Analytics Engineering Bootcamp (Udemy)

What is Analytics Engineering?

Previously, data was extracted and transformed before it entered the data warehouse. Now it’s loaded into the data warehouse right after being extracted. This was a shift from an ETL to an ELT (Extract, Load, Transform) approach. This allowed data analysts, BI analysts, etc. to be fully in charge of data transformation jobs. Since they’ve been already working very closely with the business, they have a very good understanding of business and analytics.

Finding someone who has the business acumen and also a very technical background is hard, but with the ELT shift, it’s easier to find people with good SQL skills who can bridge this gap. Analytics Engineers sit right between Data Engineers and Data Analysts. Their role is to handle data transformation and model the data so it is ready for use.

Analytics Engineer Tools

Ordered by importance.

1. SQL, dbt, BigQuery, Snowflake, Data Modelling Techniques (Kimball)
2. Git, GitHub, Looker, Tableau, Apache Airflow
3. Python, R, Apache Beam

What is a Database?

A database is where a collection of data is stored.

An online retail company can store data of customer names, phone numbers, payment methods, billing addresses, and other contact details. A music streaming service can store song and artist information, customer favourite track list, billing details, and more.

Most companies are using at least one form of database to store data and process data.

DBMS (Database Management System)

Usually a database is managed by DBMS. This is a software designed to store, retrieve, define, and manage data in a database.

Popular examples of a DBMS include, Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL, SQLite, etc. But if you have 250GBs of hard drive space, that’s not enough for a big database. What happens when you run out of that space? How do you scale it? Amazon RDS and Google CloudSQL give you an options to use their cloud database services so you can scale without having to worry about hardware and infrastructure maintenance.

Basically, a database is a collection of data, while a DBMS allows the user to interact with that data. Together this is called a database system.

Data is stored in a table as rows and columns. There are other types of databases that don’t store data in a table-form however.

To access data stored in a database we need to use SQL.

SQL (Structured Query Language)

We can use SQL to retrieve, manipulate, or store data within a database. SQL has been around since 1979 and there are variations of syntax between various databases, but most modern databases use ANSI (American National Standards Institute) standard SQL. It supports general commands like SELECT, CREATE, UPDATE, and INSERT.

Spreadsheets vs. Database

Spreadsheets (like Excel) allows to store data, they’re popular across the industry, are an easy way to store data and produce reports, you can write formulas, do calculations,

make visualizations, automate things using macros, and they're now cloud native. You can even get the same results sometimes by using either Excel or SQL. So what makes spreadsheets so different from a database? Well, they have different use-cases.

Spreadsheet	Database
Limit of 1,048,576 rows and 16,384 columns.	No limit to how much data you can store.
Most of the time it loads all the data, when you do any calculation.	Only loads the data you request which optimizes speed and performance.
Static (data is not updated frequently and it is updated manually.)	Dynamic (can receive data from various sources, and normalization allows data to be updated efficiently and quickly.)
Very easy to create.	Difficult.
Difficult to find information as spreadsheet grows across sheets.	Easier and faster due to SQL allowing you to write complex search queries.

SQL Databases

Relational (OLTP)

OLTP stands for Online Transactional Processing. It's used when working with transactional data, like CRM, banking systems, or retail applications. It is designed to INSERT, UPDATE, and DELETE efficiently and is required to be available 24/7.

Think of people making purchases online—this happens at all times so OLTP databases that deal with business transactions also need to be online all the time. Any downtime can have significant impact on business.

The queries are usually straightforward, such as accessing individual records, or updating customer numbers. Tables are highly normalized, and data redundancies are minimized so there is no duplicate data. There are usually multiple tables, (so say a sales table will be split up into customer, order, and item) and JOINS are used often to query results. A schema is the entity model (usually 3NF) which is usually built and planned before it is populated. It is ultimately optimized for data collection, not for aggregations.

OLTP systems are also ACID compliant.

ACID-Compliant

In OLTP systems, consistency is everything. Ensuring nothing is duplicated or incorrect is crucial for businesses. OLTP systems ensure one transaction is successful before another transaction starts.

So imagine a scenario where a bank transaction is happening. When it starts, it needs to either be successful or fail. Imagine if half of the money was sent and the other half arrived. Or the transaction was duplicated. This is handled by something called ACID.

When a database possesses four properties (atomicity, consistency, isolation, and durability) it is said to be ACID-compliant. These four properties together ensure data in the database is consistent at all times and the transaction is completed in a timely manner.

Atomicity	The transaction either happens, or does not happen at all.
Consistency	Data must follow the defined rules of the database, like null data check, unique check, or even a primary key check. When a transaction doesn't follow these rules, it will roll back to its previous state. So if a transaction fails, it goes back to what it was before it was initiated.
Isolation	All transactions happen in isolation and they don't affect one another. One transaction, shouldn't be able to access the result from another transaction which is not finished yet.

Atomicity	The transaction either happens, or does not happen at all.
Durability	Successful transaction changes are saved to the system, even if there’s a system failure.

Analytics (OLAP)

OLAP stands for Online Analytical Processing. They are optimized for running analytical queries and doing data aggregations across large amounts of data. This is used mostly for building data warehouses, and they become the building blocks of business intelligence tools.

Data here isn’t organized by rows and columns, it is organized column by column. This allows for high compression of data and for large datasets to be processed very quickly. Since the data is stored at column level, your analytical queries only need to read the relevant data instead of going row by row.

In OLAP databases, tables are usually denormalized. Compared to OLTP systems, we’ll have less tables, but the tables themselves are usually much larger and organized in terms of facts and dimensions.

Inserts and updates aren’t as important as OLTP data, because most of the time your data is a copy of your transactional data from your OLTP. You should consider denormalizing your data when using OLAP systems, which is basically putting all your tables into one big table.

OLTP vs. OLAP

OLTP systems store data in rows and columns and are operational in nature…

Name	Address	City	Post Code	Country
Canyon Grocery	2817 Milton Dr.	Albuquerque	87110	USA
QUICK-Stop	Taucherstravue 10	Cunewalde	1307	Germany
White Clover Markets	1029 - 12th Ave. S.	Seattle	98124	USA

While OLSAP systems are columnar by design and focused on BI and decision making…

Name
Canyon Grocery
QUICK-Stop
White Clover Markets

Address
2817 Milton Dr.
Taucherstravue 10
1029 - 12th Ave. S.

etc.

Overall, the differences between the two are…

	OLTP (Online Transaction Processing)	OLAP (Online Analytical Processing)
Function	Operational	BI, Decision Making, Analysis
Data Source	Current day-to-day activity.	Data from all places, OLTP, third-party systems.

	OLTP (Online Transaction Processing)	**OLAP (Online Analytical Processing)**
Data Type	Structured Data (Current)	Un-structured Data (Historical)
Query Types	Simple, Multiple Joins, Return Few Records	Complex, Less Joins, Return Large Records
Speed	Very fast transactions type.	Slow to medium, depends on complexity of query.
Database Design	Normalized (many tables), Relational	De-normalized (few tables), Star or Snowflake Schema
Storage Type	Row Based	Columnar Based
Inserts/Update/Delete	Prefer fast inserts, updates are unscheduled and frequent.	Batch jobs, update and delete are usually from source and scheduled.
Users	Operational Users	Data Scientists, Data Analysts

NoSQL Databases

NoSQL is referred to as “Not only SQL”. It’s an alternative to relational database systems. NoSQL is a non-relational database, and is able to handle various data formats, such as unstructured or semi-structured data.

Similar to how we have OLTP and OLAP as sub-types of SQL databases, there are also many sub-types of NoSQL databases, for different use cases. Some of the major types are…

- Key Value Database
- Document Database
- White Column Database
- Graph Database
- Search Engine Database

Key-Value Stores

These are very simple databases that use a simple key-value pair to store data. Values are retrieved by providing an associated key. These types of databases are suitable when speed of the data retrieval is the most important factor, and the data is not too complex.

key	**value**
123	Name, Address, Email
456-321-211	John Turner
223	Name, Address, Email, Payments

Think of key-value store as a phone directory, where key is the person’s name, and value is the phone number.

Key value stores can store various types of data, including string, sets, hashes, lists, and sorted sets. They’re often used for storing shopping carts, storing user sessions, game session management, or product recommendations.

Document Stores

These are non-relational databases that are designed to store and query JSON-like documents. Most modern APIs and web applications use and return JSON data so this type of database can feel natural for developers to work with.

Instead of storing data as rows and columns, it is stored in a JSON format. Each record is stored within a single document. You don’t need to predefine or create a schema before loading your data.

Three tables, containing ids, suppliers, and products, might be stored as a JSON document in a document store and appear like…

{
“id”: “1”,
“first_name”: “John”
“last_name”: “Turner”
“age”: “35”
}

Here there are no primary keys, we can see all order information associated with one particular user stored together.

Document stores can store catalogs, web applications, and realtime analytics because they are either use or output some sort of JSON data.

Wide Column Stores

These store data in flexible columns instead of rows. Unlike relational databases, names and formats of the columns can very across rows in the same table. This table shouldn’t be used for joining, aggregations, or even running ah-hoc queries.

Wide column stores can store real-time data, timeseries, and trading data.

Graph Database

These store and navigate relationships between data. Graph databases store relationships alongside the data in the model—they are regarded as first-class citizens. Here there is a concept of nodes and edges, where data entities are stored in nodes and relationships are stored in edges.

So imagine a social platform, where John and Mary are connected by the relationship ‘friend.’ Then each person has another relationship to something they posted, say ‘post1’ or ‘post2.’ Graph databases build recommendation engines, social networks, fraud detection, natural language processing, etc. All of these require relationship processes to be understood.

Search Engines

These are dedicated to searching for data in the form of web search. Data is stored in JSON document form and is schema-less, it uses indexes to categorize similar characteristics among data.

These are used for full-text search, time-series data, logging and analysis, and auto completion.

SQL vs. NoSQL Databases

	SQL	**NoSQL**
Volume	GB-TB	TB-PB
Velocity	Low	High
Variety	Structured	Unstructured
Consistency	Strong, ACID-Compliant	Less Rigid
Schema	Normalized, Schema Design	Denormalized, Schema Varies
Scalability	Vertically Scalable by upgrading Hardware	Horizontally Scalable by Partitioning
Data Model	Relational	Non-Relational

On-Prem vs. Cloud

On-Prem stands for on-premises, and it means you're using software or a database that is installed and running on the premises of the organization. It usually requires teams of engineers and experts managing the hardware and software and database updates. Backups are done by data warehouse experts on scheduled time and due to many moving parts, upgrades require some downtime. There's a higher cost upfront due to hardware cost and additional maintenance resource.

Cloud solutions are usually hosted by third-party vendors as IaaS (Infrastructure As A Service) and the cloud service provider takes care of cost associated with maintenance, resourcing, and bandwidth. Google, Microsoft, Amazon—they all provide cloud solutions. If anything breaks on their end, you get compensation. Companies only pay for what they use so no large expense upfront, it's usually a monthly subscription. You also don't need a team of experts to manage and maintain the infrastructure. This is also only accessible via a web browser.

