous two segments. The purpose of this piece is two-fold:

1. We want to give you a taste of what it's like to model data using a data modeling layer tool. Naturally, we will be using Holistics, since that is what we use internally to measure our business. But the general approach we present here is what is important, as the ideas we apply are similar regardless of whether you're using Holistics, or some other data modeling layer tool like dbt or Looker.
2. We want to show you how we think about combining the Kimball-style, heavy, dimensional data modeling approach with the more 'just-in-time', lightweight, 'model how you like' approach. This example will show how we've evolved our approach to modeling a particular section of our data over the period of a few months.

By the end of this segment, we hope to convince you that using a data modeling layer-type tool along with the ELT approach is the right way to go.

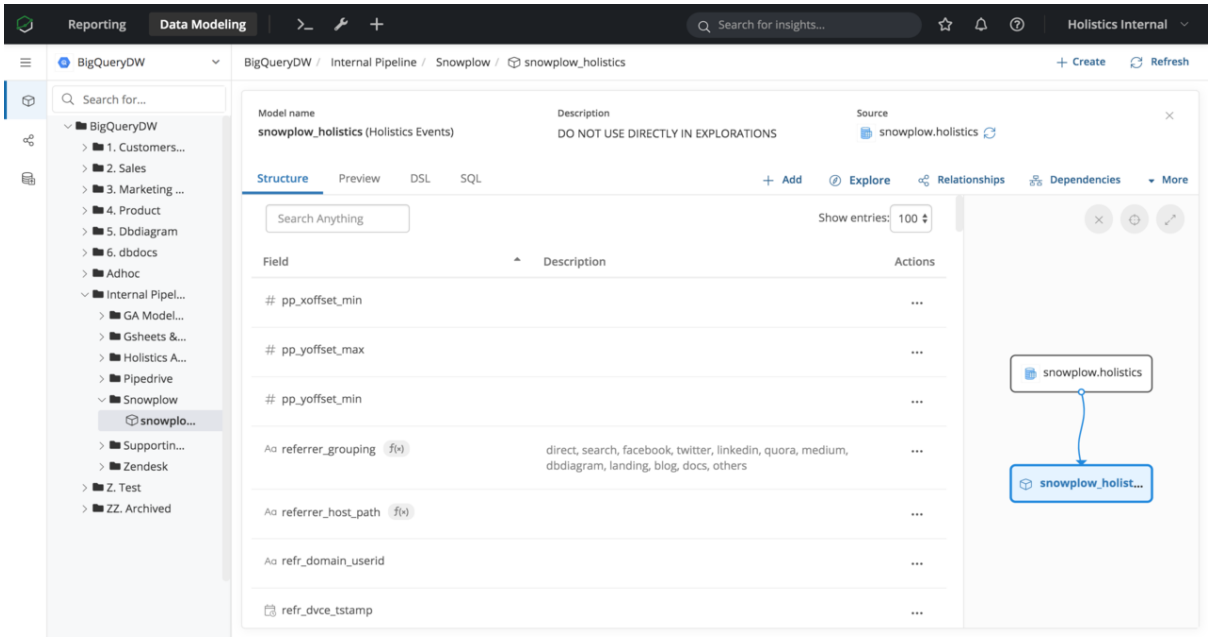
The Problem

In the middle of 2019, we began to adopt Snowplow as an alternative to Google Analytics for all our front-facing marketing sites. Snowplow is an open-source data delivery platform. It allows us to define and record events for any number of things on https://www.holistics.io/ —

if you go to the website and click a link, watch a video, or navigate to our blog, Snowplow's Javascript tracker captures these events and pushes them to the Snowplow event collector that runs on our servers.

Our Snowplow data pipeline captures and delivers such event data to BigQuery. And our internal Holistics instance sits on top of this BigQuery data warehouse.

Snowplow raw event data is extremely granular. The first step we did was to take the raw event data and model it, like so:



Note that there are over 130 columns in the underlying table, and about 221 fields in the data model. This is a *large* fact table by most measures.

Our data team quickly realized two things: first, this data was going to be referenced *a lot* by the marketing team, as they checked the

performance of our various blog posts and landing pages. Second, the cost of processing gigabytes of raw event data was going to be significant given that these reports would be assessed so regularly.

Within a few days of setting up Snowplow, we decided to create a new data model on which to run the majority of our reports. This data model would aggregate raw event data to the grain of the *pageview*, which is the level that most of our marketers operated at.

Notice a few things that went into this decision. In the previous section on Kimball data modeling we argued that it wasn't strictly necessary to write aggregation tables when working with large fact tables on modern data warehouses. Our work with the Snowplow data happened within BigQuery — an extremely powerful MPP data warehouse — so it was actually pretty doable to just run aggregations off the raw event data.

But our reasoning to write a new data model was as follows:

- The series of dashboards to be built on top of the Snowplow data would be used *very* regularly. We knew this because various members of the sales & marketing teams were already asking questions in the week that we had Snowplow installed. This meant that the time cost of setting up the model would be justified over the course of doing business.
- We took into account the costs from running aggregation queries across hundreds of thousands of rows every time a marketer opened a Snowplow-related report. If this data wasn't so regularly accessed, we might have let it be (our reasoning: don't waste employee time to reduce BigQuery compute costs if a report isn't

going to be used much!) but we thought the widespread use of these reports justified the additional work.

Notice how we made the decision to model data by considering multiple factors: the time costs to create a new model, the expected usage rate, and our infrastructure costs. This is very different from a pure Kimball approach, where every data warehousing project necessarily demanded a data modeling effort up-front.

Creating The Pageview Model

So how did we do this? In Holistics, we created this pageview-level data model by writing some custom SQL (don't read the whole thing, just skim — this is for illustration purposes only):

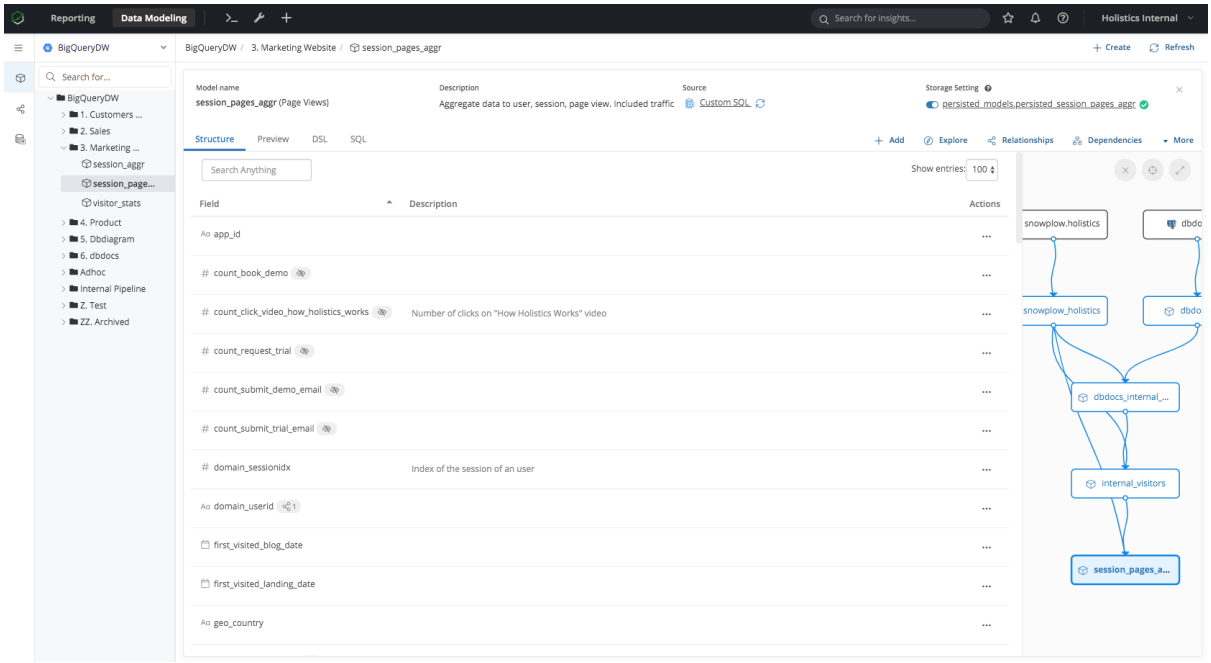
```
with
page_view_stats as (
  select
    {{#e.domain_userid}}
    , {{#e.domain_sessionidx}}
    , {{#e.session_id}} as session_id
    , wp.id as page_view_id
    , {{#e.app_id}}

    , min({{#e.derived_tstamp}}) as pv_start_at
    , max({{#e.derived_tstamp}}) as pv_stop_at
    , timestamp_diff(max({{#e.derived_tstamp}}), min({{#e.derived_tstamp}}),
second) as time_spent_secs
    , timestamp_diff(max({{#e.derived_tstamp}}), min({{#e.derived_tstamp}}),
second) / 60 as time_spent_mins

    , max((case when {{#e.pp_yoffset_max}} > {{#e.doc_height}} then
{{#e.doc_height}}
          else {{#e.pp_yoffset_max}} end) / {{#e.doc_height}}) as
max_scroll_depth_pct

  from {{#snowplow_holistics e}}
  left join
```

```
unnest({{#e.contexts_com_snowplowanalytics_snowplow_web_page_1_0_0}})
as wp
  left join {{#internal_visitors iv}} on {{#e.domain_userid}} =
{{#iv.domain_userid}}
  where {{#e.app_id}} in ('landing', 'docs', 'blog')
    and cast(derived_tstamp as date) >= '2019-07-30'
    and {{#iv.domain_userid}} is null
  group by 1, 2, 3, 4, 5
)

, session_stats as (
  select
    p.domain_userid
    , p.session_id as session_id
    , min(p.pv_start_at) as session_started_at
    , cast(min(p.pv_start_at) as date) as session_started_date
    , sum(time_spent_mins) as session_time_mins
    , round(sum(time_spent_mins) / 60) as session_time_hours
  from page_view_stats p
  group by 1, 2
)

, visitor_stats as (
  select
    p.domain_userid
    , cast(min(case when app_id in ('landing', 'docs') then p.pv_start_at else null
end) as date) as first_visited_landing_date
    , cast(min(case when app_id = 'blog' then p.pv_start_at else null end) as date)
as first_visited_blog_date
  from page_view_stats p
  group by 1
)


select
  {{#e.app_id}}
  , {{#e.domain_userid}}
  , vs.first_visited_landing_date
  , vs.first_visited_blog_date

  , {{#e.domain_sessionidx}}
  , {{#e.session_id}} as session_id
  , ss.session_started_at
  , ss.session_started_date
```