What is a Data Warehouse?

A data warehouse is NOT the same as a database. It stores databases, a copy of the OLTP database, or data from third-party services and is optimized to do this for analysis and reporting purposes.

For example, imagine you want to build an analytics solution which gives your company a 360 view your customers. You can't just have data from your OLTP systems, you need to also aggregate data from various other places, such as Salesforce, data call centre data, Google Analytics, and so on. You would do all of this aggregation and data mapping here in a data warehouse.

You're going to end up with a fairly large amount of data, but data warehouses can scale pretty well. When we say data warehouse, it is usually using OLAP systems. Data warehouse technologies such as BigQuery, Redshift, or Snowflake are OLAP. Data stored in them is usually in the form of a dimensional model. And since they're usually optimized for analytical purposes, the ACID requirement is less strict.

Data Warehouse Example Use-Cases

1. Social Media Platform: Collects data related to users, groups, locations. This is stored in a data warehouse and drives insights for ad campaigns.
2. Banking: Analyzes spending patterns between accounts using data warehouses to see spending patterns and fraud detection.
3. Governments: To detect tax fraud they store and analyze tax payments within the data warheouses.

Characteristics of a Data Warehouse

Subject Oriented

- Focused on a subject, like the customer, marketing, or sales.
- Doesn't focus on operational day-to-day.
- Simple concise view around one specific subject.
- Optimized to answer analytical questions.
- Excludes unhelpful data to support decision processes.

Integrated

- Holds data from all different sources and entire organization.
- Has well defined naming conventions, measures, and attributes.
- Extracted data is uniformly transformed.

Time-Variant

- Data is organized in time-periods, such as daily, weekly, monthly, or annually.
- Allows for historical analysis.

- Contains elements of time either implicitly or explicitly.
- Once data is stored it can’t be modified or changed. You don’t want to be changing your historically stored values. If you do, without your OLTP system acknowledging these changes, it can create big problems. Also, since OLAP systems store data in a columnar style, update commands are usually very expensive.

`Non-Volatile`

- Data is permanent, doesn’t get deleted.
- Data is read-only.
- Only loading and accessing data is allowed.
- Data keeps growing and is refreshed at scheduled times.
- Does not require ACID-compliance or transaction processes.
- DELETE, UPDATE, INSERT are all omitted.

Benefits of a Data Warehouse

1. `Historical Insight`: Data is stored for all historical data, so we can run reports like what was our profit for the last ten years.
2. `Improves Quality of Data and Lineage`: Collects data from various data sources, structured and unstructured, then makes it consistent with correct naming conventions and standards.
3. `Increased Efficiency`: Imagine everytime you want to create a report you have to collect, clean, and write business logics. That’s very inefficient. Data warehouses make this data readily available.
4. `Fast Query Results`: Columnar style allows for very quick querying and frees up OLTP systems to do the day-to-day operational work.
5. `Business Intelligence Capability`: Underlying data needs to be of highest quality, data warehouse enables this by standardizing procedures and semantic layers.
6. `High Reliability and Scalability`: Most of these solutions are available on cloud so scalability is never an issue.
7. `Connects On-Prem and Cloud`: Most data warehouse technologies are built with multi-cloud in mind so you’re no longer limited by your on-premise infrastructure.
8. `Enhanced Data Security`: You can access the data warehouse from anywhere due to cloud. Most cloud solutions offer extra security features.
9. `Increase in Revenue`: This new visibility into your business allows for more money making opportunities.
10. `Competitive Advantage`: Again, knowledge is power. You can spot inefficiencies in business, improve processes, find gaps in the market.

Data Warehouse Architecture

This refers to the fundamental structure of a data warehouse system, defining what components will be part of the data warehouse built.

There are three main types of architectures…

1. `Single Tier Architecture`
   All source data is stored directly in data warehouse layer. This saves storage and redundancies, but not the best if you need multiple data streams and complex enterprise requirements. You’ll be querying your data directly from your data warehouse which can make your BI tools slow down, returning query results.
2. `Two Tier Architecture`
   Here there is separation between sources and the data warehouse. We have introduced a Data Lake and a Staging Layer, to prevent data warehouses being the main area of raw data being stored. Adding a data lake ensures that we have a single storage area for all of our source data. After this, the staging layer is there so we can do any data transformation and cleaning prior to storing our data inside the data warehouse layer

(because just because we *can* store all our data in the data warehouse layer, doesn't mean we should—it's not scalable and makes data lineage very difficult).

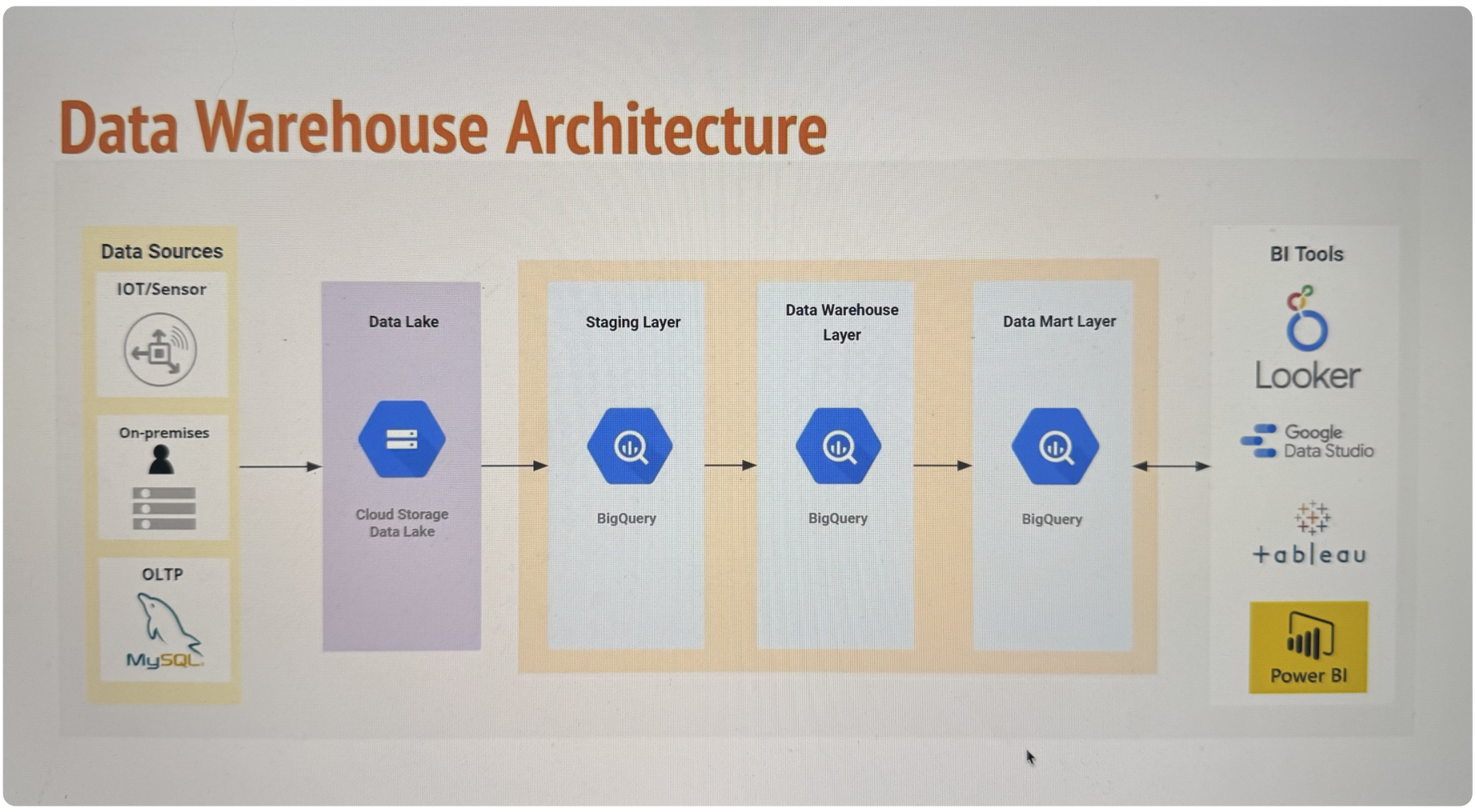
3. `Three Tier Architecture` (Most Popular)
   Here, along with Data Lake and a Staging Layer, there is also a Data Mart layer being introduced. This allows better flow of data from raw to transformed. Data Marts basically contain domain specific information, and so datasets are usually smaller, close to or less than 100GB. This makes business specific reporting possible by focusing on specific tracks such as marketing, finance, customers, etc. This also helps BI tools because they can connect to the Data Mart layer instead of the big data warehouse layer.

The Three Tier Architecture is the most popular one. The three tiers are Bottom, Middle, and Top. Using these terms makes it easy to communicate with business and IT.

`Bottom Tier`: Contains your database system, logs, third-party-data sources. All the raw data is stored in the Data Lake which is our source of truth. Then we put all our required data from the Data Lake into our data warehouse Staging Layer, where cleaning and transformation happens.

`Middle Tier`: In the Data Warehouse Layer you then apply your data modelling techniques, such as Kimball, Inman, hybrid model, data vault, etc. Within this you can have multiple layers. In the Data Mart layer you would model your data to serve a specific business purpose.

`Top Tier`: This is the front end layer where your business users will interact with the data you have modelled. This is where reports and dashboards are built and shared using Looker, Tableau, or Power BI.



Data Warehouse vs. Data Warehouse Layer

Data Warehouse is the technology specific data warehouse tool, like BigQuery or Snowflake.

The Data Warehouse Layer is a data set created within the tool BigQuery or Snowflake and conceptually termed as a layer.

Data Sources

Referring back to the three-tier data warehouse architecture, `data sources are part of the first tier`.

What is a data source?

It is any source of data you can retrieve. Data from a CRM, an Excel sheet, currency conversion services from third-party companies, transaction data recorded in an OLTP, etc. It can be structured or unstructured, but it is mostly messy data that requires a lot of cleaning.

Data Lake

Again, this is also part of the first-tier of our data warehouse architecture.

Before we can do any data cleaning or modelling or making it useful to the business or organization, we need to store all this data we got from the data sources. It needs to be scalable, secured, and easily recoverable in case of a failure. This is where the Data Lake comes in.

What is a Data Lake?

This is where you store everything. Structured, unstructured, images, logs—data lake basically becomes your single store of data. It's in its raw form, no transformation or cleaning is done here. Since storage in the cloud isn't expensive, you can afford to store everything here. You don't overwrite anything, you only add new data. This also gives you better data governance. Since all your data is now in one place, it's important that you have strong policies to ensure who can actually access what data.

Later, you will only use what is important by ingesting data into the data warehouse or any other layer.

The 5 Vs of Data

Volume: This means you have terabytes, petabytes, or even exabytes of data. Your global platform is generating real-time structured and unstructured data and you need to store it and so you need storage space. Cloud services provide storage, and data lake services is one of those where you don't need to worry about volume of data and scalability.