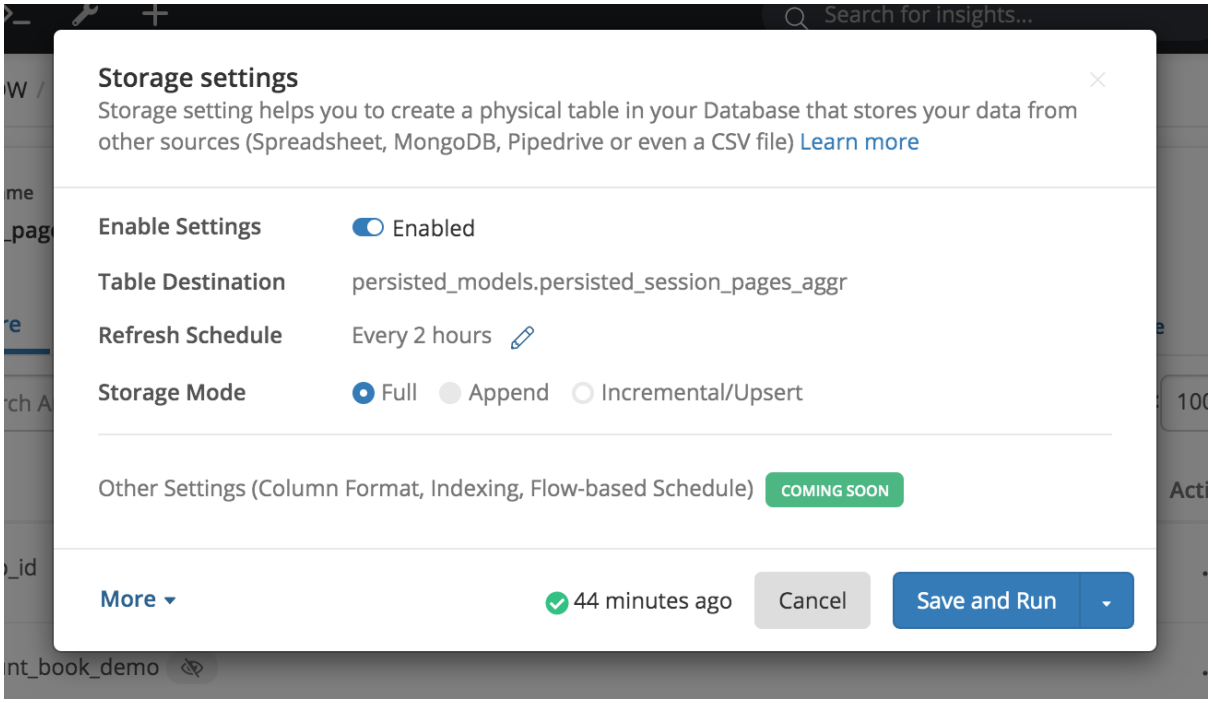
```
  , {{#e.mkt_source_grouping}} as mkt_source_grouping
  , {{#e.utm_source_grouping}} as utm_source_grouping
  , {{#e.utm_referrer_grouping}} as utm_referrer_grouping
  , {{#e.mkt_medium}}
  , {{#e.mkt_campaign}}
  , {{#e.os_timezone}}
  , {{#e.geo_country}}

  , case when {{#e.refr_urlhost}} = 'www.holistics.io' and {{#e.utm_referrer}} is not
null
      then {{#e.utm_referrer_host}} else {{#e.refr_urlhost}} end as refr_urlhost
  , case when {{#e.refr_urlhost}} = 'www.holistics.io' and {{#e.utm_referrer}} is not
null
      then {{#e.utm_referrer_path}} else {{#e.refr_urlpath}} end as refr_urlpath
  , case when {{#e.refr_urlhost}} = 'www.holistics.io' and {{#e.utm_referrer}} is not
null
      then coalesce({{#e.utm_referrer}}, '/')
      else concat({{#e.refr_urlhost}}, coalesce({{#e.refr_urlpath}}, '/')) end as referrer

  , {{#e.referrer_grouping}} as referrer_grouping

  , {{#e.page_urlhost}}
  , {{#e.page_urlpath}}
  , concat({{#e.page_urlhost}}, coalesce({{#e.page_urlpath}}, '/')) as page
  , {{#e.page_grouping}} as page_grouping

  , wp.id as page_view_id
  , pvs.pv_start_at
  , pvs.pv_stop_at

  , coalesce(pvs.max_scroll_depth_pct, 0) as max_scroll_depth_pct
  , pvs.time_spent_secs as time_on_page_secs
  , pvs.time_spent_mins as time_on_page_mins

  -- Actions aggregation
  , {{#e.count_click_vid_how_holistics_works}} as
count_click_video_how_holistics_works
  , {{#e.count_submit_demo_email}} as count_submit_demo_email
  , {{#e.count_book_demo}} as count_book_demo
  , {{#e.count_submit_trial_email}} as count_submit_trial_email
  , {{#e.count_request_trial}} as count_request_trial

from {{#snowplow_holistics e }}
, unnest(
{{#e.contexts_com_snowplowanalytics_snowplow_ua_parser_context_1_0_0}})
```

```
as ua
left join unnest(
{{#e.contexts_com_snowplowanalytics_snowplow_web_page_1_0_0}}) as wp

left join session_stats ss on {{#e.session_id}} = ss.session_id
left join page_view_stats pvs on {{#e.session_id}} = pvs.session_id and wp.id =
pvs.page_view_id
left join visitor_stats vs on {{#e.domain_userid}} = vs.domain_userid
left join {{#internal_visitors iv}} on {{#e.domain_userid}} = {{#iv.domain_userid}}

where
  {{#e.app_id}} in ('landing', 'docs', 'blog')
  and {{#e.event}} != 'page_ping'
  and cast({{#e.derived_tstamp}} as date) >= '2019-07-30'
  and {{#e.is_test}} = false
  and {{#iv.domain_userid}} is null

  -- Remove bots
  and not regexp_contains(ua.useragent_family,'(?i)
(bot|crawl|slurp|spider|archiv|spinn|sniff|seo|audit|survey|pingdom|worm|capture|
(browser|screen)shots|analyz|index|thumb|check|YandexBot|Twitterbot|a_archiver|
 facebookexternalhit|Bingbot|Googlebot|Baiduspider|360(Spider|User-agent))')
  and  coalesce(regexp_contains( {{#e.refr_urlhost}}, 'seo'), false ) is false
  and  {{#e.page_urlhost}} != 'gtm-msr.appspot.com'
  and ({{#e.refr_urlhost}} != 'gtm-msr.appspot.com' or {{#e.refr_urlhost}} is null)
group by 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29
```

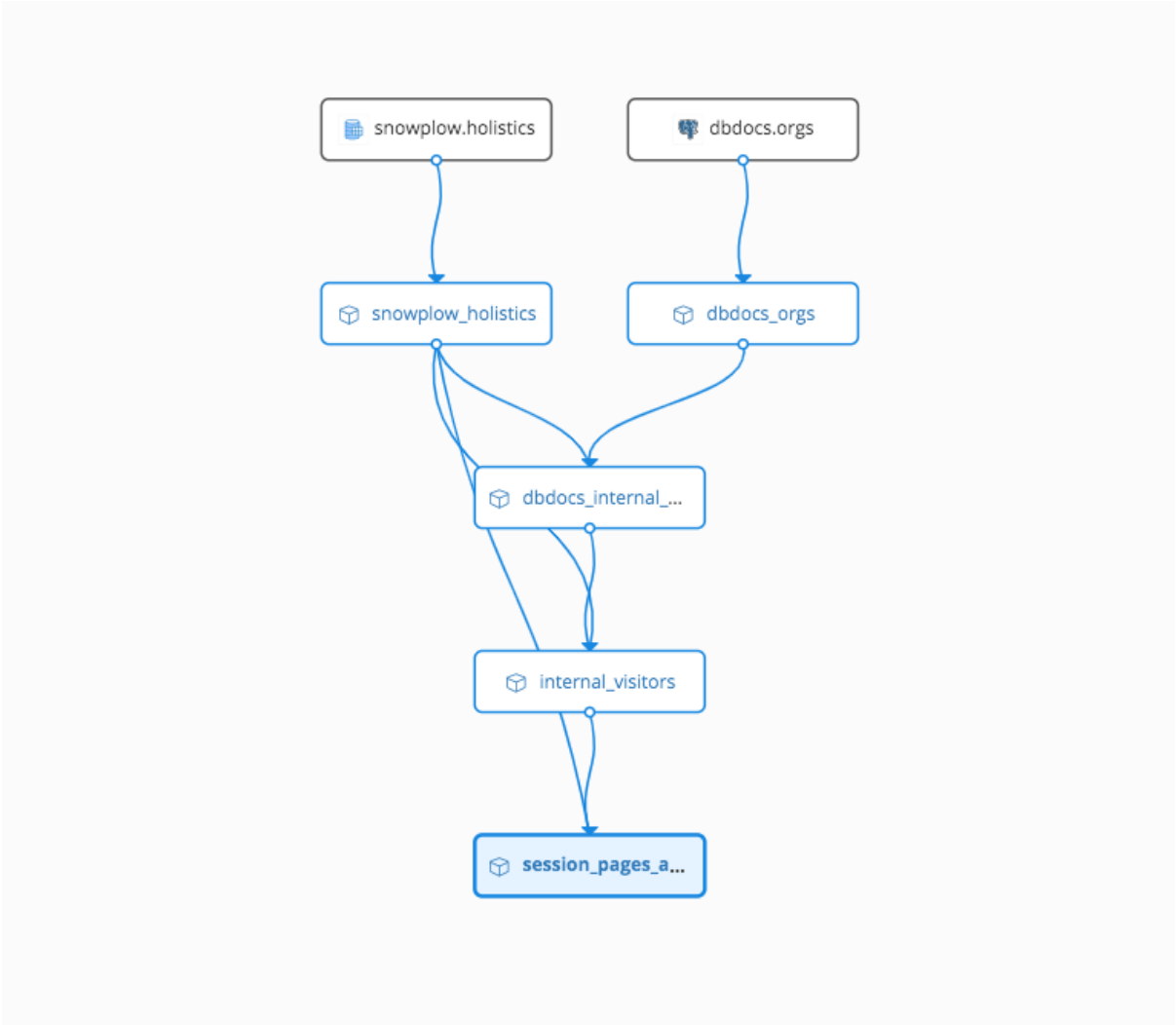
Within the Holistics user interface, the above query generated a model that looked like this:



We then persisted this model to a new table within BigQuery. The persistence settings below means that the SQL query you saw above would be rerun by the Holistics data modeling layer once ever two hours. We could modify this refresh schedule as we saw fit.



We could also sanity check the data lineage of our new model, by peeking at the dependency graph generated by Holistics:



In this particular case, our pageview-level data model was generated from our Snowplow event fact table in BigQuery, along with a `dbdocs_orgs` dimension table stored in PostgreSQL. (dbdocs is a separate product in our company, but our landing pages and marketing materials on Holistics occasionally link out to dbdocs.io — this meant it was important for the same people to check marketing performance for that asset as well).

Our reports were then switched over to this data model, instead of the raw event fact table that they used earlier. The total time taken for this

effort: half a week.

Evolving The Model To A Different Grain

A few months later, members of our marketing team began to ask about funnel fall-off rates. We were running a couple of new campaigns across a handful of new landing pages, and the product side of the business began toying with the idea of freemium pricing for certain early-stage startup customers.

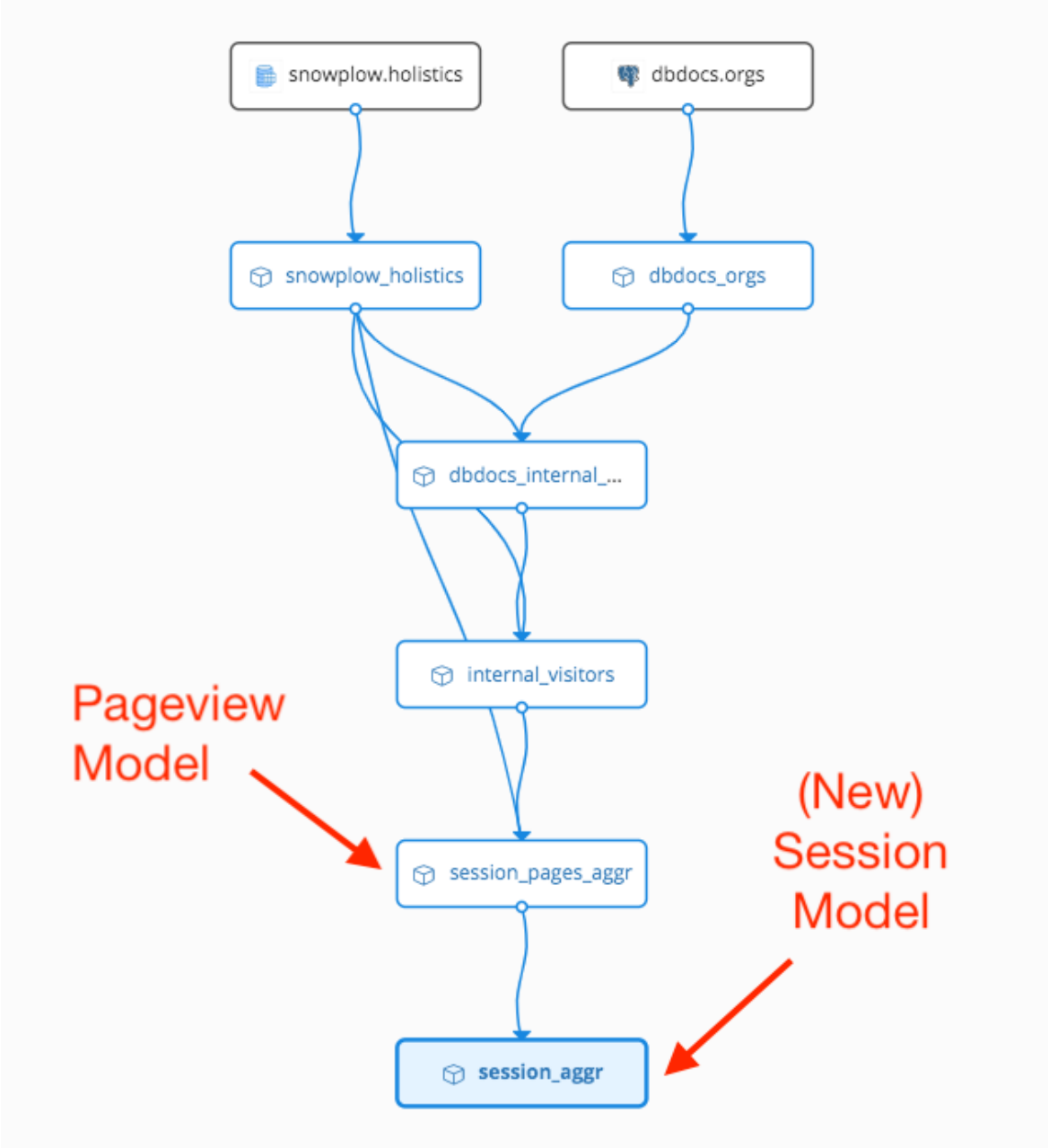
However, running such marketing efforts meant watching the bounce rates (or fall-off rates) of our various funnels very carefully. As it turned out, this information was difficult to query using the pageview model. Our data analysts found that they were writing rather convoluted queries because they had to express all sorts of complicated business logic within the queries themselves. For instance, a 'bounced session' at Holistics is defined as a session with:

- only one page view, with no activities in any other sessions, or
- a session in which the visitor did not scroll down the page, or
- a session in which the visitor scrolled down but spent less than 20 seconds on the page.