Velocity: This means speed of data that is being transferred, or high processing speed. Due to the volume of data, velocity also gets higher. Sometimes it is important to have less data in the form of real time as you can unlock insights faster.

Variety: Organizations have many different data sources and those data sources have many different types of data, like rows, columns, CSV files, PDF files, images, videos, etc. When we have many types of data, it can be very difficult to standardize everything and create uniformity. Data lakes are a solution for all of this. This makes it easier for data transformation tools to access and load it for analytics.

Veracity: Most of the time data we get from our sources is messy. It's hard to make sure the data is trustworthy and clean. It is important to add certain types of assertions, filtering, and cleaning early on.

Value: The end goal of every business is to unlock potential in their business through data. Bad data means bad value. It is very important you collect high quality data, and when your data is messy, you clean it to increase the quality.

Why use Data Lake?

It is unlimited, scalable and flexible, you're not limited by hardware storage or data types. You can now control who can access what data from once central place. The data is stored in its original form so it is not limited to just relational or transactional. You never need to move the data. It enhances analytics. And it allows easier access to perform AI and machine learning tasks because data is in its purest form.

Data Lake vs. Data Warehouse

	Data Lake	Data Warehouse

	Data Lake	Data Warehouse
Data Structure	Structured, Unstructured, Raw from all available sources.	Structured, processed data after all transformations and cleaning have been applied.
Users	Data Engineers, Data Scientists	BI Developers, BI Analysts
Schema-Position	Schema-on-Read	Schema-on-Write
Purpose	Location to store all your raw data.	Defined purpose for business intelligence, reporting layer.
Storage	Enables low cost storage for vast amounts of data.	Can be expensive to store large amounts of data.

Staging Layer

This sits right between the Data Lake and the Data Warehouse layer. It stores all the raw data from the data lake, and data from multiple data sources to be aggregated. Most of the data transformation, cleaning, and validation happens within this layer. A standardized naming convention is usually defined here. Usually this data ingestion from the data lake to the staging layer is done using ETL or ELT tools. Storing data in the staging layer makes data processing much easier.

Data Warehouse Layer

All the transformed data from your staging layer is stored inside your data warehouse layer. In this layer, various data modelling techniques are considered, such as dimensional model data vaults, one big table, data vault, etc.

Since the data here is already cleaned, the structure of the data is different than it was in the data source, or data staging layer. This is why we need data mapping and diagrams to document all the changes that took place in our transformation.

This is also when we would consider any kind of data masking, if our data has personal identifying information. Therefore, we need to define our access control policies here.

Various optimization strategies take place here too, for example if you complete the data model and find that some queries take a long time to execute, you might then choose to further aggregate some fact tables.

Some data warehouse technologies also provide metadata information that you can use to track your data lineage.

Data Mart Layer

This is a subset of our data warehouse. It collects data from the data warehouse but is much smaller in size. Inside here we want to organize the data for a very specific purpose, for specific departments, or business tracks.

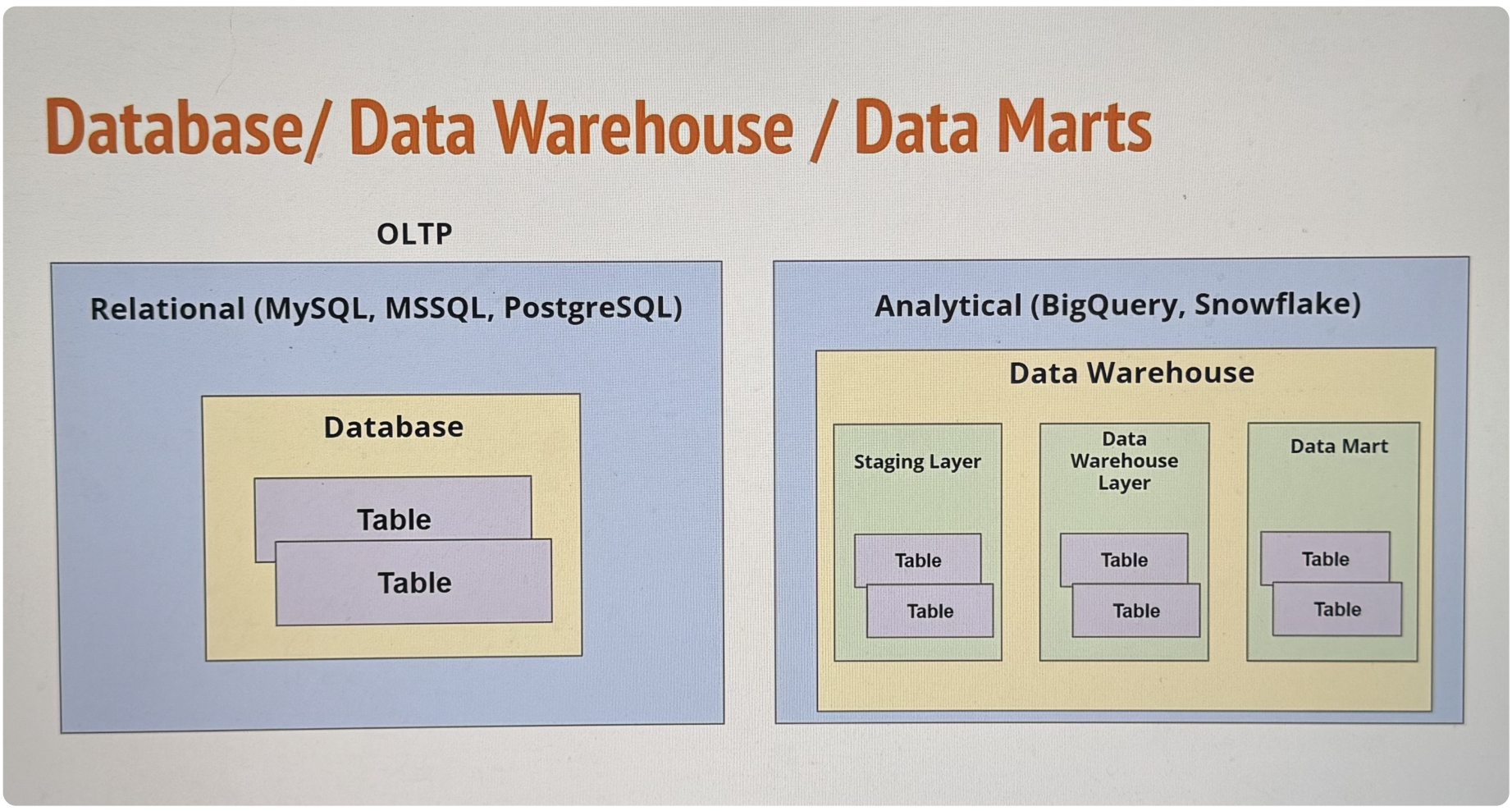
This also provides better control over security as data is organized in small manageable chunks and permissions are given only to those who are part of that department.

This also acts as a source data/layer for Business Intelligence.

Data Warehouse vs. Data Mart

	Data Warehouse	Data Mart
Subject Area	Multiple Subjects (Entire Company’s Data)	Subject Oriented (Department – Finance, Sales, Marketing, etc.)
Sources	Collects from multiple places including OLTP	Mainly derived from Data Warehouse
Granularity	Lowest level of granularity.	High level of granularity (aggregated or summarized).

	Data Warehouse	Data Mart
Focus	Data integration and decision making process.	Strategic decision making, business intelligence.
Size	Optimized for holding large volumes of data (GB/TB+)	Optimized for analysis and data access (<100GB)
Implementation Time	Months to Years	Few Months
Access	Tightly controlled with various level of permissions at organization level.	Promotes ease of access, departmental access.



What is Business Intelligence?

Referring back to the three-tier data warehouse architecture, `BI is at the (or rather THE) top tier`.

`Business Intelligence is a combination of tools, technologies, or strategies, used by companies or enterprises to enable insights and analytics of current and historical data.`

What we build inside a data warehouse needs to be consumed by the business in one form or the other, and one of those is in the form of visualizations, KPI reports, or even ML model results. Business users can use this information to unlock insights and improve data driven decision making.

BI as a field is much more complex than just producing reports. For example, analyzing customer behaviour, understanding customer 360, tracking performance of marketing campaigns, making business predictions for next quarter, these are all a form of business intelligence.

Business Intelligence Tools

BI tools such as Looker, Tableau, PowerBI all sit on top of your data warehouse, and can sometimes connect directly to other systems too.

Being able to actually tell stories using these visualization tools is very important, because nobody wants to stare at a boring dashboard. People want to look at something, and within a few minutes they want to understand the story that the data is telling.

Looker

Acquired by Google, it is the first API data platform that connects to your database. It’s also cloud native so you don’t need to do any performance tuning, you only focus on developing your Looker data models.

Their main selling point is their propriety data modelling language called LookML. With LookML you don't need to write SQL anymore. LookML in combination with modelling methods, allow you to create a streamlined definition across the organization, giving you a single source of truth for KPIs.

Looker also has strong data governance features, and gives true self-service capabilities because of the separation between the modelling layer and UI layer. It also has git source control natively built in.

Data Studio (Easiest BI Tool!)

Direct integration from the Google ecosystem. It's very popular among teams who just want a simple and easy way to build reports and dashboards. It can connect to multiple sources and can build fairly complex and effective charts.

Because it's cloud native it also offers real time dashboard collaboration. One of the main advantages of Data Studio is its simple, native integration with Google's BigQuery, allowing analysts to build reports very quickly.

ETL — ELT Introduction

Referring back to the three-tier data warehouse architecture, when we say data is being loaded, we mean that usually, a data engineering team is responsible for ingesting data from data source, to data lake, and then into the staging layer, ensuring all the batch and streaming requirements are taken into consideration. They would use various data pipeline and message queue services, such as Kafka, Spark, Beam, etc.

Once this data is loaded into the staging layer, depending on the team structure, it is the job of the analytics engineer to continue developing a data warehouse model using SQL and data transformation tools such as dbt. Sometimes, you will need to use Python to automate some processes.

Once the data model is available, it is up to the BI analysts and visualization experts to take it to the next level by enabling insights within the organization. ETL or ELT is basically the flow of these tasks.

Extract, Transform, Load (ETL)

ETL is a mix of techniques and tools you use for data ingestion and transformation. Regardless of whether you're doing ETL or ELT, you'd be following these three steps, just in a slightly different order.

It connects to various sources to gather data, extract and do transformations, and then load transformed data into a data warehouse.

Extract

This is the first step in both ETL or ELT. It pulls data from original data sources, such as structured databases or unstructured data files, web services, or APIs. Extracted data usually goes to either a persistent, or temporary staging area where all the data is processed and stored. At this stage, frequency of extraction is also determined, as well as if the data is batch or streaming data. Depending on that, we will need to choose the right tool for the job.

We also need to run a data validation check here, to check format, null issues, new columns, etc.

Transform

This is the process of changing data by either enriching or cleansing the data. There are usually multiple steps involved here.

1. Cleansing: Solving and cleaning out any inconsistencies.
2. Standardization: Applying formatting rules.

3. `Deduplication`: Removing duplicates and redundancies.
4. `Verification`: Flagging anomalies and removing unusable data.
5. `Sorting`: Organizing by type or transposing rows and columns.
6. `Other Tasks`: Any other task/rules you want to apply to improve data quality, like using lookups, or splitting a column into multiple columns.

Load

This is when transformed and cleaned data is then loaded into a data warehouse to be consumed by business intelligence tools.

During the load phase, some tools allow you to do data quality checks so that we can make sure that we're not loading wrong data. There is also usually a recovery mechanism to handle load failures.

There are two types of loading.

1. `Full Refresh Load`: Where all the content from the table is removed and replaced with fresh data.
2. `Incremental Load`: Where data load is scheduled by tools or Cron jobs to be loaded at specific intervals.

So say a missing record gets accidentally deleted. We would do a full data refresh that repopulates the table and puts the missing record back.

And if we realize there is new data in our source system that needs to be added to our target system, we can use an incremental data load, which will update only the records for yesterday.

ETL/ELT Tools

Some popular ETL/ELT tools include Matillion, xplenty, Microsoft SQL Server, Talend, Apache Airflow, Alteryx, Fivetran, and Stitch.

ETL in Data Warehouse Architecture

So say we have our source data from multiple places, and all that data is being extracted and placed in the Staging Area where it needs to be transformed. Various transformations then happen here, such as removing duplications, data cleaning, or data integration.

In the ETL workflow, extracting and transformation is mostly done by data engineers and to some extent analytics engineers. But once the data is loaded, it will be the job of analytics engineers to do data modelling work and make that data available for various purposes, such as for BI reporting. A Business Intelligence Analyst will use that model data, connect it to a BI tool, and unlock business insights.

Advantages and Disadvantages of ETL

Advantages	Disadvantages
Allows you to perform complex transformations before loading data. This makes the ETL method easier because you pick and choose which data to transform and load.	ETL processes usually require the heavy involvement of the Data Engineering team. They also need to be involved to a certain extent on the transformation side, due to complex flow of extraction and transformation.
Loads only important or required data for analytics and reporting, while unnecessary data stays within the source system, reducing storage cost too.	Since ETL only extracts necessary data, this puts more pressure on the engineering team, since the BI team has limited data and is waiting on them or need to go back and forth requesting access to more data.
Allows for speedy, efficient analysis, since your transformed data is landing very quickly and you can make it available fast for analytics and BI.	Difficult to achieve a full Self-Service BI solution, hence it gives less agility.
Removes or encrypts data before putting it into data warehouse. Since it's removed before being	Due to the complexity of the pipeline, it requires continuous maintenance to ensure

Advantages	Disadvantages
stored, there is no log information for systems admins to access. This helps organizations comply with security protocols such as GDRP and HIPAA.	everything is up to date and workflows are well defined and working as intended.

ELT in Data Warehouse Architecture

Compared to the ETL approach, here as soon as data is extracted it is loaded into Data Lake. All of the source data is stored here and we can use tools such as Airflow, Fivetran, and Stitch to do our ELT.

From the data lake, it is then loaded into the staging layer. Until this point, only the data engineering team needs to be involved in setting up the solution. Once that data is available in the staging layer, analytic engineers or BI developers can take over and start doing transformation and data modelling work.

Depending on modelling technique, data is made available by data marts or other options where BI analysts connect BI tools to these layers, and start building business reports and unlocking insights.

Advantages and Disadvantages of ELT

Advantages	Disadvantages
More flexible than ETL approach and gives you the ability to store any form of data.	ELT approach requires you to store all your data as soon as it is extracted, so you’re going to need more storage space to handle this volume of data (which is less of a concern if you’re on cloud).
Since complex transformation no longer happens after extraction, it empowers developers to perform better optimizations to enable building reports faster.	If the ELT systems are not designed correctly, it can be slow due to multiple steps involved and additional persisting of the data during these steps.
Since ELT approach loads all the data into your data lake, there is no waiting involved, and analytic engineers can determine what data they need to transform.	Can be low on compliance if how to handle private information data is not planned properly during the design process. There need to be strong data governance policies put in place to control who sees what.
Architecture is also future proof due to data lake handling most of the heavy data volume and scalability.	ELT is kind of new technology because of the recent explosion in cloud solution in offerings, which means it can be harder to find resources and experts who are well equipped.
We also no longer require Data Engineers to set up complex pipelines, instead they can focus on data solutions, and analytic engineers can focus on writing transformations.	

ETL vs. ELT

	ETL	ELT
Volume	Good for small to medium data volumes.	Good for large data volumes as we don’t need data processing.
Load Time	Slower than ETL due to transformation before loading to DWH.	Fast since transformation happens later.
Transformation	Outside of warehouse, usually within a staging area, also meaning extra time is required here.	Within warehouse, data size and volume is not an issue.
Analysis Time	Faster analysis due to data being quickly available to analysts.	Slower than ETL since you have to wait for all the data to be loaded.
Data Type	Prefers structured data because transformation happens before load and if there’s unstructured data it will take longer.	Structured and unstructured data are fine since pressure of transformation is pushed to later.

	ETL	ELT
Complexity	Can be more compute intense as transformations happen in memory which can introduce complexity.	Since transformation happens in the warehouse, you use SQL and no additional requirement.
Cost	High	Low
Availability	Only required for reporting and analysis.	Everything can be accessed from the Data Lake.
Maintenance	High maintenance due to on-prem solutions, but can be solved by going hybrid approach.	Much cheaper as you can make use of cloud services.
Data Governance and Security	Removes PII before it's loaded, so stronger security.	Option to remove PII before load, but it requires more work so is usually removed after load.
Implementation	Easier to find experts, ETL tools are matured.	Difficult to find experts, ELT tools are evolving.
Usability	Developers (requires extra steps for Business Users)	Developers to Business Users

Data Modelling and ERD Notation

What is a Data Model?

A data model is basically the foundation to building any data warehouse and business intelligence solutions. They act as a blueprint you can use or follow to understand relationships between different tables.

It can also be used to build different levels of data models such as conceptual, logical, or physical. It defines how the logical structure of the database is modelled, specifies business requirements and data structure, and provides a visual way of communicating with the business.

A data model becomes a main communication method between the business and the development team due to its visual approach.

What is Data Modelling?

It's the process itself. It's the structured approach to developing a data model by identifying and analyzing all the requirements, working closely with the business to discover and define the requirements using various modelling tools and techniques, to create a well designed data model.

Identify Business Requirements

You would start by identifying business requirements. What are they trying to achieve? Why are they trying to build a database or data warehouse? What problems are they trying to solve?

Collaborate with Business

Then you will work with the business on defining data models through a series of interviews, meetings, and reviewing existing processes to find gaps and improvements that we could bring in through the new system.

Define ERD Notation and Data Model Type

During this process you will also work with business analysts, architects, and developers to identify what ERD notation style to use, like Information Engineering, Chen's Style, Barker, UML. You'd also define what data model types will be used, like conceptual, logical, or physical.

Define Data Modelling Methodology

This again depends on what we're trying to build. If it's a database system, we are likely to go with a relational model known as an entity relational mode, or if we are building a data warehouse then we could go with a dimensional model.

`Develop and Deploy`
This is the last step in the process.

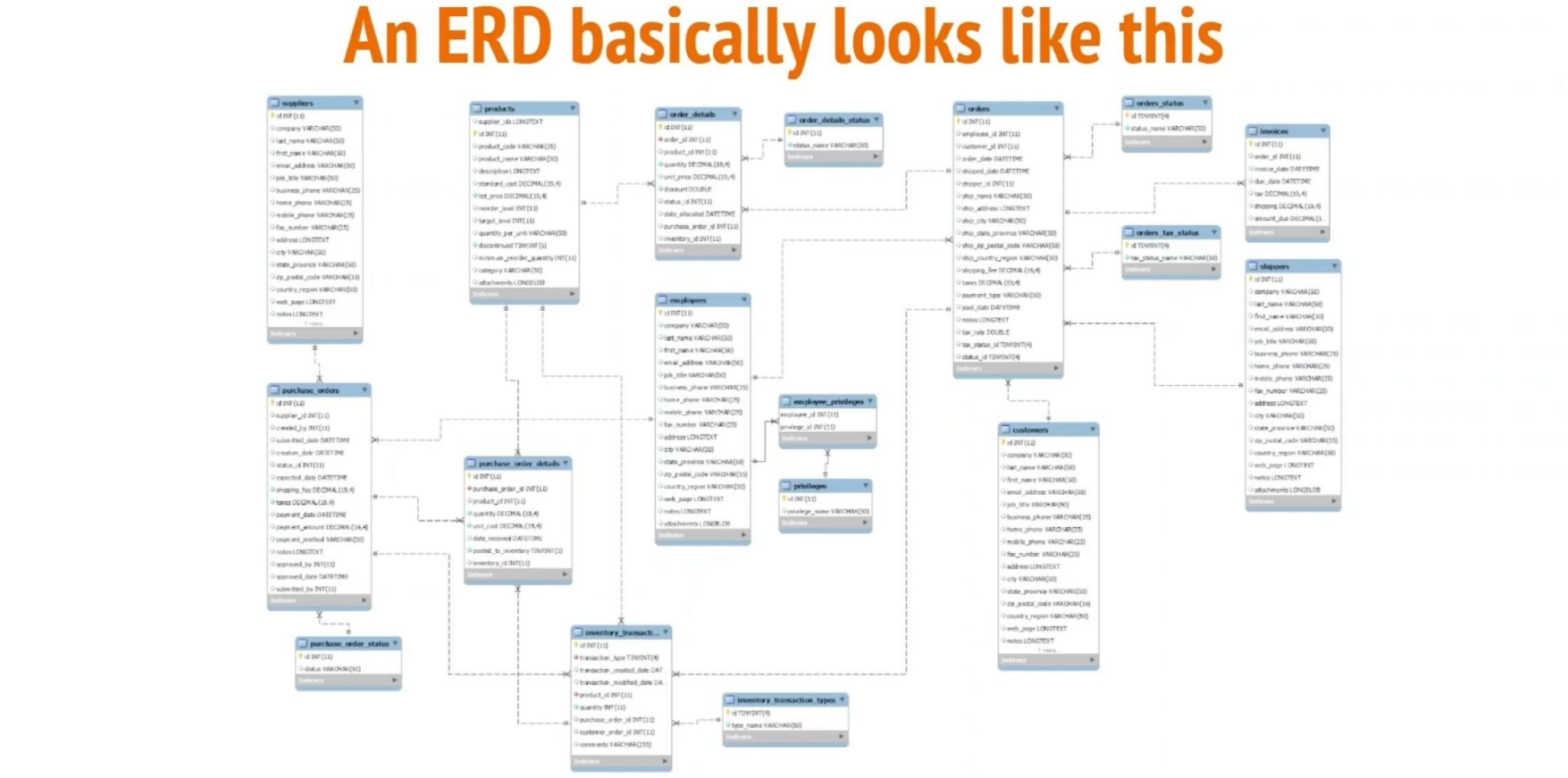
ERD Notation

ERD notation is a part of the data modelling process, it creates a visual representation of your entities and the attributes. `It essentially provides a preview of how all your tables should connect, what fields are going to be on each table, and puts forth your conceptual ideas in a readable format.`

Usually database or data warehouse implementations can be very complex, and without a visual representation of things, things can go wrong. That's why ER diagrams are used as a blueprint.