Including complex business logic in one's SQL queries was a 'code smell' if there ever was one.

The solution our data team settled on was to create a new data model — one that operated at a **higher grain** than the pageview model. We wanted to capture 'sessions', and build reports on top of this session data.

So, we created a new model that we named `session_aggr`. This was a data model that was derived from the pageview data model that we had created earlier. The lineage graph thus looked like this:

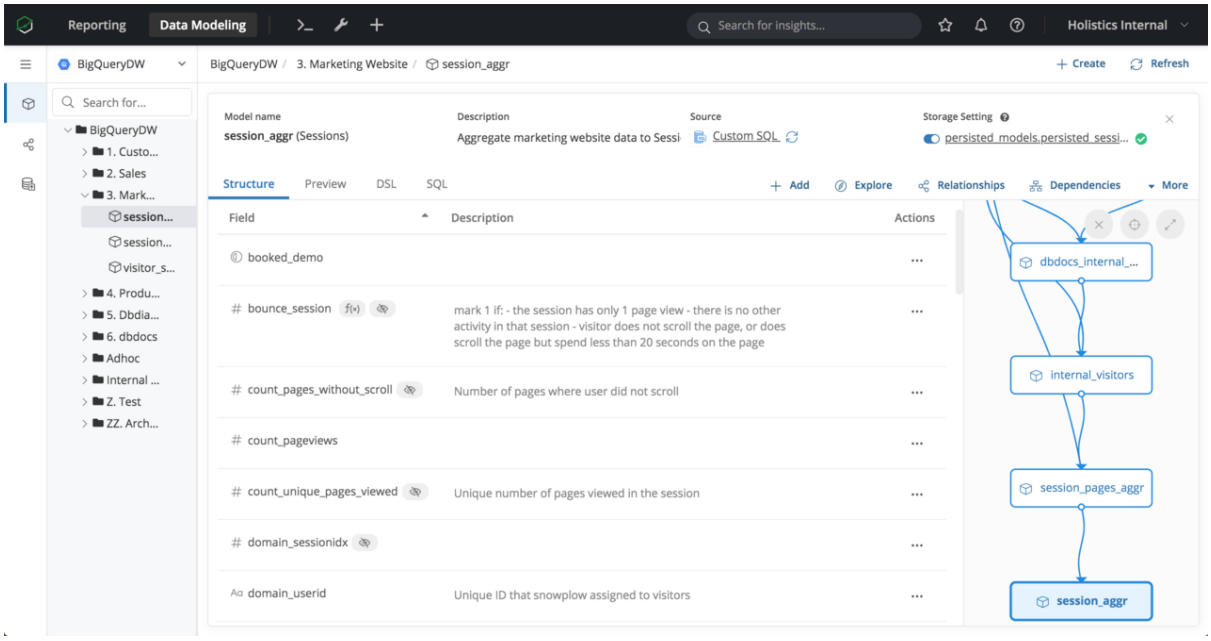


And the SQL used to generate this new data model from the pageview model was as follows (again, skim it, but don't worry if you don't understand):

```
#standardsql
with
session_ts as (
  select
    {{#s.domain_userid}}
    , {{#s.domain_sessionidx}}
    , {{#s.session_id}} as session_id
    , min( {{#s.session_started_at}}) as session_started_at
    , max( {{#s.pv_stop_at}}) as session_latest_ts
  from {{#session_pages_aggr s}}
  group by 1, 2, 3
)

, first_page as (
  select * from (
    select
      {{#p1.domain_userid}}
      , {{#p1.domain_sessionidx}}
      , {{#p1.session_id}}
      , {{#p1.mkt_source_grouping}} as mkt_source
      , {{#p1.mkt_medium}}
      , {{#p1.mkt_campaign}}

      , {{#p1.page_urlhost}} as first_page_host
      , {{#p1.page}} as first_page
      , {{#p1.page_grouping}} as first_page_grouping

      , {{#p1.refr_urlhost}} as first_referrer_host
      , {{#p1.referrer}} as first_referrer
      , {{#p1.referrer_grouping}} as first_referrer_grouping

      , row_number() over (partition by {{#p1.domain_userid}},
{{#p1.domain_sessionidx}} order by {{#p1.pv_start_at}} asc) as page_idx
    from {{#session_pages_aggr p1}}
  ) where page_idx = 1
)

select
  {{#p.domain_userid}}
  , {{#p.domain_sessionidx}}
  , {{#p.session_id}}
  , st.session_started_at
  , st.session_latest_ts
  , cast(st.session_started_at as date) as session_started_date
```

```
  , fp.mkt_source
  , fp.mkt_medium
  , fp.mkt_campaign

  , first_referrer_host
  , first_referrer
  , first_referrer_grouping

  , first_page_host
  , first_page
  , first_page_grouping

  , string_agg( concat({{#p.page_urlhost}}, {{#p.page_urlpath}}) ) as visited_pages

  , {{#p.count_pageviews}} as count_pageviews
  , {{#p.count_unique_pages_viewed}} as count_unique_pages_viewed
  , {{#p.count_pages_without_scroll}} as count_pages_without_scroll
  , {{#p.sum_time_on_page_secs}} as total_session_time_secs
  , {{#p.sum_time_on_page_mins}} as total_session_time_mins

  -- demo
  , sum({{#p.count_submit_demo_email}}) > 0 as submitted_demo_email
  , sum( {{#p.count_book_demo}}) > 0 as booked_demo

  -- trial
  , sum({{#p.count_submit_trial_email}}) > 0 as submitted_trial_email
  , sum({{#p.count_request_trial}}) > 0 as requested_trial


from {{#session_pages_aggr p}}
left join session_ts st on {{#p.session_id}} = st.session_id
left join first_page fp on {{#p.session_id}} = fp.session_id
group by 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
```

And in the Holistics user interface, this is what that query looked like (note how certain fields were annotated by our data analysts; this made it easier for marketing staff to navigate in our self-service UI later):



This session model is regenerated from the pageview model once every 3 hours, and persisted into BigQuery with the table name `persisted_models.persisted_session_aggr`. The Holistics data modeling layer would take care to regenerate the pageview model first, before regenerating the session model.

With this new session data model, it became relatively easy for our analysts to create new reports for the marketing team. Their queries were now very simple `SELECT` statements from the session data model, and contained no business logic. This made it a lot easier to create and maintain new marketing dashboards, especially since all the hard work had already been captured at the data modeling layer.

Exposing self-service analytics to business users

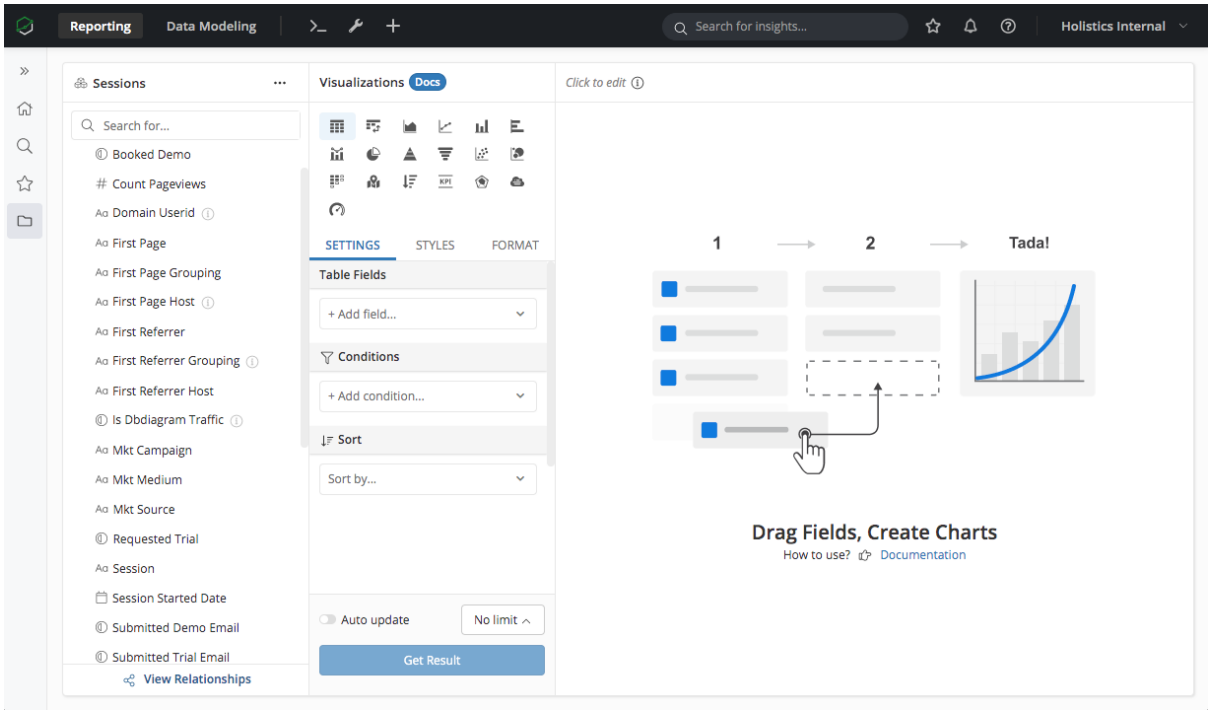
It's worth it to take a quick look at what all of this effort leads to.

In *The Data Warehouse Toolkit*, Ralph Kimball championed data modeling as a way to help business users navigate data within the data warehouse. In this, he hit on one of the lasting benefits of data modeling.

Data modeling in Kimball's day really *was* necessary to help business users make sense of data. When presented with a BI tool, non-technical users could orient themselves using the labels on the dimensional tables.

Data modeling serves a similar purpose for us. We don't think it's very smart to have data analysts spend all their time writing new reports for business users. It's better if their work could become reusable components for business users to help themselves.

In Holistics, the primary way this happens is through **Holistics Datasets** — a term we use to describe self-service data marts. After model creation, an analyst is able to package a set of data models into a (waitforit) dataset. This dataset is then made available to business users. The user interface for a dataset looks like this:



On the leftmost column are the fields of the models collected within the data set. These fields are usually self-describing, though analysts take care to add textual descriptions where the field names are ambiguous.

In Holistics, we train business users to help themselves to data. This interface is key to that experience. Our business users drag whatever field they are interested in exploring to the second column, and then generate results or visualizations in the third column.

This allows us to serve measurements throughout the entire organization, despite having a tiny data team.

Takeaways

What are some lessons we may take away from this case study? Here are a few that we want to highlight.

Let Usage Determine Modeling, Not The Reverse

Notice how sparingly we've used Kimball-style dimensional data modeling throughout the example, above. We only have one dimension table that is related to dbdocs (the aforementioned `dbdoc.org` table). As of right now, most dimensional data is stored within the Snowplow fact table itself.

Is this ideal? No. But is it enough for the reports that our marketing team uses? Yes, it is.

The truth is that if our current data model poses problems for us down the line, we can always spend a day or two splitting out the dimensions into a bunch of new dimension tables according to Kimball's methodology. Because all of our raw analytical data is captured in the same data warehouse, we need not fear losing the data required for future changes. We can simply redo our models within Holistics's data modeling layer, set a persistence setting, and then let the data warehouse do the heavy lifting for us.

Model Just Enough, But No More

Notice how we modeled pageviews *first* from our event data, and sessions later, only when we were requested to do so by our marketing colleagues. We *could* have speculatively modeled sessions early on in our Snowplow adoption, but we didn't. We chose to guard our data team's time judiciously.

When you are in a fast-moving startup, it is better to do just enough to deliver business insights today, as opposed to crafting beautiful data models for tomorrow. When it came time to create the session data model, it took an analyst only two days to come up with the SQL and to materialize it within Holistics. It then took only another day or so to attach reports to this new data model.

Use such speed to your advantage. Model only what you must.

Embed Business Logic in Data Models, Not Queries

Most of the data modeling layer tools out there encourage you to pre-calculate business metrics within your data model. This allows you to keep your queries simple. It also prevents human errors from occurring.

Let's take the example of our 'bounced session' definition, above. If we had not included it in the sessions model, this would mean that all the data analysts in our company would need to remember exactly how a bounced session is defined by our marketing people. They would write their queries according to this definition, but would risk making subtle errors that might not be caught for months.

Having our bounced sessions defined in our sessions data model meant that our reports could simply `SELECT` off our model. It also meant that if our marketing team changed their definition of a bounced session, we would only have to update that definition in a single place.