ERD stands for Entity Relationship Diagram. ERD is made up of three components…

1. `Entity`: the object from where data is collected and must have attributes.
2. `Attributes`: entity properties that can be descriptors or identifiers.
3. `Relationship`: the connection type between entities.



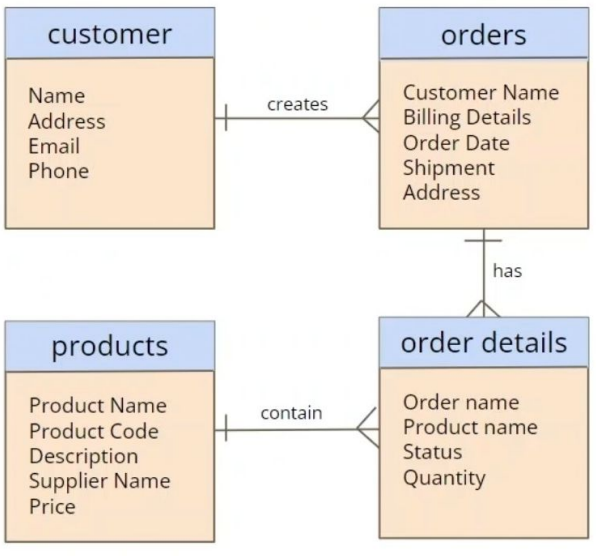
ERD Notation Styles

There are a variety of notation styles: IE, Chen, Barker, UML. In the database world, we will mostly see Chen and IE style notations.

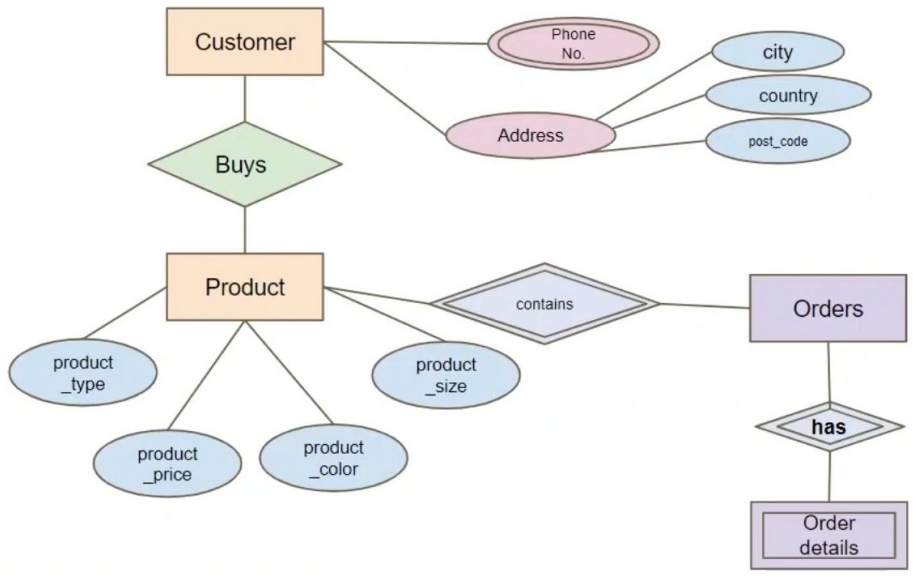
Information Engineering (IE) Notation Style	Chen Notation Style
Most widely used.	Not widely used anymore.
Uses lines with graphical symbols to indicate the relationship.	Uses different shapes and text values to explain the relationships.
Uses crow's foot notation to represent cardinality.	

ERD Notation Styles

IE Notation Style



Chen Notation Style



Entity

An easy way to identify an entity is to think about real world objects, something that exists logically, physically, or conceptually, such as a person, customer, employee, product, transaction, or account.

Both IE and Chen represent entities with a rectangle. However only the Chen style has a representation for a weak entity, which is a double rectangle.

A weak entity means it cannot exist by itself. So for example, ‘order details’ is a weak entity because the details of that order don’t exist unless that order is placed successfully, so without that, order details can’t exist.

Attributes

These are properties that describe an entity, so for example for a customer it’ll be name, email, address.

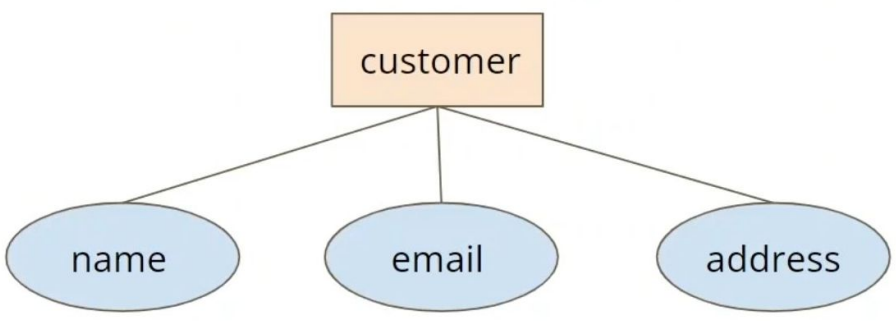
IE notation represents attributes by listing them within the rectangle box, whereas Chen style puts each attribute into an oval, stemming from the entity they are related to.

Attributes

- **Attributes** are properties that describe an entity (for customer it’ll be name, email, address etc)

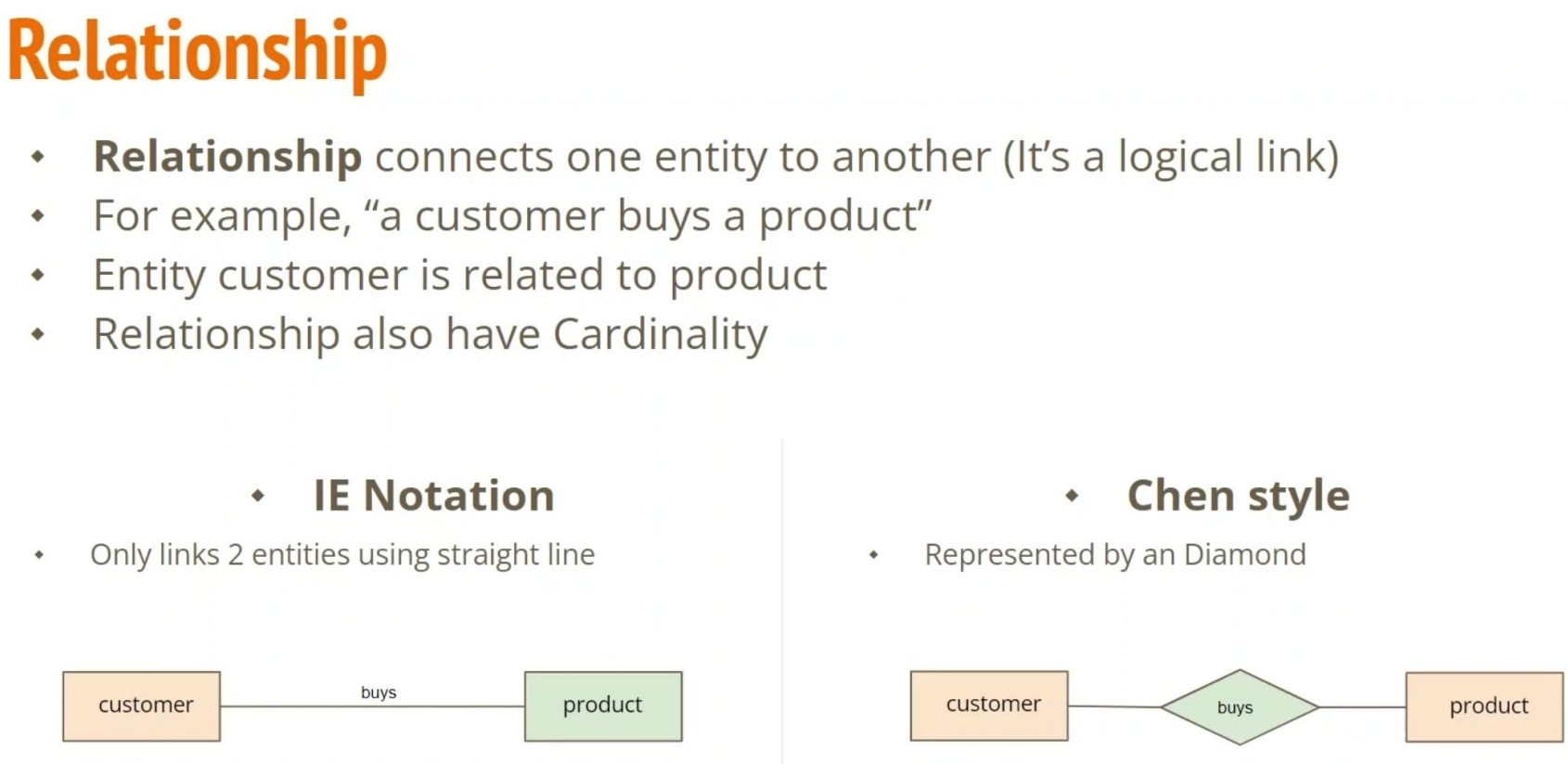
- **IE Notation**
  - Attributes listed within the rectangle box

- **Chen style**
  - Represented by an oval shape
  - Specific type of attributes also exists: multivalue, derived, unique (key), partial etc.



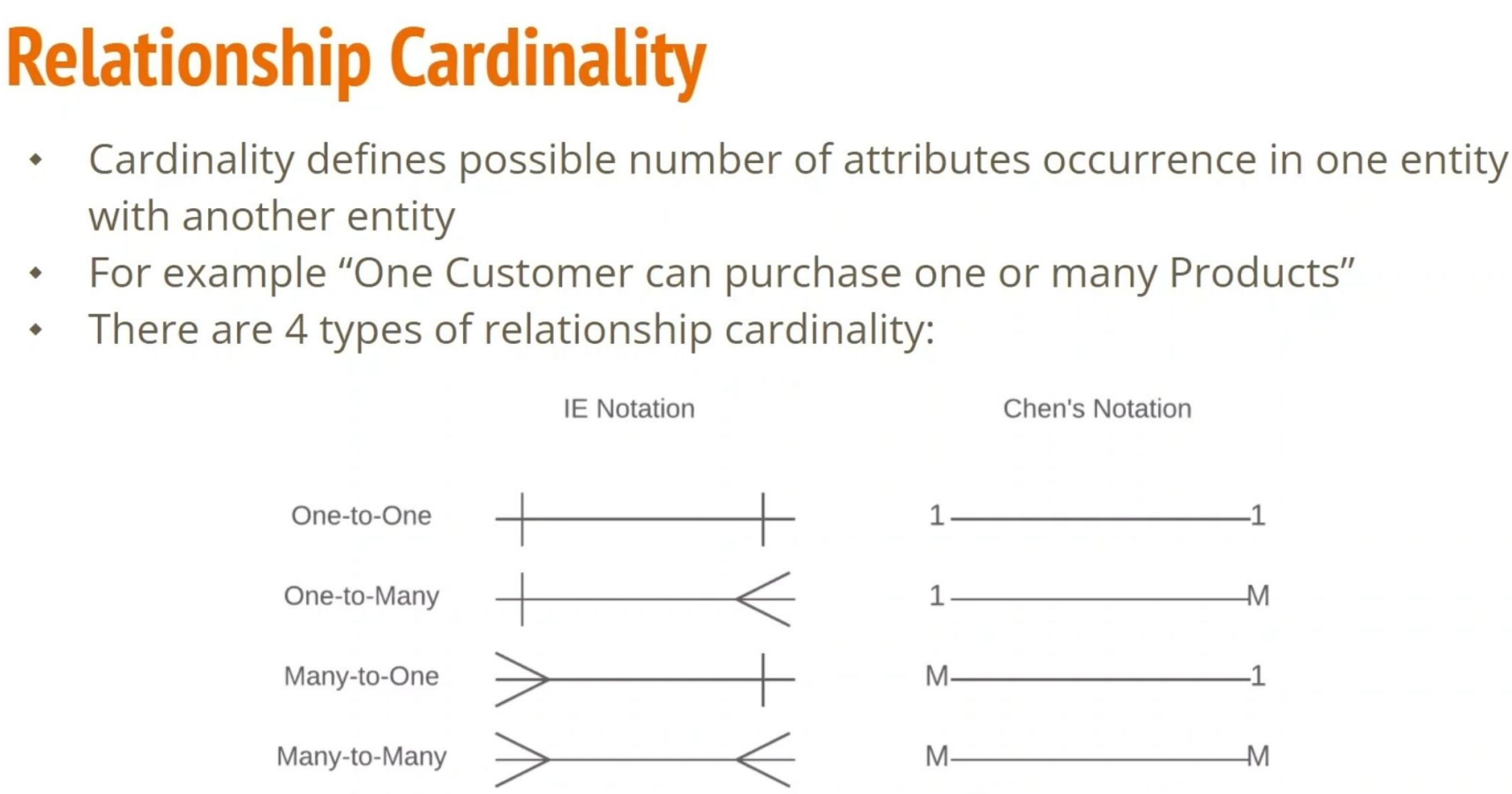
Relationship

Relationships connect one entity to another, it’s a logical link. For example, a customer buys a product, so the entity customer is related to product.



Cardinality

Relationships also have cardinality. Cardinality defines the possible number of attributes occurrence in one entity with another entity. For example, one customer can purchase one or many products.

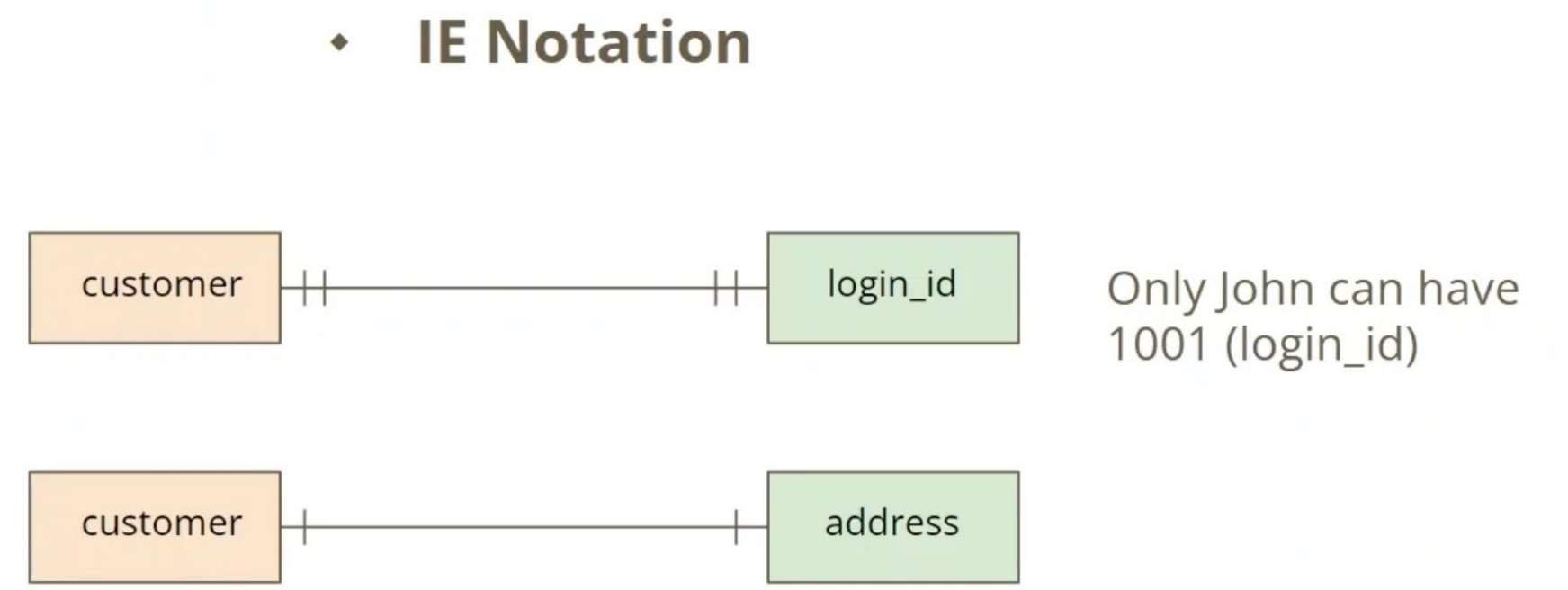


There are four types of relationship cardinality. IE notation adopts crow’s feet notation, using a single line to define a one-to-one relationship, and a crow’s foot to indicate many relationship cardinality.

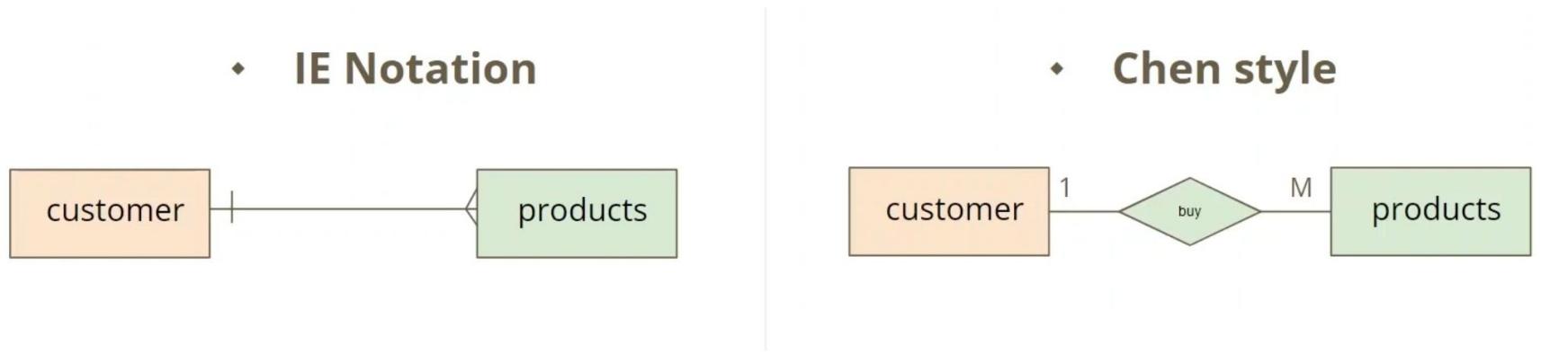
One-to-One: This means only a single instance of an entity is associated with a second entity. For example, a customer can only have one living address at a time. (But addresses

can be reassigned.)

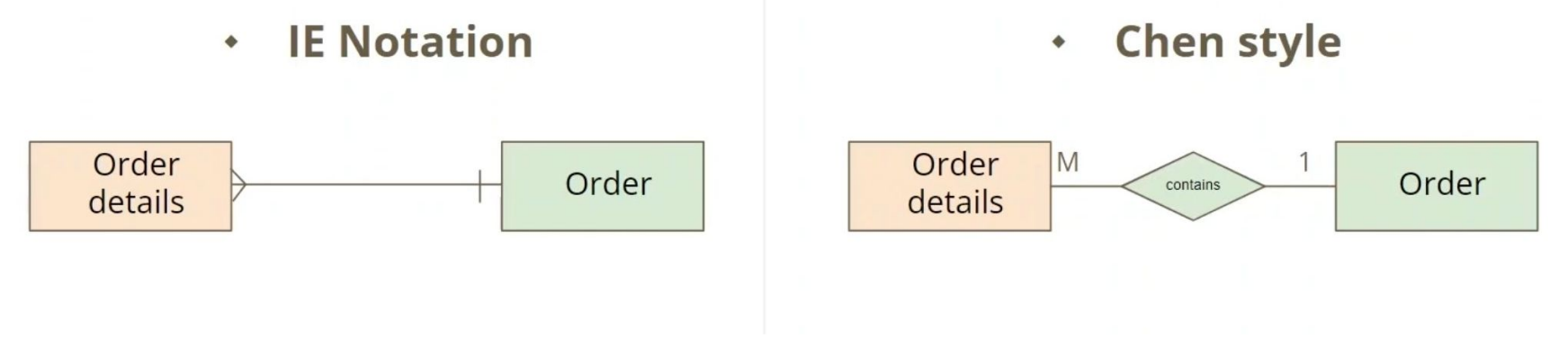
One-and-Only-One: Represented with double vertical bars on each side. For example, a customer can have one, and only one login_id.



One-to-Many: An entity can be associated with multiple instances of a second entity, but the second entity can only be associated with at least one entity. For example, a customer can buy multiple products.