The Goal of Modeling Is Self Service

Like Kimball, we believe that the end goal of modeling is self-service. Self-service is important because it means that your organization is no longer bottlenecked at the data team.

At Holistics, we've built our software to shorten the gap between modeling and delivery. But it's important to note that these ideas aren't limited to just our software alone. A similar approach using slightly different tools are just as good. For instance, Looker is known for its self-service capabilities. There, the approach is somewhat similar: data analysts model up their raw tables, and then use these models to service business users. The reusability of such models is what gives Looker its power.

Going Forward

We hope this case study has given you a taste of data modeling in this new paradigm.

Use a data modeling layer tool. Use ELT. And what you'll get from adopting the two is a flexible, easy approach to data modeling. We think this is the future. We hope you'll agree.

Chapter 4:
Using Data

Data Servicing — A Tale of Three Jobs

This chapter is about delivering data to your users.

There are many people-oriented considerations when it comes to data delivery, especially when compared to the topics we've covered in the previous three chapters. For instance, we may talk about how to structure your data team for the organization you're in, how to push acceptance of metrics and data-oriented thinking within your company, and how to *not* feel like an English-to-SQL translation monkey whenever your CEO asks for metrics for the nth time.

This makes sense: when you're delivering data to business people, it helps if we talk a little about the *people* end of things, not just the data end of things. The end goal of the analytics process is to discover some actionable insight about the business. It is reasonable to think that we should spend at least an equal amount of time thinking about the effectiveness of our business intelligence as we have about the infrastructure that goes into delivering it.

Alas, we do *not* plan to dive into many of these people-oriented topics. Our goal with this book is to give you a high-level overview of analytics stacks — no more, no less. With that goal in mind, this chapter will give you three things:

- It will explain to you certain shifts that have happened in the past three decades, so that you will have a basic historical understanding of the tools you will encounter throughout your career. This will help orient you, and prevent you from feeling lost.

- It will give you a lay-of-the-land view of the entire business intelligence reporting market. This way, you'll be able to slot the tools into a couple of neat buckets in your head, and evaluate them accordingly.
- It will give you a taste of the evolution of reporting requirements you will see in your own company.

In some of these sections, we will discuss the people side of things, because it is inevitable that we do so. But you should understand that our focus is on giving you a lay-of-the-land orientation on this topic, and that you may find our discussion of the people side of things a little lacking. This is by design — a proper treatment of the people element will take a full book to tackle adequately!

Anyway, with that out of the way, let's get started.

A Tale of Three Jobs

In order to understand the current landscape of Business Intelligence tools, it helps to have a rudimentary understanding of the history of BI tools. To illustrate this, we'll return to our fictional data analyst Daniel from Chapter 3, and give you a rundown of all that he has experienced over the course of his career.

Daniel's First Job: Cognos and the Eternal Wait

When Daniel first started in business intelligence, he landed a job at a multinational and financial services corporation named the Divine People's Bank. 'Multinational and financial services corporation' is a long-winded way of saying that, yes, DPB was a bank, with all the normal trappings and regulatory requirements of a bank. Daniel

started as a data analyst in the consumer banking division. He wore a shirt and tie to work.

Daniel's day-to-day job was to deliver various performance reports to his manager, one of the VPs in the consumer banking arm. Above Daniel's manager was the head of consumer banking at DPB. In 1996, when Daniel was hired, the head of consumer banking was rumored to be a real taskmaster, and remarkably data-driven for his time.

Daniel hated his job.

Much of DPB's data was stored in a central data warehouse. This was a set of massive RDBMS databases that had been bought in a wave of digitization that DPB underwent in the late 80s. Daniel didn't know what these databases were — in fact, he never interacted directly with them. Instead, an 'IT services' team was assigned to him, and he interacted primarily with Cognos — at the time, one of the most dominant business intelligence tools on the market.

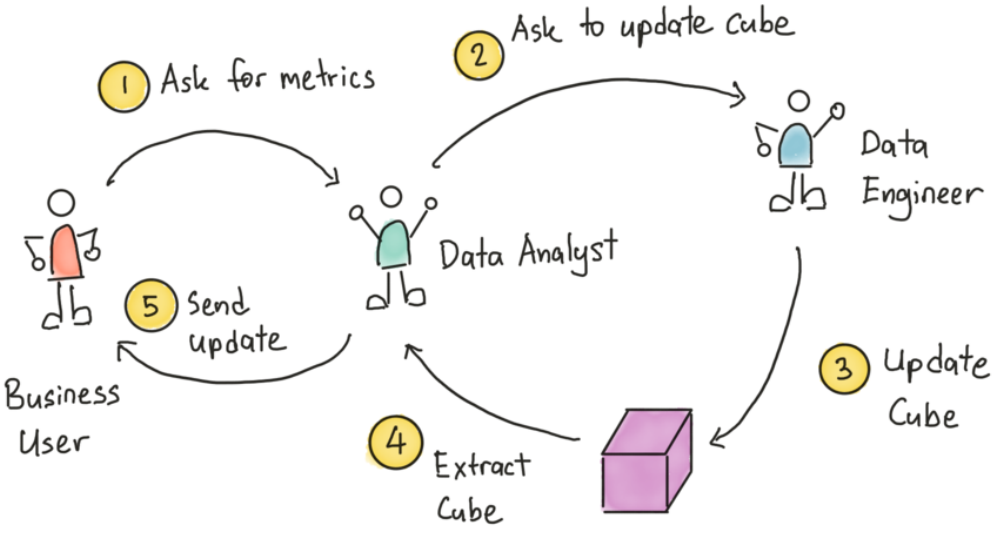
A typical day would look like this: Daniel's boss would ask for a set of numbers, and Daniel would go to his Cognos system to check the PowerPlay cubes that were available to him. Most of the time, the data would be in one or more cubes that had already been built by the 'IT Services' team (the cube would be built with a subset of the main data in the data warehouse). Daniel would point his PowerPlay client to the cubes he wanted on the bank's internal Cognos server, and then sliced and diced the data within the cube to extract the desired numbers for his boss. Most of the time, this went out in the form of an Excel spreadsheet — because Daniel's boss would want to do some additional analysis of his own.

(Note to less-technical readers: OLAP cubes or data cubes were efficient data structures that were optimized for data analysis. In the era that Daniel operated in, such cubes could not be very large, because many operations were done in memory; in fact, the maximum limit on Cognos PowerPlay cubes today remains at 6GB, due to internal limitations built into the data structure.)

The problems with Daniel's job emerged whenever Daniel's boss asked for numbers that he didn't have access to. Whenever that happened, Daniel would have to start a process which he quickly learned to hate. The process went something like this:

1. Verify that none of the cubes he currently had to access to contain the numbers that he needed to generate the report for his boss.
2. Contact the IT Services department with a work order. Within the work order, Daniel would input his request for new data to be added to an existing cube (or materialized into a new cube). This work order would then be sent to a central enterprise resource planning system, and the work order would count as an instance of inter-departmental resource usage; at the end of the month, a certain dollar amount would be taken out of Daniels's department's budget, and be marked up a payment to the IT Services department.
3. Wait three weeks.
4. At the end of three weeks, Daniel would be notified that the work order had been processed, and that his new data was waiting for him within the Cognos system. He might have to wait a few hours for the data to be refreshed, because Cognos Transformer servers took four hours on average to build a new PowerPlay cube.

5. If Daniel had made *any* mistake in his request, or left *any* ambiguity, he would have to go back to step 2 and start over.



Naturally, Daniel had to obsess over his work orders. The cost of delay with one bad request would be incredibly bad, because his boss would be expecting numbers by the end of the reporting period. Daniel lived under constant fear that the IT Services department would assign him a dim-witted data engineer; he also felt helpless that he had to rely on someone else to give him the resources he needed to do his job well.

What made things worse was when Daniel's boss's boss (yes, he of the fearsome data-driven reputation) dominated the requests of the other data analysts in Daniel's department. During such events, both the data analysts and the IT Services department would prioritize the big boss's request, leaving Daniel to fight over leftover resources at the services scrap table. It was during times like these that he was most likely to be assigned a dim-witted data engineer; over the course of a few years, Daniel learned to be SUPER careful with his work order requests whenever the big boss went on one of his data-requesting sprees.

Eventually, Daniel rose high enough in the ranks to count himself a senior data analyst. After 10 years at DPB, he left.

Daniel's Second Job: The Metrics Knife Fight

In 2006, Daniel joined an early video streaming company named YouDoo. YooDoo had a slightly updated business intelligence stack compared to the Divine People's Bank — they used Microsoft SQL Server as the basis for their datastore, built cubes in Microsoft SQL Server Analysis Services (or SSAS), and then fed data extracts from these systems to Tableau Desktop. Daniel also stopped wearing a tie to work.

At Youdoo, Daniel reported directly to the head of data, a veteran of the Cognos-style paradigm named Joe. "The goal here", said Joe, when Daniel came in on his first day of work, "The goal here is to give the business users and the PMs direct access to the company's data. If we can get them to do self-service, we would have less busy work to do!"

Daniel thought back to all the hell he went through in his previous job, and agreed that this sounded like a good idea.

Tableau desktop was and still is a beautiful piece of business intelligence software. It worked in the following manner: you would pull data out from your SQL database and dump it into a copy of Tableau running on your desktop machine. You would pull Excel spreadsheets and dump them into Tableau. You would pull up CSV files — sent to you by the data team — and dump them into Tableau.

Occasionally — though with a little more trepidation — you would connect Tableau to an OLAP cube, or directly to an SQL database

itself.

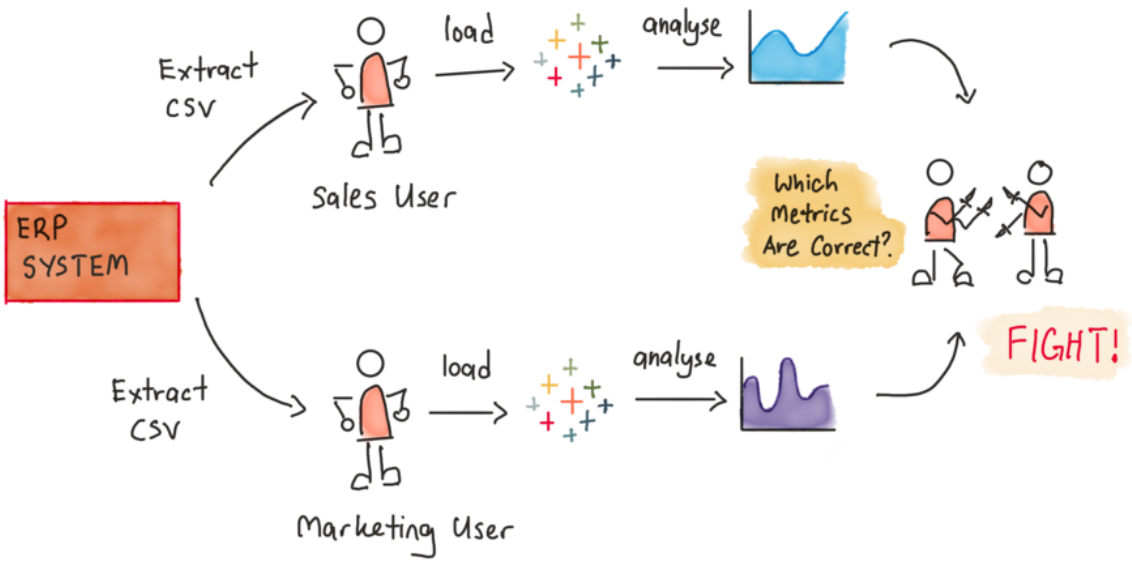
Then, you would use Tableau to create beautiful, *beautiful* visualizations for the company to consume. These would come in the form of colorful heatmaps, slick graphs, and shiny bar charts, delivered straight from the hands of the Tableau user to the business people in the company. The best bit about this was that Tableau was *completely drag-and-drop.* This meant that non-technical business users could learn to use Tableau and — assuming they got the right data extracts from the data team — could come up with fancy graphs for the rest of the company to consume.

From his time at YouDoo, Daniel learned that Tableau was essentially the best tool in a new breed of BI tools, all of which represented a new approach to analytics. This new approach assumed that the data team's job was to prepare data and make them available to business users. Then, capable business users could learn and use intuitive tools like Tableau to generate all the reports they needed.

But then came the problems.

It was six months into Daniel's tenure at YouDoo that he was first dragged into a metric knife fight. Apparently, marketing and sales were at loggerheads over something numbers-related for a couple of weeks now. Daniel and Joe were booked for a meeting with the respective heads of sales and marketing. They learned quickly that marketing's numbers (presented in a beautiful Tableau visualization, natch) didn't match sales's. Sales had exported their prospects from the same data sources as marketing's — what was going on?

Daniel dug into the data that week. Over the course of a few hours, he realized that marketing was using a subtly different formula to calculate their numbers. Sales was using the right definitions for this particular dispute — but they, too, had made subtle errors in a few other metrics. To his dawning horror, Daniel realized that multiple business departments had defined the same metrics in slightly different ways ... and that there was no company-wide standardization for measures across the company.



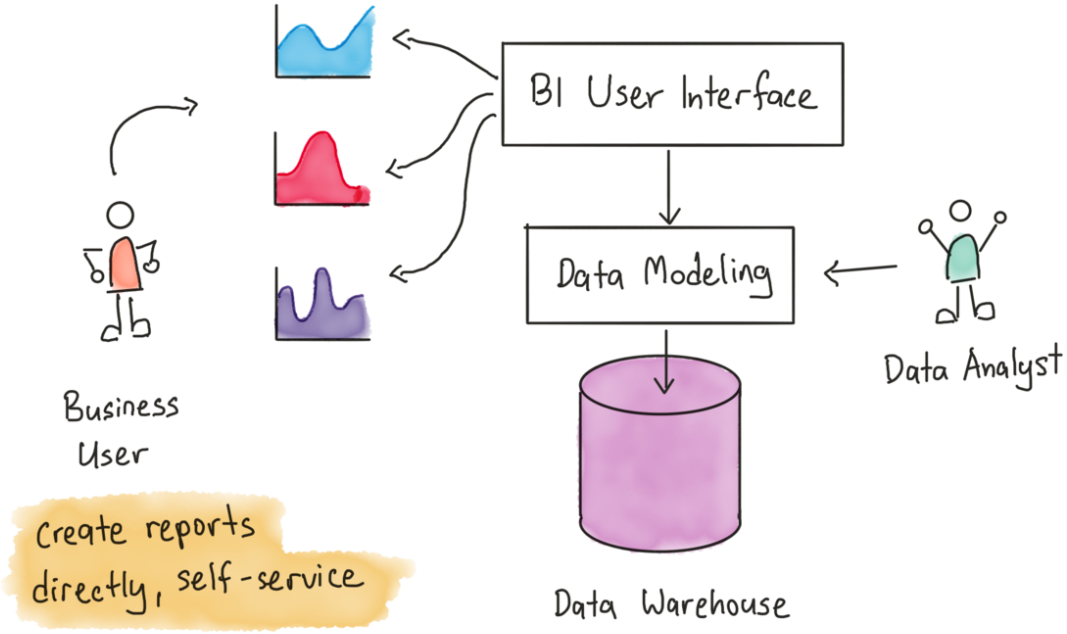
Daniel alerted Joe. Joe alerted the CEO. And the CEO called them into his office and exploded at both of them, because he had just presented the wrong numbers to the board in a quarterly meeting that had concluded the previous week. Daniel and Joe were forced to work overtime that day to get the numbers right. The CEO had to update the board members with a follow-up email, and then he issued a directive that all metric definitions were to be stored in a central location, to be maintained by the business intelligence team.

Daniel realized that this new Tableau workflow may have solved some problems ... but it led to others as well.

Daniel's Third Job: The Data Modeling Layer

Eight years later, in 2016, Daniel left YouDoo to work at a mid-stage startup named PropertyHubz. PropertyHubz used a relatively new Amazon cloud data warehouse called RedShift, along with a (then) two-year-old business intelligence tool named Looker. Along with the new stack, Daniel made other changes to his life: he dropped the shirt from his dress code entirely and came into work in a polo tee and pants.

Looker was amazing. Unlike the Cognos workflow Daniel started in, or the Tableau workflow he grappled with at his previous company, Looker assumed a completely different approach to data analytics. At PropertyHubz, Looker was hooked up to RedShift, and ran SQL queries directly against the database. Daniel's team of data analysts spent most of their time creating data models in LookML, the proprietary data modeling language in Looker. They then handed those models off to less-technical members of the organization to turn into dashboards, reports, and self-service interfaces.



Daniel could immediately see the benefits of building with this workflow. Unlike in the Tableau paradigm, business logic was written *once* — by the data team — in the data modeling layer. These models were then recombined by other analysts and by non-technical business users to produce the reports and exploratory interfaces the business needed. Daniel thought this was a step-up, because it sidestepped all the metrics drift they fought so hard to contain at YouDoo. But Daniel was also wary. He had seen his fair share of gotchas to know that nothing was perfect.

At this point in our story, Daniel had spent 18 years in business intelligence. He was scarred by the pain of waiting on IT Services in his first job, in the mid-90s. He spent many years grappling with metrics drift in his second, in the mid-2000s. As Daniel settled into his new role in PropertyHubz, he looked on all that he had experienced and thought that the new tools were definitely better — and easier! — than the old tools. He was eager to see what new problems this new

paradigm would bring — and in turn, what the *next* paradigm would do to solve them.

Frank Bien's Three Waves

In 2017, Looker CEO Frank Bien wrote a piece titled Catching the Third Wave of Business Intelligence. In it, he described exactly the three waves that we've just illustrated for you, above.

In his essay, Bien writes in a matter-of-fact way, as if each successive approach to BI emerged as surely as the tide, ebbing away the way it arrived. He — and the team at Looker — deserve much credit for creating the third wave of business intelligence. But the waves aren't as clearly delineated as real waves on a beach. The truth is more complicated.

Let's recap Bien's argument, before picking apart the parts that don't fully match with reality. Those parts will have real implications on how you view the business intelligence tool landscape today.

In the first wave of data analytics, companies developed and sold monolithic stacks — that is, an all-in-one solution that came with data warehouse, data transformer, data cube solution, and visualization suite. This approach evolved out of technical necessity as much as anything else.

What do we mean by this? Well, it is very easy to forget just how expensive hardware was in the 90s. In 1993, for instance, 1GB of RAM cost $32,300 — an insane amount of money for what seems like a piddling amount of memory today! Thus, the Cognos era of business intelligence tools had *no choice* but to have data engineers take

subsets of data and build them out into cubes: data warehouses were simply too expensive and too slow to be used for day-to-day analysis.

Naturally, this caused the analysts of that generation to bottleneck on data engineers, who were called on to build data cubes in response to business demands. The pains that Daniel experienced in his first job led to the development of the 'self-service'-oriented tools like Tableau that he experienced in his second.

In this 'second wave' of business intelligence, data cubes and Cognos-like stacks continued to evolve, but new tools championed a 'self-service' orientation. Tools like Tableau gave business users beautiful dashboards and visualizations with not a line of code in sight. These tools were in turn fed by data exports drawn from the earlier first-wave environment. The basic idea was that analysts and business users would download datasets from these central data systems, and then load these datasets into tools that they could install on their own computers.

Of course, as we've seen from Daniel's story, these tools came with their own set of problems.

It's important to note that even as these 'second-wave' tools came into prominence, the Cognos-type first-wave environments *continued to gain ground within large corporations*. It wasn't as if people adopted Tableau, and then the monolithic workflows went away. In fact, Cognos still exists today, albeit under IBM's umbrella of business intelligence tools. (Even the PowerPlay cubes that Daniel used at the beginning of his career are still part of the product!)

When Bien talks about the 'second-wave emerging', it's important to understand that reality is messier than the picture he paints. In our story with Daniel, for instance, his bank — like many other Cognos clients — continued to use and expand its usage of the product in the years since. Similar tools that emerged in competition with Cognos, like Microsoft's SSAS suite of tools, may be rightly considered first-wave or second-wave, but are **still going strong in large enterprises today**.

But some things *have* changed.

In the past decade, two major technological breakthroughs have shifted the landscape yet again:

- Massively parallel processing (MPP) data warehouses began to be a thing, and
- Columnar datastores began to match OLAP cubes in analytical performance.

The first is easy to understand: MPP data warehouses are data warehouses that are not limited to one machine. They may instead scale up to hundreds or thousands of machines as is needed for the task at hand. These data warehouses were often also coupled with another innovation — that is, the cloud vendor pricing model. Business today only have to pay for the storage and the computing power that they use: no more, no less.

The second breakthrough is only slightly more difficult to understand. Generally speaking, data warehouses of the past adopted a row-oriented relational database architecture. This architecture was not well-suited to analytical workloads, because analytical workloads

required rollups and aggregations over thousands of rows. This was the main reason that early BI vendors opted to slice off a small portion of data and load them into efficient data cubes, instead of running them inside databases.

In recent years, however, data warehouses have adopted what is called a columnar storage architecture. These columnar databases are built for analytical workloads, and are finally comparable in performance to data cube solutions.

This *doesn't* mean that the cube-oriented systems and the decentralized Tableau-type analytical workflows have vanished from our industry. In fact, many companies have doubled down on their investments in earlier generations of these tools. They have layered on *more* tools, or have had departments add additional tools from different waves in order to augment their existing BI capabilities.

But for new companies — and large tech companies like Google, Uber, Facebook and Amazon — business intelligence that is implemented today is often built entirely within the third wave. This is the viewpoint that this book has attempted to present.

In a sentence: modern data warehouses have finally become cheap enough and powerful enough to stand on their own. Looker was the first BI vendor to realize this. They built their entire offering around the MPP data warehouse ... and we've never looked back.

The Major Takeaways

We have presented Daniel's story in narrative form because we think it captures some of the nuances that are lost in Bien's presentation.

Daniel is, of course, not real. But his story is a pastiche of **real events** and **real problems** that were taken from analysts we know. The pains that Daniel felt were real pains experienced by thousands of analysts in the previous decade.

Why is this important to know? It is important to know because *the ideas that BI tools adopt are more important to understand than the selection of tools themselves*. As we walked you through Daniel's story, and then Bien's argument, three trends seem to emerge:

1. First, approaches in business intelligence tools are limited by the technology of the day.
2. Second, approaches in business intelligence are often reactions to pains in the previous generation of tools.
3. Third, as we've mentioned in the previous section on Frank Bien's essay: each generation sticks around for a long, long time.

In a very particular sense, the business intelligence world is confusing today for that third reason: tools and approaches stick around for a long time. A new purchaser in the market would be confused by the mix of terminologies, ideas, architectural diagrams and approaches available to her. Daniel's story should help explain why that is: many of these tools were developed in successive generations, yet co-exist uncomfortably today.

In the next section of this chapter, we will give you a taxonomy of business intelligence tools — that is, categories or buckets to lump things in. Many of these tools will reflect the ideas that we have presented here. Others are recombinations of old ideas, applied to new paradigms. But it is important to understand that all of them — Holistics included — are shaped by the three factors above.

The insight we want to give you here is that *ideas or approaches change slower than tools do*. If you understand this, it will be easier to evaluate a new BI tool when it comes to the market. You will be equipped to cut through all the industry hype, the noisy whitepapers, the expensive conferences, the industry jargon, the breathless Medium posts and the shouty vendor presentations. You will be able to find the signal in the noise.

Navigating The Business Intelligence Tool Space

In biology, a taxonomy is a scheme of classification for living things. It helps scientists in the field categorize different types of animals, such as 'this fish has a swim bladder' and 'this fish has a really prominent dorsal fin' and 'oh my god this fish has a lantern sticking out of its head'.

Business intelligence tools can be roughly categorized in the same way that scientists categorize animals. Having a taxonomy for the business intelligence space is useful because it allows you to quickly place the tool within the first few minutes of a sales call (or within the first few minutes of browsing a vendor's website). As we've mentioned in the previous section, the BI landscape can be incredibly confusing because tools from previous paradigms stick around for a long, long time. Having a categorization scheme in your head cuts through all that noise.

In this section, we're going to build on the historical narrative of the previous section by giving you some of the most useful ways you may categorize such tools.

SQL vs Non-SQL

Some business intelligence tools demand knowledge of SQL. Others do not.

We've already talked about Tableau's workflow in the previous section. Tableau assumes that data workers would have already transformed

the data into a form that is suitable for analysis before handing it to a business user. The Tableau user then loads such data into Tableau, in order to generate visualizations and reports. Tableau assumes no knowledge of SQL in order to produce such visualizations.

Similarly, BI tools that operate on top of OLAP cubes tend to not use SQL. For example, Microsoft's MDX language was developed specifically for operations on a data cube, and came to prominence with their SSAS suite. The language was then adopted by many other OLAP cube vendors in the market, and is today considered a solid alternative to SQL.

Other BI tools are unabashed about their SQL-orientation. Chartio, Redash, Mode, Cluvio and Holistics fall into this category. All of these tools come with powerful SQL query editors. All of them assume some familiarity with SQL.

There are variants on this approach, of course:

- Holistics treats direct SQL query creation as a secondary access path. We believe that data analysts should do the bulk of their business intelligence work with a data modeling layer.
- Chartio comes with a sophisticated Visual SQL mode, which makes SQL more accessible to non-technical users ... but without giving up any of the power of the underlying language.
- Mode Analytics assumes that data extracted from your data warehouse should be used for both data science (that is, available for analysis within Jupyter Notebooks) as well as for business intelligence. It has an in-memory datastore that sits in between the tool and your data warehouse, in order to serve both purposes.