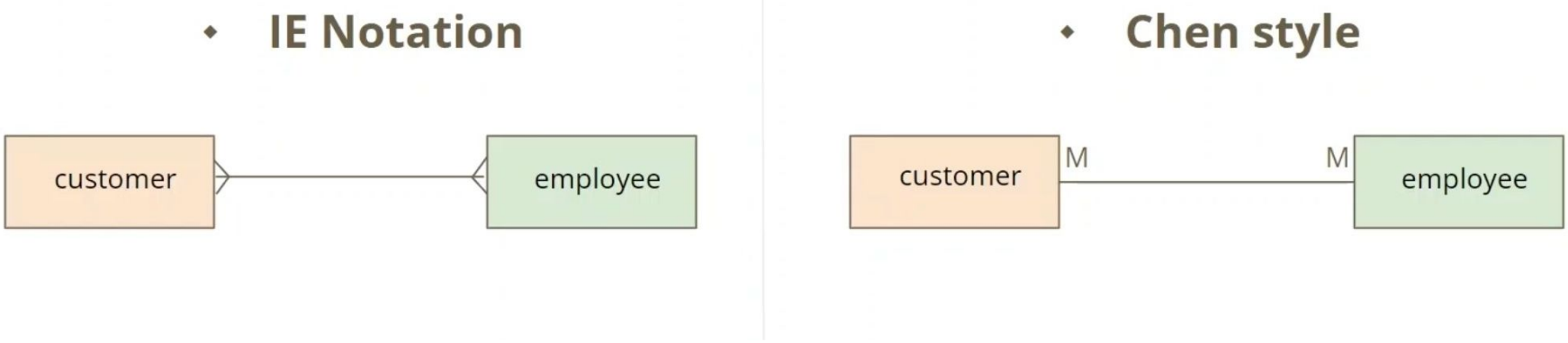


Many-to-One: More than one instance from an entity is linked or can be associated with one instance of second entity. For example, one order can have many order details.



Many-to-Many: This is where more than one instance of an entity and more than one instance of another entity can be linked with a relationship. For example, many customers

can be an employee, while any of the employees can also be customers.



Steps to Create an ERD

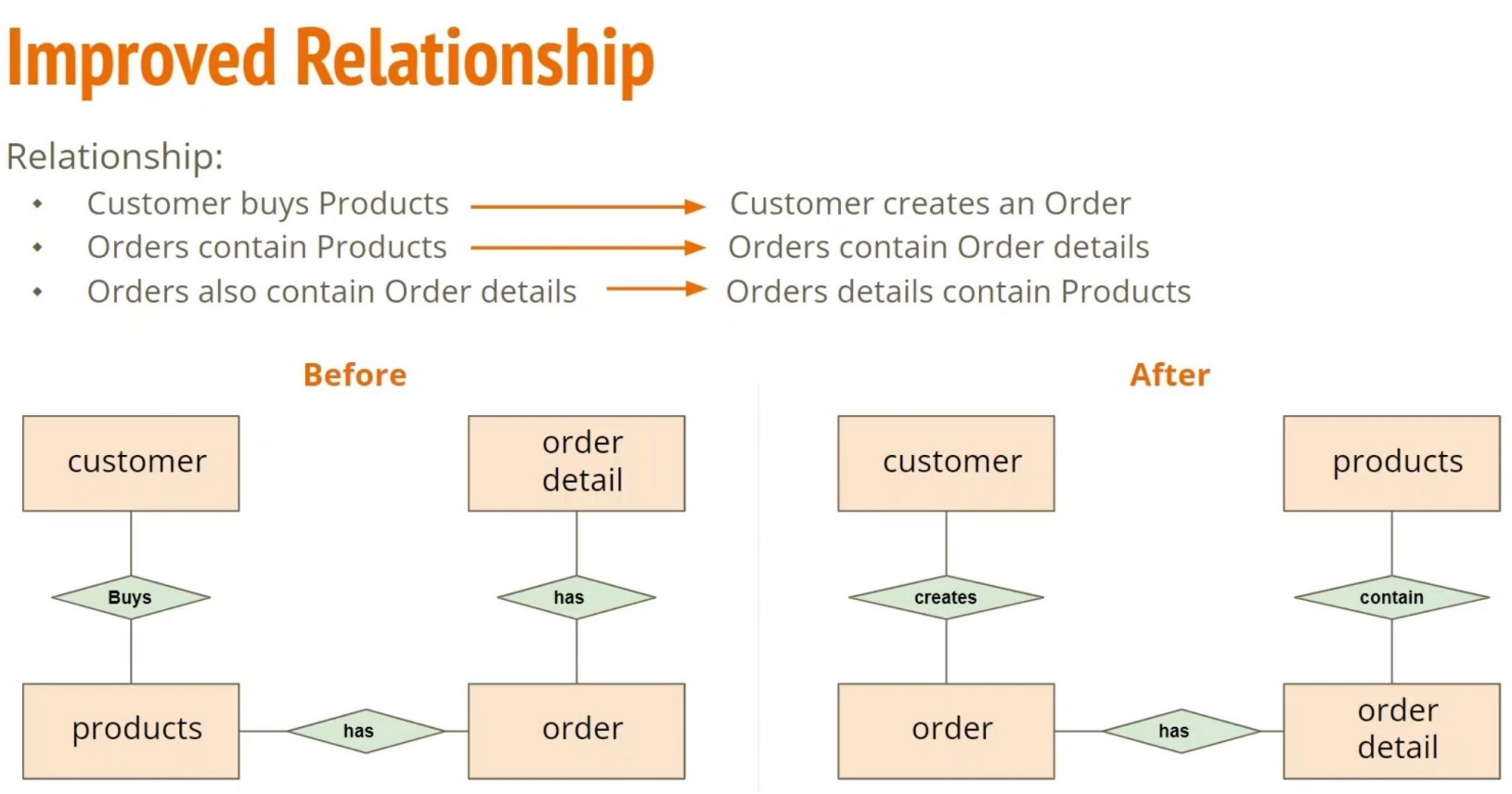
1. Identify Entity
2. Identify Relationship
3. Identify Cardinality
4. Identify Attributes
5. ERC

An easy way to remember these steps is `ERCA`.

ERD (Chen's Style)

Example: *On our e-commerce site, a customer can buy one or more products. Each product has multiple attributes such as product name and types. Once customers have placed a successful order we use order details associated with products to deliver it to the customer delivery address. At this stage customer details such as name, address, email, and phone numbers are also recorded.*

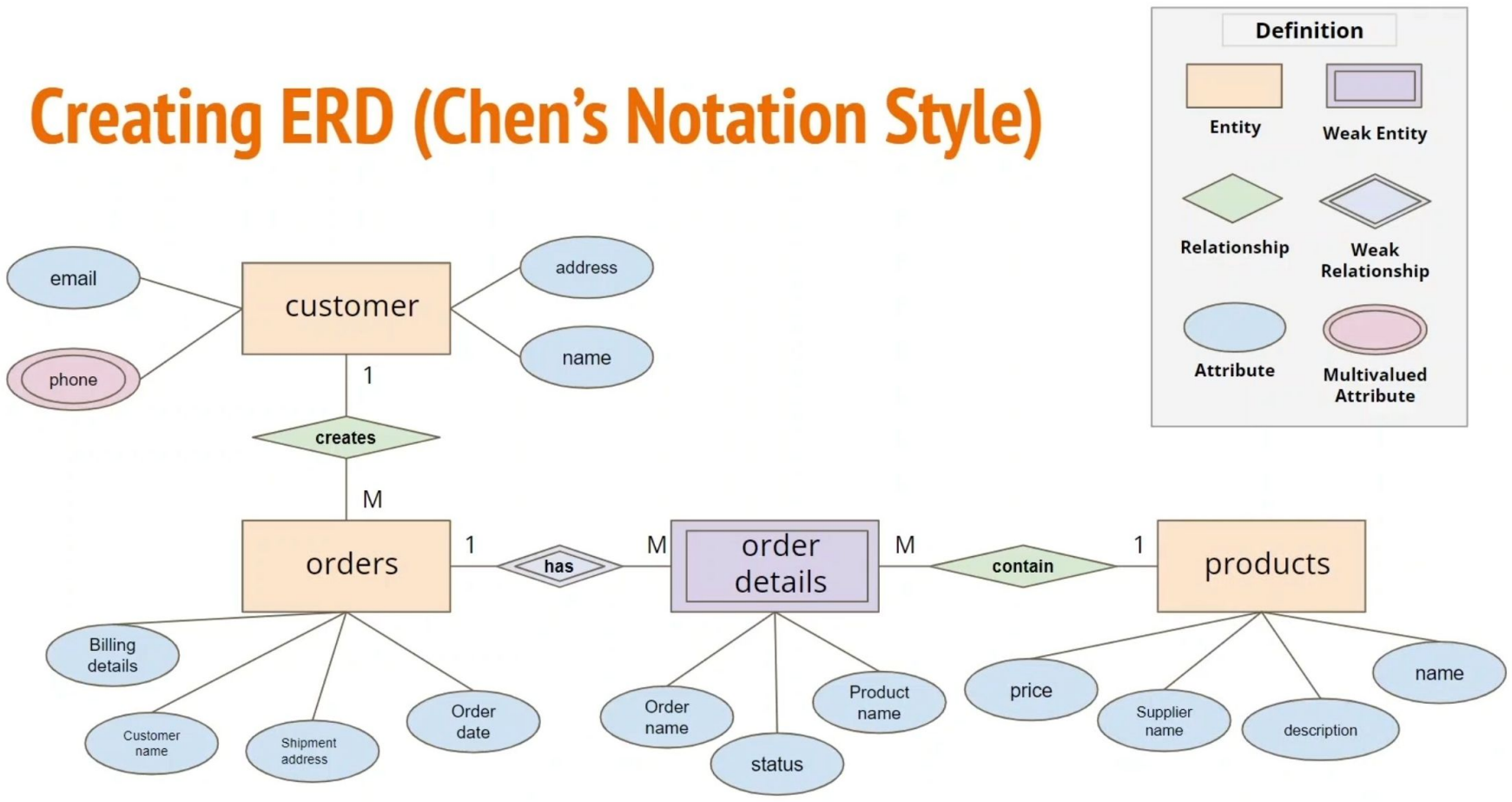
1. `Identify Entity`
   The entities are Customer, Products, Order, and Order Details.
2. `Identify Relationship`
   - Customer buys Products
   - Orders contain Products
   - Orders also contain Order Details



3. `Identify Cardinality`
   - One Customer can buy multiple Products
   - One Order can contain multiple Products
   - One Product can contain multiple Order Details
4. `Identify Attributes`
   To get this we have to ask the business for more details. At this point we'd learn that…

- A customer has an address, name, email, and phone number.
- An order has billing details, customer name, shipment address, and order date.
- Order details contain the order name, status, and product name.
- Products contain supplier name, name, description, and price.

5. Creating ERD (Chen’s Notation Style)



Weak Relationship and Weak Entity

Remember, a weak entity relies on the existence of another entity. It can’t be identified by its own attributes. It doesn’t have a primary key and cannot exist without its parent entity.

For example, in our scenario, if an order doesn’t exist, then order details also cannot exist.

ERD (Information Engineering Style)

Why do we want to convert the ERD diagram we built using Chen’s style to IE?

Pro and Con of Chen’s Notation

Chen’s notation is easier to draw simple ERs, easier to understand, and good for teaching database concepts, but it makes it too busy to read, takes up way too much space, and provides too musch information for conceptual level.

Pro and Con of IE Notation

IE notation is most widely used, easy to understand, works well for designing physical models, and uses crow’s foot notation for cardinality, but it has no strong set of standardization, and variation of representations can be confusing.