- Looker requires all data analysts to write in LookML, which is then translated into SQL for the underlying database.

The shift from non-SQL based analytics to SQL based analytics

Before we close this section, we should point out that there's an existing shift *towards* SQL that has happened over the past five years.

This was not always obvious. Starting around 2010, there was a huge amount of hype around NoSQL datastores like MongoDB, Cassandra, and CouchDB. These datastores promised superior performance, but did *not* use SQL. There was also an earlier wave of excitement over big data technologies like Hadoop and Spark, the majority of which eschewed SQL for proprietary APIs.

Both trends have died in the years since. The vast majority of analytical datastores today have standardized around SQL. Even non-SQL datastores like the proprietary Spanner database from Google — and even Hadoop today! — have adopted SQL as a query language.

In an essay titled The Rise of SQL-Based Data Modeling and Data Ops, Holistics co-founder Thanh argued that the standardization around SQL happened for a few reasons:

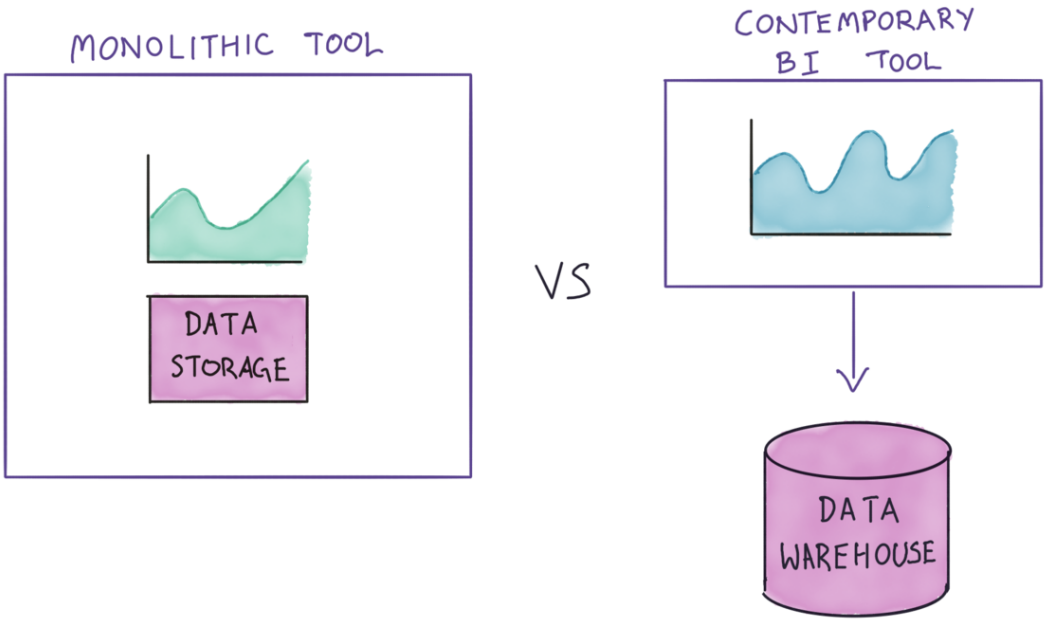
- Proficient SQL users were able to utilize the power of *any* SQL cloud-based data warehouse to produce beautiful charts and dashboards ... without the need to learn a new proprietary language or tool. This meant transferrable skills. It also meant that it was easier for companies to hire and train new analysts.
- SQL is text-based, and may be stored in a version control system. This made it trivially easy to manage.

- The dominance of SQL has led to an increased number of BI tools embracing SQL as a primary interface language.

We do not think this trend will reverse anytime soon, for the simple reason that standards only become more entrenched over time. The upshot here is that if you work in data, you should assume that SQL is the lingua franca of data analytics in the future. Pick tools accordingly.

Embedded Datastore vs External Datastore

Early business intelligence tools came with embedded datastores. This was as much for pragmatic reasons as anything else. For instance, the Cognos systems we discussed in the previous section came with embedded datastores, with the expectation that modeled data would be transferred from Cognos Transformer servers to Cognos PowerPlay cubes. This was the norm back then: if you wanted to buy a BI tool, you would be expected to buy the *entire* shebang — data warehouse, visualization suite and all.



Today, however, things are different. The majority of contemporary BI tools have opted to remain datastore agnostic, and to connect to as many data warehouse products as possible. This was a natural adaptation to the rise of the cost-effective, extremely powerful modern data warehouse. Tools like Metabase, Holistics, Chartio, and Redash belong to this category, as they come with connectors to pretty much every SQL datastore that exists on the market.

(This was also made easier because modern data warehouses have all standardized on SQL. See how awesome standardization is?)

Not every modern BI tool assumes an external datastore, of course. A notable exception to this is the original Sisense product that was launched in 2004. Sisense took an early columnar data warehouse and packaged a business intelligence tool around it. Their unique selling proposition was the fact that Sisense could run large datasets on commodity, off-the-shelf hardware — like a puny consumer laptop, for instance. This was remarkable for its time, especially given the dominant approach of purchasing expensive machines for data warehousing and cube materialization.

With that said, things changed in 2019 when Sisense acquired Periscope Data and rebranded it into the rather verbose 'Sisense for Cloud Data Teams'. This effectively means that they, too, offer a product that connects to an external datastore today.

Sidebar: Massively Parallel Processing vs Relational Database?

While we are on the topic of connecting to external databases, let's talk about what we mean when we say a 'modern data warehouse'. Holistics, for instance, connects to MySQL, SQL Server and

Postgresql, in addition to cloud data warehouses like Redshift, Snowflake, and BigQuery. Many tools have an equally broad set of connectors. Are these databases all counted as 'modern'?

The short answer is that no: they're not. We've covered this in a previous section, in Understanding The Data Warehouse, but let's go through this quickly, again. When we say 'modern data warehouse', we really mean data warehouses that have two properties:

- **They are a column-oriented database** (as opposed to a row-oriented database). Column-oriented databases are adapted for analytical workloads, and match OLAP cubes in performance.
- **They have a massively parallel processing (MPP) architecture** (as opposed to running on a single machine or a small cluster of machines). MPP architectures mean that such data warehouses arbitrarily scale up or down their computing resources depending on the complexity of your query. It also means that you might pay nothing up front, but will have to pay for compute time on a variable cost (pay for what you use) basis.

This is not to say that you can't use Holistics and other similar tools with RDBMSes like MySQL and Postgresql. In fact, we've seen many startups start out with a Postgres replica as a primary analytical database.

What it *does* mean, however, is that row-oriented databases like Postgres will eventually top out in performance once you reach large analytical data sizes. This performance limitation is what led to the 'data warehouse and cube' workflows of decades past, and what leads to the 'cloud-based data warehouse' workflows of today.

In-Memory vs In-Database

Another variant of this 'embedded datastore' vs 'external datastore' spectrum is the idea of 'in memory' vs 'in database' tools.

Tools like Holistics, Redash, Chartio, Metabase and Looker run SQL queries on top of a powerful database. The heavy lifting is done by the database itself; the connected BI tool merely grabs the results of generated queries and displays it to the user.

In contrast, a BI tool like Tableau or PowerBI assumes the analyst will take data *out* of the central data warehouse and run analysis on their own machines. In terms of workflow, this is similar to taking data out of a central system and then dumping it into Excel. The performance of the analytical tool is thus limited by the power of the tool itself, along with the computational resources of the analyst's machine.

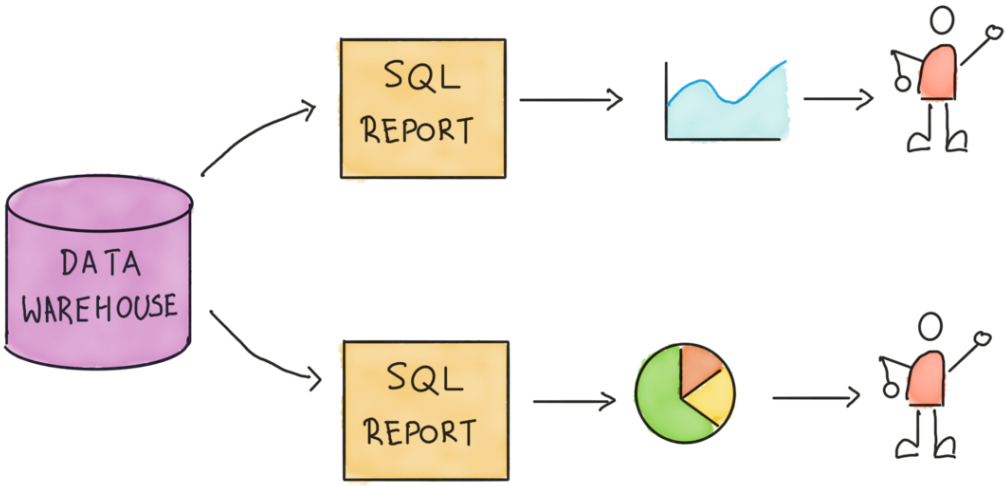
When you are evaluating BI tools, it helps to understand which process the tool assumes you would use. Does it leverage the power of the data warehouse? Or does it assume you're going to be pulling data out and running it on an individual analyst's machine?

Modeling vs Non-Modeling BI Tools

On that note, it's clear that certain BI tools combine modeling with their core offering, while others do not.

The Cluvio, Redash and Mode Analytics tools we've mentioned don't provide any modeling capabilities whatsoever. In practice, many contemporary data teams that we know either implement ELT techniques using data modeling layers like dbt or Dataform, or use a more traditional ETL approach using tools like Pentaho or Talend.

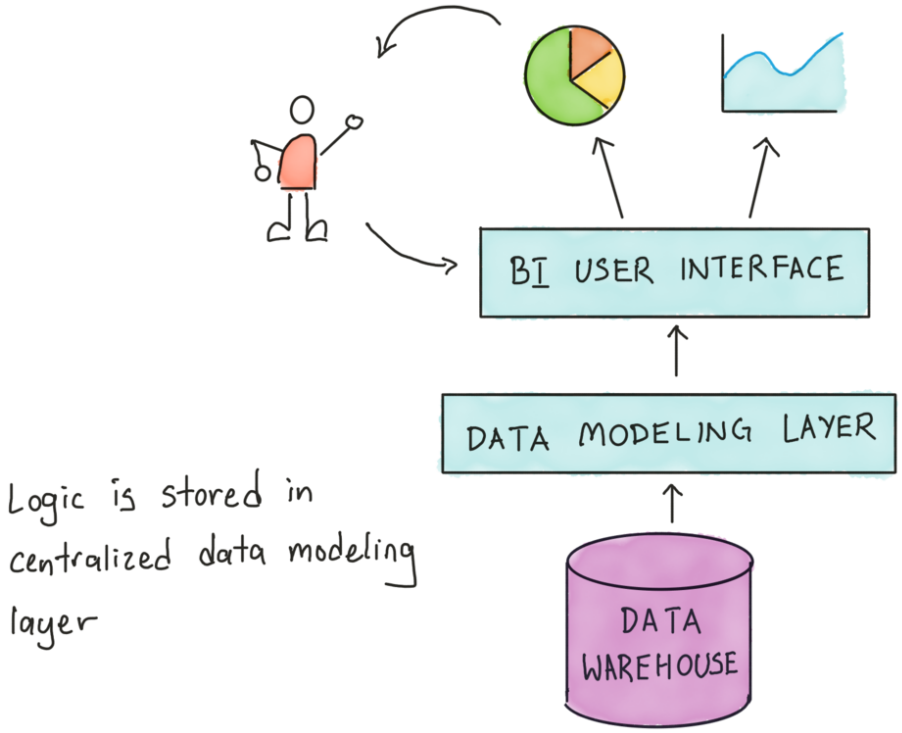
The key thing to take note of in a non-modeling BI approach is that either you modeled the data using a separate data modeling tool, or you have to hardcode the business logic directly into the report itself. If it's the latter, you stand the risk of getting into business logic discrepancy, because now there are multiple places in your BI system that contain duplicates of the same logic.



Logic is hardcoded in the SQL report

Holistics and Looker are somewhat unique in this sense, in that they *include* a modeling layer alongside BI functionality. Because of that, your entire logic is centralized in the data modeling layer, thus greatly increase metric consistency and logic reusability across the organization.

An additional benefit of having a modeling layer baked in the same BI tool is *maintaining context*: you are able to trace the full lineage of report data back to its original form, because the tool plays a part in every transformation along the way.



Our Biases, Revisited

In Chapter 1 of this book, we spent a section talking about our biases regarding a good analytics stack. We wrote then that our biases were:

- We prefer **ELT over ETL**
- We prefer using a **cloud data warehouse** over an on-premise data warehouse. We also prefer MPP analytics databases over Hadoop-like systems.
- We believe **Data Modeling is essential** in an analytics setup and should not be overlooked.
- We think that **SQL based analytics** will win over non-SQL based analytics.

- We believe that **analytics workflow/operations** is more important than a singular focus on visualizations.

Given what we've covered over the past couple of chapters, and given what we've shown you about the landscape of BI tools in this section, you should now be able to place these biases against a larger context.

- We prefer **ELT over ETL** because we think that ELT gives you the power of more flexible data modeling practices. You may choose to skip the up-front modeling costs of the original Kimball paradigm, and only chose to do so when your reporting requirements call for it.
- We prefer using a **cloud data warehouse** because modern MPP data warehouses represent a fundamental shift in capabilities. The business model of a modern data warehouse also fits into the overall reorientation to cloud that we're seeing in the broader marketplace. On this note, we are not alone: nearly every BI tool today assumes that a cloud data warehouse sits at the center of the stack. We have adapted accordingly; you should, too.
- We believe **data modeling is essential** because data modeling at a central layer enables **organizational self-service** ... *without* the challenges of inconsistent metric definitions.
- We think that **SQL based analytics** will win over non-SQL based analytics, because the entire industry has standardized on SQL in the last five years.
- We believe that **analytics workflow/operations** are more important than a singular focus on visualizations. Much of the difficulty in business intelligence is the work needed to get data to a point of analysis. Beautiful visualizations alone will not determine the success of your department. It is more important that the business

not bottleneck on its data team in order to get the insights they need.

Wrapping Up

So, let's recap. When you're in the market for a business intelligence tool, you may categorize the tools you see in the following ways:

Sample Table of BI Tools

Name	One	Two
SQL vs Non-SQL	**Non-SQL**: Tableau, PowerBI, Sisense	**SQL:** Holistics, Looker, Mode, Redash, Metabase
Embedded Datastore vs External Datastore	**Embedded:** MicroStrategy, Tableau, PowerBI, Sisense	**External:** Holistics, Looker, Metabase, Redash,
In-memory vs In-database	**In-memory:** Tableau, MicroStrategy, Sisense, PowerBI, etc.	**In-database:** Holistics, Looker, Redash, Metabase, etc.
Modeling vs non-modeling BI tools	**Non-modeling:** Tableau, Mode, Redash	**Modeling:** Qlik, PowerBI, Looker, Holistics

1. **SQL vs Non-SQL**: Does the tool assume SQL as its primary query interface? Or does it export data out to a non-SQL data engine? Does it use cubes? The answer to this will tell you a lot about the paradigm the tool is from.
2. **Embedded Datastore vs External Datastore**: Does the tool come with an embedded datastore, or does it expect to be connected to an external data warehouse? This, too, tells you to expect a monolithic tool, or one that is designed for the modern era of powerful data warehouses.
3. **In-Memory vs In-Database**: Does the tool assume that data must be extracted out from a source system to an analyst's machine? Or does it perform all operations within a database? This can have real implications on your analysts' workflows.

4. **Assumes ETL vs Assumes ELT:** Does the tool assume a particular transformation paradigm? Is it agnostic about the data it is fed?
5. **Modeling vs Non-Modeling BI Tools:** Does the tool include a modeling component, or does it assume data will be modeled separately? Tools with modeling components take pains to give you full context; tools without do not.

There *are* other considerations when you're shopping around for a BI tool, of course, but these are usually more obvious things that you already know about: for instance, you might have considerations around pricing (contract vs per-seat), licensing (open source vs proprietary), on-premise status (either on-premise only, or cloud only, or both) and so on. We've left them out because they are obvious.

We think that the list we've laid out above is the shortest possible taxonomy that gives you the most interesting information about the tools you are evaluating. We hope they give you a way to orient yourself, the next time you look out to the landscape of business intelligence.

Alright. We're nearly done. We've taken you on a quick tour of business intelligence tool history, and we've given you a taxonomy with which to organize the field in your head. In our next, final, section, we shall explore the different kinds of data delivery you'll experience over the arc of your company's life.

The Arc of Adoption

There's one last piece that might help you navigate the business intelligence tool space. You've already seen a quick history of tool development, and we've just given you a basic taxonomy of the BI landscape. This last section will help you understand the evolution of requirements that you should see in your organization.

What do we mean by this? Well, most companies go through a very similar arc of data adoption. They do this because data usage is determined by organizational culture, and organizations go through similar cultural changes when given access to data. Understanding what that process looks like will help you understand why so many tools advertise an ability to provide 'true self-service'. It will also help you prepare for future growth.

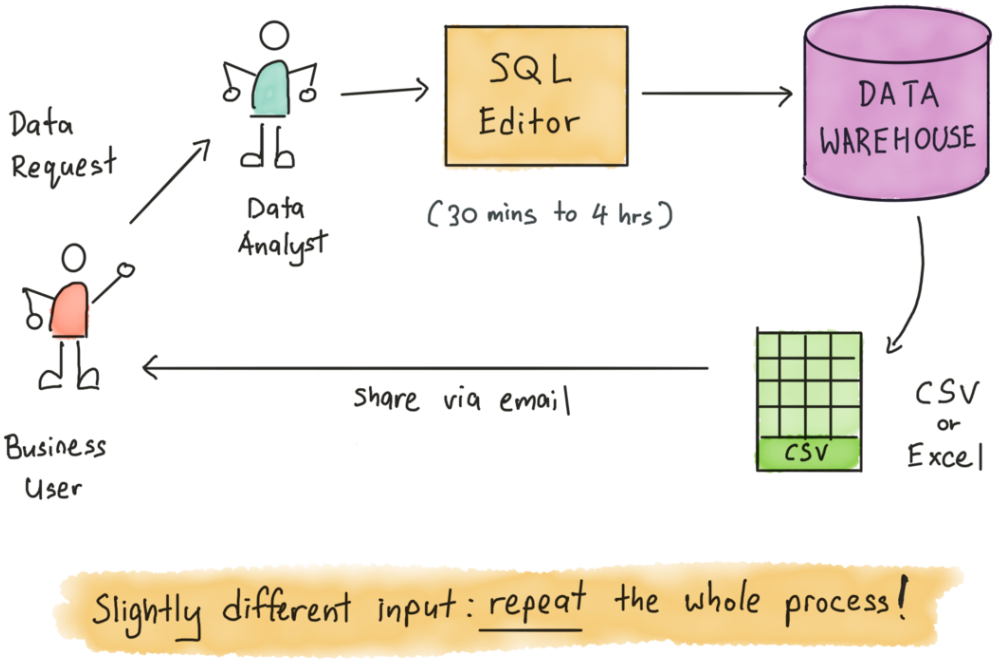
One: Ad-hoc Queries

In the beginning, the business people in your organization will have ad-hoc questions. This emerges naturally, and is as inevitable as the sun rising in the East.

How this happens is as follows: at some point in your company's life, a salesperson or marketing person or ops person will say something like "But our customers don't want that!" and someone will shoot back "How do you know?", and then everyone will turn to a data analyst to give them the answers they need.

How you serve these queries depends heavily on the tools you have available to you. If you have access to a centralized data warehouse, it is likely that you would write some ad-hoc SQL query to generate the

numbers you need. If you operate in a more 'decentralized' data environment, you would have to find the right data sources, grab the subset of data that you need, and then analyze it in whatever tool you have sitting on your machine.



Two: Static Reports and Dashboards

Eventually, as more business people catch on to the idea of getting data to bolster their arguments (and as the company expands in complexity) a data team would begin to feel overwhelmed by the sheer number of requests they receive. The head of data then reaches for an obvious next step: a reporting solution to get some of the requests off his team's back.

This, too, is inevitable. Over time, data people will learn that there is a steady, predictable cadence to some of the requests they receive. For instance, at an early-stage company that we worked with, the head of

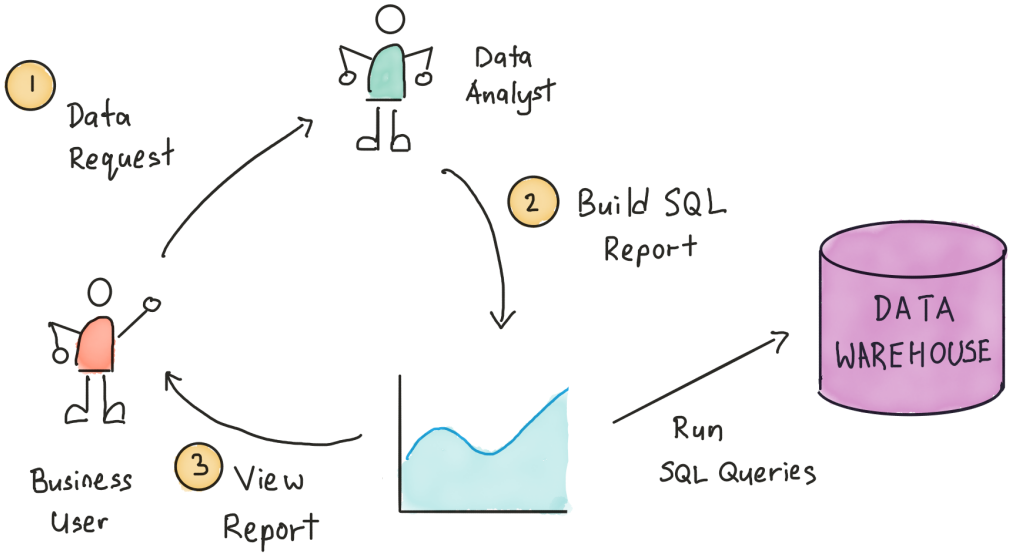
data quickly realized that the product team had a certain set of metrics they wanted to look at on a weekly basis, while the marketing team had a natural tempo of requests once every three weeks.

This head of data began to look for a BI tool to create dashboards for those predictable metrics, in order to free up his team for the more ad-hoc requests that they received from other parts of the company. Once he had created those reports, his data team immediately began to feel less overwhelmed.

"We're very happy," he told us, "The product team and the marketing team each got their own dashboard, and once we set everything up, the number of requests we got from those two teams went down. We now try and give them a new report every time they ask for something, instead of running ad-hoc queries all the time for them."

Many companies realize the importance of having good reporting functions fairly quickly. If they don't adopt a dashboarding solution, they find some *other* way of delivering predictable data to their decision-makers. For instance, a small company we know uses email notifications and Slack notifications to deliver timely metrics to their business people. The point is that the numbers reach them on a repeatable, automated basis.

Eventually, new hires and existing operators alike learn to lean on their 'dashboards'. This leads us to the next stage.



Three: Self-Service

It is perhaps ironic that more dashboard usage leads to more data-driven thinking ... which in turn leads to more ad-hoc requests! As time passes, business operators who lean on their dashboards begin to adopt more sophisticated forms of thinking. They learn to rely less on their gut to make calls like "let's target Japanese businessmen golfers in Ho Chi Minh City!", or "let's invest in fish instead of dogs!" This leads to an increase in ad-hoc, exploratory data requests.

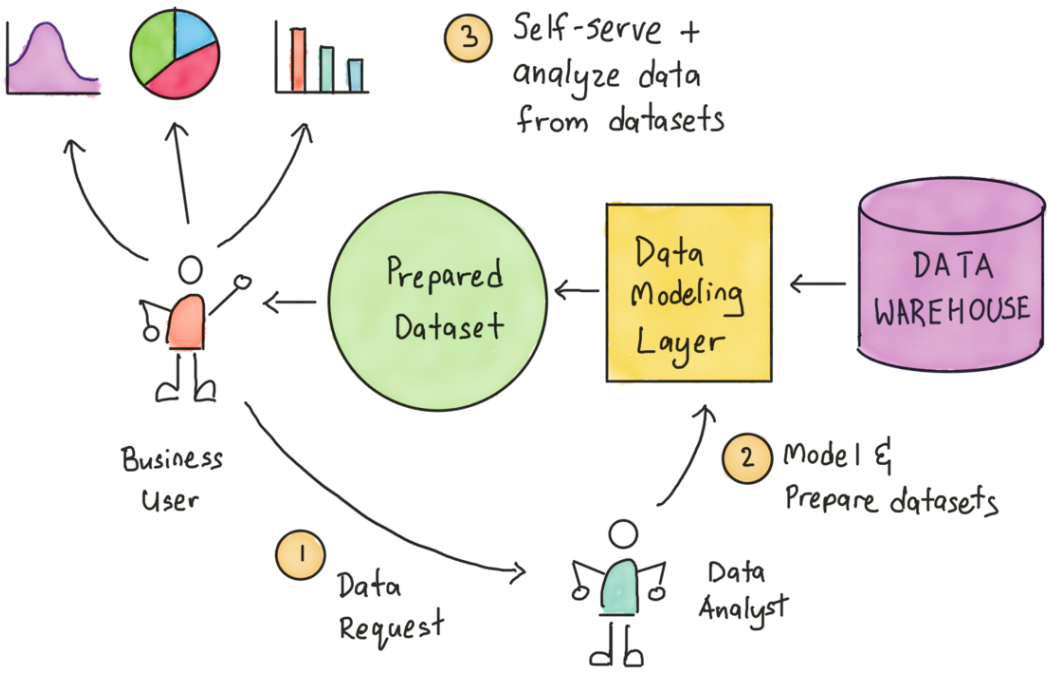
The data team finds itself overwhelmed yet again. The data lead begins to think: "if only there was some way to let our business people explore metrics on their own!"

What happens next greatly depends on the business intelligence paradigm you come from. Some companies have experimented with SQL training for their business people. "Teach them SQL," they think,

"And they will be able to get more sophisticated data from the SQL-oriented tools we have (like Redash, Mode or Metabase)."

Other companies buy into the self-service narrative sold by the 'second wave' of BI tools. This includes things like PowerBI's usage paradigm and Tableau Desktop's drag-and-drop interface. "Give them such tools," they think, "And they'll be able to help themselves to the answers they need, *without* bottlenecking on the data team."

Both approaches have problems, but the biggest problem is that they often lead to the metrics knife fight we've talked about at the beginning of this chapter. Different business users may accidentally introduce subtly different metric definitions to their analyses. These inconsistencies often lead to miscommunication, or — worse — errors of judgment at the executive level.



The Arc: Then and Now

The point we want to make here is that this arc is universal. It has happened to companies in the past, and it will continue to happen to new companies in the future.

The idea of the arc is sometimes known as a 'data maturity model'. Such models are often used by consultant-types to evaluate the data maturity of an organization's behavior. Here is an example of one, from Jon Bratseth:

Latent	Data is produced but not systematically leveraged	
	Example	**Logging**: Movie streaming events are logged.
Analysis	Data is used to inform decisions made by humans	
	Example	**Analytics**: Lists of popular movies are compiled to create curated recommendations for user segments.
Learning	Data is used to make decisions offline	
	Example	**Machine learning**: Lists of movie recommendations per user segment are automatically generated.
Acting	Automated data-driven decisions online	
	Example	**Big data serving**: Personalized movie recommendations are computed when needed by that user.

Notice how Branseth's model assumes that an organization must progress from latent, to analysis, to learning, to acting.

Another data maturity model (and one that we particularly like) is from Emily Schario of GitLab. Schario argues that all organizations go through the same, three levels of data analysis:

1. **Reporting** — This is the lowest level. As Schario puts it: when you have no answers, you never get beyond looking for facts. Example questions at this level are things like: 'how many new users visited our website last week?' and 'how many leads did we capture this month?' Some companies do not get to this level, because they lack

an organizational capability to systematically collect data in their business. Other times, they *do* collect the data, but they don't spend any time paying attention to it. Reporting is the lowest level of data analytics; if you do not collect data or if you do not have the cultural expectation of using it, you're not going to base your decisions on facts.

2. **Insights** — Insights is the next level above reporting. If reporting is about gathering facts to report on them, insights are about understanding relationships between facts. This implies combining data from multiple sources. For example: the number of new customers who cancelled their subscription this month is a reporting metric. If we combine this data with deals data in our sales CRM, however, we might learn that we have been targeting a bad subsegment of our market. This latter observation is an insight, and can lead to behavioral change among sales and product ('don't target or serve this market segment in the future; we're not a good fit for them').

3. **Predictions** — Predictions come after insights. It is at this level that you begin to see sophisticated techniques like statistical analysis and machine learning. This makes sense: after your organization increasingly understands the relationships between various metrics, you may begin to make informed business decisions to drive outcomes that you desire. A famous example here is Facebook's discovery that users who add at least seven friends in their first 10 days are the most likely to stick around. This single discovery drove an incredible amount of product decisions at Facebook — leading them, eventually, to win the social media race. Such predictive discoveries can only come after a reporting function and an insight-mining function are second-nature throughout the organization.

Such models can be useful because they show business leaders what is possible within their organizations. The fact that so many data maturity models exist tells us that there is some base truth here. It is interesting to ask why these models exist.

In other words: why does the arc occur?

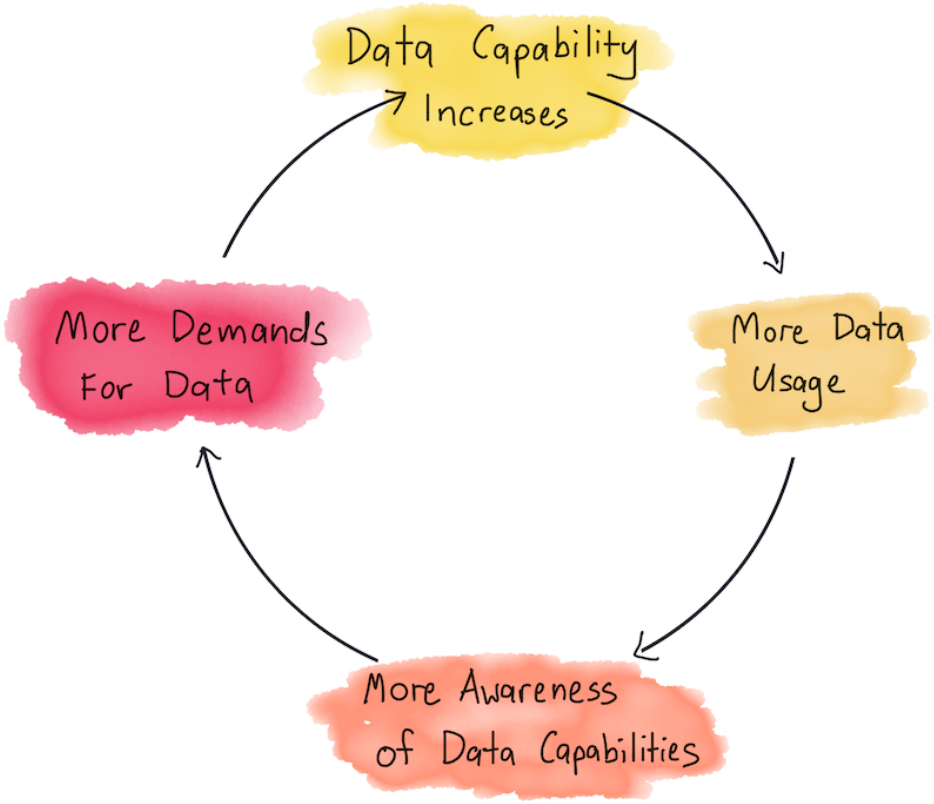
The arc occurs because *data-driven thinking is a learned organizational behavior*. It spreads slowly throughout a company's culture. As a data team member, you are given front-row seats to this phenomenon.

Most people are not data-driven by nature. They have to learn it, like they learned reading or writing. In a sufficiently large company, however, you will find certain people who are naturally data-driven in their thinking; others that seem data-driven from the beginning may have come from more data-mature organizations and therefore seek to continue the practices that they were used to. Depending on the culture of your company, these attitudes will spread in your organization (or not!)

When viewed through this lens, the data capabilities that you build out in your team will have an impact on the spread of data-driven thinking in your organization. The more data capabilities you have, the more people will be exposed to the potential of using data to advance their arguments. The more data capabilities you build up, the more leverage you give to data-driven people in your company's culture to spread their way of thinking.

As a result, the amount of work that your data team has to do increases linearly with the spread of data-driven thinking in your company!

That cycle looks something like this:



(Notice how this assumes that data-driven thinking will spread successfully throughout your org. There are other prerequisites for this to occur, of course, but we regard that as outside the scope of this book.)

The upshot is that if all goes well, your data team will find itself buried under a wave of ad-hoc requests. You will seek solutions to this problem. You will discover that dashboards and automated reports will buy you some time. But eventually, as your organization moves from reporting to insights to predictions, you would have to tackle this problem head-on.

This arc shouldn't surprise us. Spend even a small amount of time looking at industry conferences, or data thought leadership articles, or

marketing materials from vendors, and you'll find that many of these professionals are obsessed over self-service as an ultimate goal. "Listen to us!" the thought leaders cry, "We will show you a way out of this mess!" To be clear, this is an understandable desire, because data-driven decision-making so often bottlenecks on the data team. Also to be clear: a majority of companies do not succeed in this effort. True self-service is a difficult challenge.

The most important takeaway from this section is that the *arc is real.* We've talked about the challenges of scaling data infrastructure in the past, but the flip side of that discussion is the idea that you must scale your BI tools to match the data consumption patterns in your company. Keep the arc of adoption in mind; nobody really escapes from it.

Solving the Self-Service Problem Today

How are things different today? Is it possible to do better than the previous generations of BI tools?

If you've read this far in the book, you can probably guess at what we think about this: unlike 'second wave' business intelligence tools, we think that data modeling at a central data warehouse is a solution to this problem. Define your business definitions *once*, in a modeling layer, and then parcel these models out for self-service. In this way, you get all the benefits of self-service *without* the problems of ill-defined, inconsistent metric definitions.

As far as we can tell, the only business intelligence tools to adopt this approach is Looker and Holistics. We expect more tools to adapt

accordingly, however, especially if these ideas prove their worth in practice.

Will this approach win out in the end? We'd like to think so. We think that there are many advantages to this approach. However, as our intrepid data analyst Daniel has shown us, we cannot know what problems will fall out of this new paradigm. We will have to wait a couple of years to see.

Chapter 5: Conclusion

The End

So what have we shown you?

We've shown you that *all* analytical systems must do three basic things. Those three things give us a useful framework for talking about and thinking about building data infrastructure. The three things that you must do are, again:

1. You must collect, consolidate and store data in a central data warehouse.
2. You must process data: that is, transform, clean up, aggregate and model the data that has been pushed to a central data warehouse.
3. And you must present data: visualize, extract, or push data to different services or users that need them.

Within these three steps, we've taken a look at each with a fair amount of nuance:

- We've examined the rise of the modern data warehouse as being the center of most contemporary analytics stacks. We've also explored how this shift happened.
- We've shown you how ELT provides powerful benefits that stretch beyond minor operational improvements.
- We've shown you the outline of a contemporary dimensional data modeling approach, shaped by all that is possible with modern technology.
- We've demonstrated how to do that using a new breed of tools we've named "data modeling layer tools", and walked you through a

set of concepts that are common to those tools.

- Finally, we've sketched for you the shape of the entire BI tool landscape.

The Future

What do we think the future holds for data analytics?

In Chapter 4, we explained that much of the business intelligence world is confusing because competing approaches from different paradigms tend to stick around. A newcomer to the field would be overwhelmed as to which approach to take; more experienced practitioners may themselves be taken aback by the sheer amount of tools, ideas, and approaches that seem to emerge every couple of years.

We don't think this is going to change. Data analytics and business intelligence continue to play a major role in most modern businesses, and vendors are highly motivated to get new products, new ideas and new approaches off the ground. This isn't a bad thing! It's just the way it is. Our hope is that this book gives you a solid enough foundation to understand new changes in the industry as they emerge over the next decade.

In the short term, we think that many of the trends we've pointed out throughout this book will continue to proliferate. The industry's current shift to standardize around SQL will only strengthen; new BI tools will continue to adapt to the power of the modern data warehouse. We (rather biasedly, it must be said) believe that the approach we have described in this book is The Way Of The Future, and that it would eventually seep into large enterprises — that is,

whenever it is that large enterprises finally move their operations to the cloud. This may take decades, and may also never happen — because this is a bit of wishful thinking on our part. But we believe it to be true.

Now: how does this affect your career?

If you are a data analyst, one implication is that you should have passing familiarity with *all* the approaches from all three paradigms in the business intelligence world. This doesn't mean that you must master them — but it does mean that you should be aware of the alternative approaches that exist. This is because — as we've pointed out — old approaches in business intelligence stick around for a long time.

Concretely, what it looks like is the following: if you are working as a data analyst in a startup today, you may find yourself operating in a 'first wave' BI environment if you decide to move to a larger, older company tomorrow. Conversely, if you are a data analyst in an old-school data department, you may be taken by surprise if you leave and find yourself in an ELT-first paradigm at a younger company.

As a business intelligence tool vendor ourselves, we find ourselves chomping at the bit to change the industry. But if you are a data practitioner and not a vendor, it is probably a good idea to be pragmatic. Maintain awareness of all the approaches that exist in our industry. Stay abreast of new developments. That way, you won't be terribly surprised when you see a data team that does things very differently from the way you did things in the past.

We hope you enjoyed this book, and that you've learned a great deal from it. If you thought this was useful, we would be very grateful if you shared this book with the people who need it — new data professionals, startup founders looking to set up a data analytics capability for the first time, product people who just want the bare minimum to hit the ground running.

We'd also love to hear from you if you have feedback or thoughts on the book, you can:

- Send us your feedback via this form, or
- Share your private feedback to us at content@holistics.io, and
- Follow us on Twitter at @holistics_bi

Godspeed, and good luck.

holistics

Try Holistics for your company today!

Holistics helps you set up, run and maintain your analytics stack without data engineering help.

www.holistics.io →