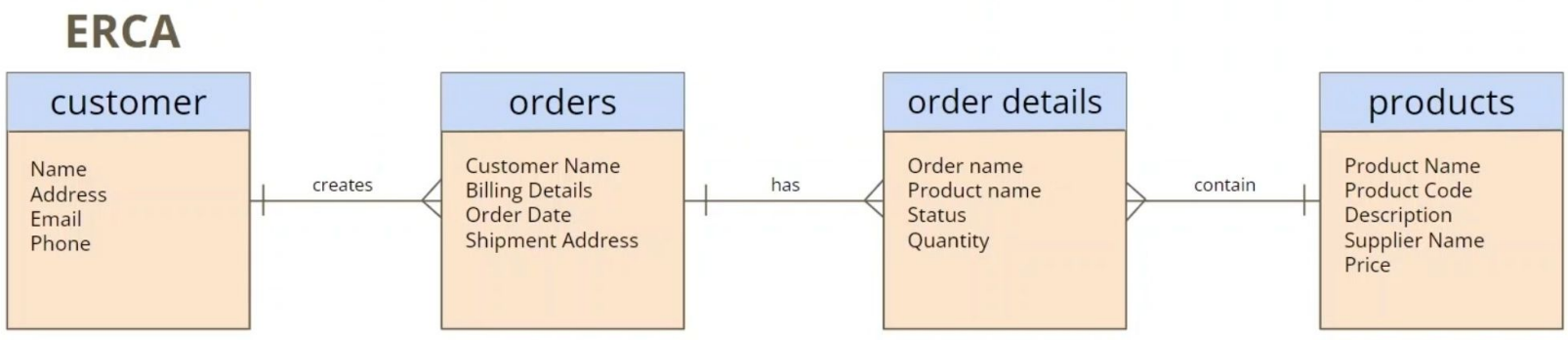
ERD (Information Engineering style)

Relationship:

- Customer creates an Order
- Orders contain Order details
- Orders details contain Products

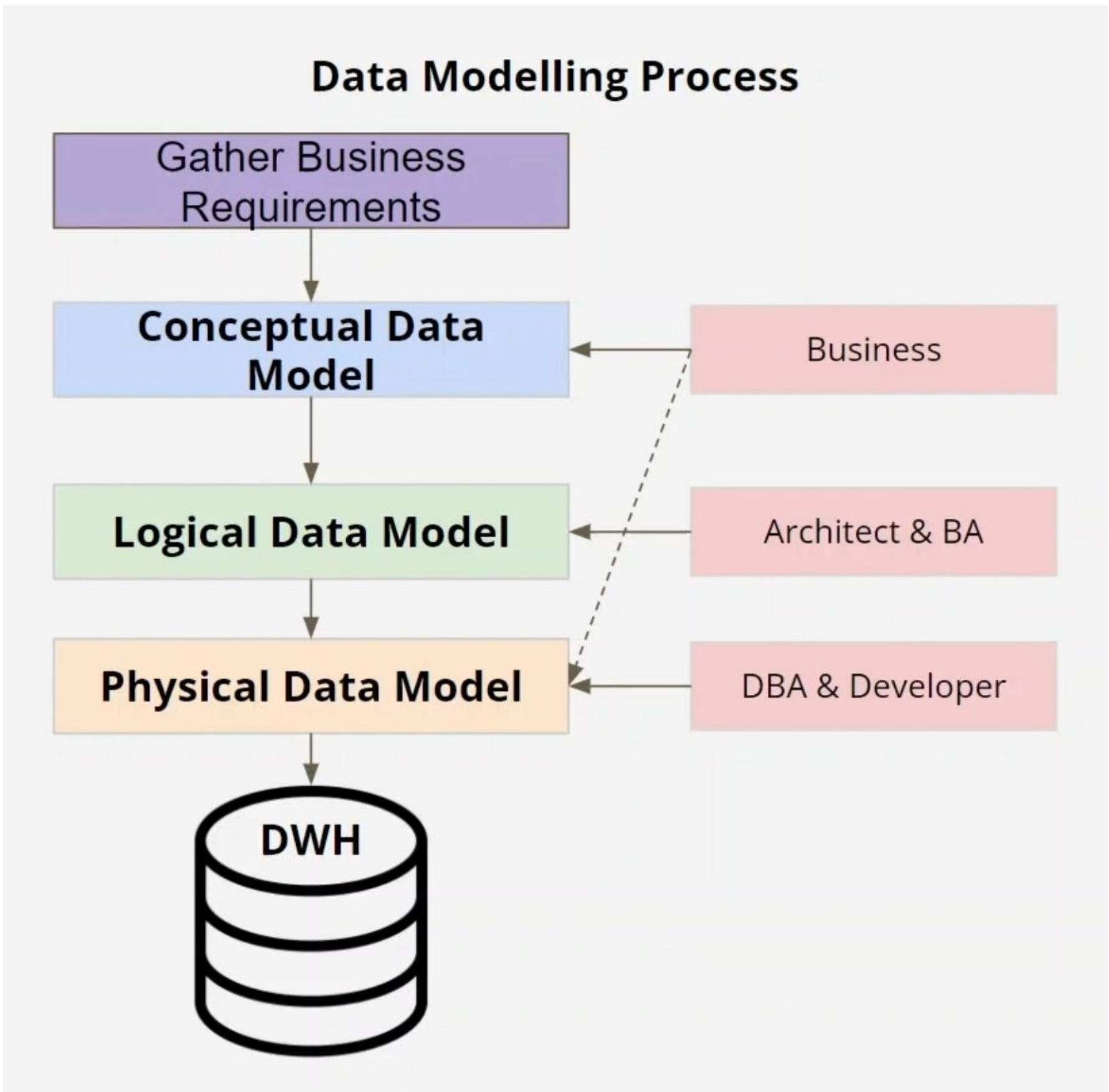
Cardinality:

- One customer can buy multiple products
- One order can contain multiple products
- One product can contain multiple order details



Three Types of Data Model

1. Conceptual Data Model (Mostly used by Business)
2. Logical Data Model (Mostly used by Architect and BA)
3. Physical Data Model (Mostly used by DBA and Developer)



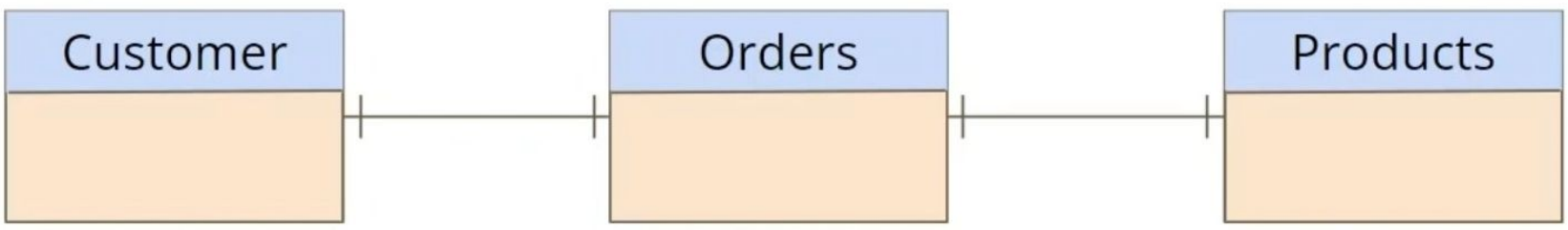
Conceptual Data Model

Identifies what the system contains but not the details. It shows how things are organized between entities and relationships, and is the first version of the data model so can't be used to build database.

Its goal is to act as a business requirements reference, to give an understanding of high level and how each entity relates to each other. But it doesn't describe the process flow itself.

It doesn't show primary keys or attributes, only entities and relationships. It's a very high level view for businesses to show the structure of data. It helps identify and helps with scoping requirements, and is useful for PMs as a communication tool.

At the conceptual data model stage, we have only identified entities and cardinality.

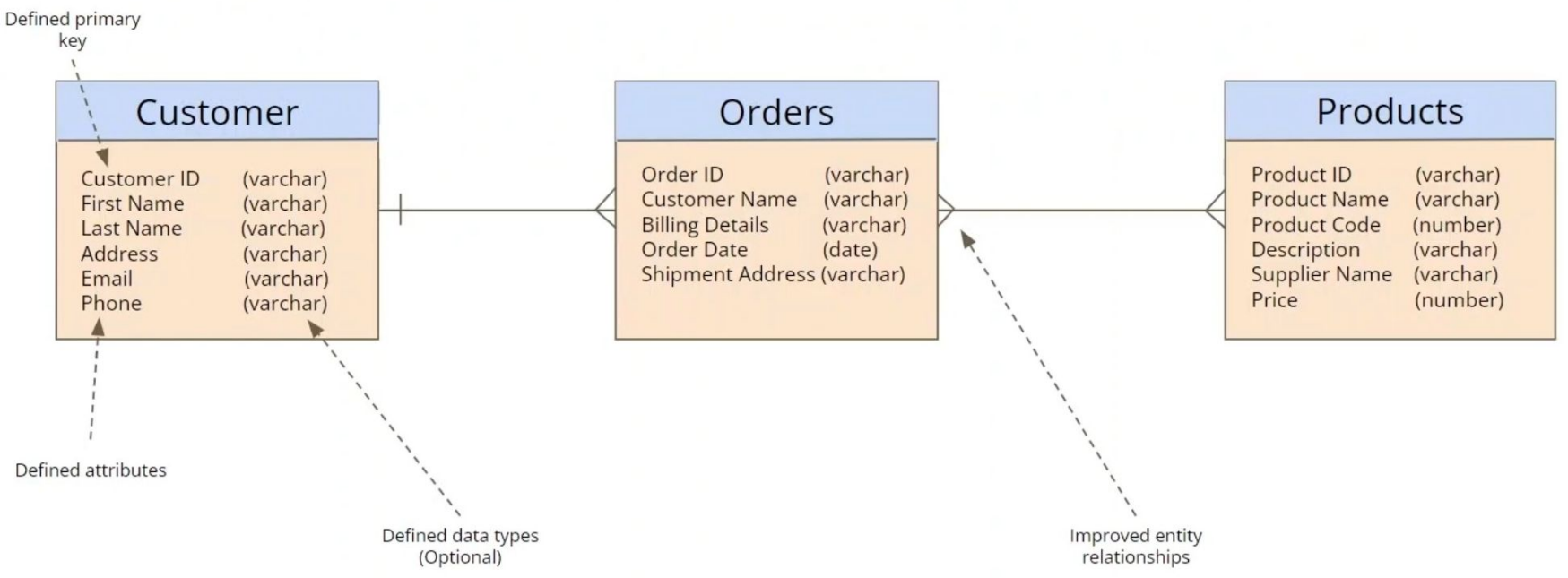


Logical Data Model

Identifies how the system should be implemented. They are less abstract and give more detail compared to conceptual data models. They focus on business requirements and the design of the data, not the physical implementation. It doesn't include any technical specifications, nor is it designed for a particular type of database or warehouse. Sometimes it is omitted due to people designing physical data models directly, and skipping this step.

Primary and foreign keys are present here, all attributes are specified within an entity and have data types. Relationships are specified using primary and foreign keys. You might choose to start normalizing your data at this stage.

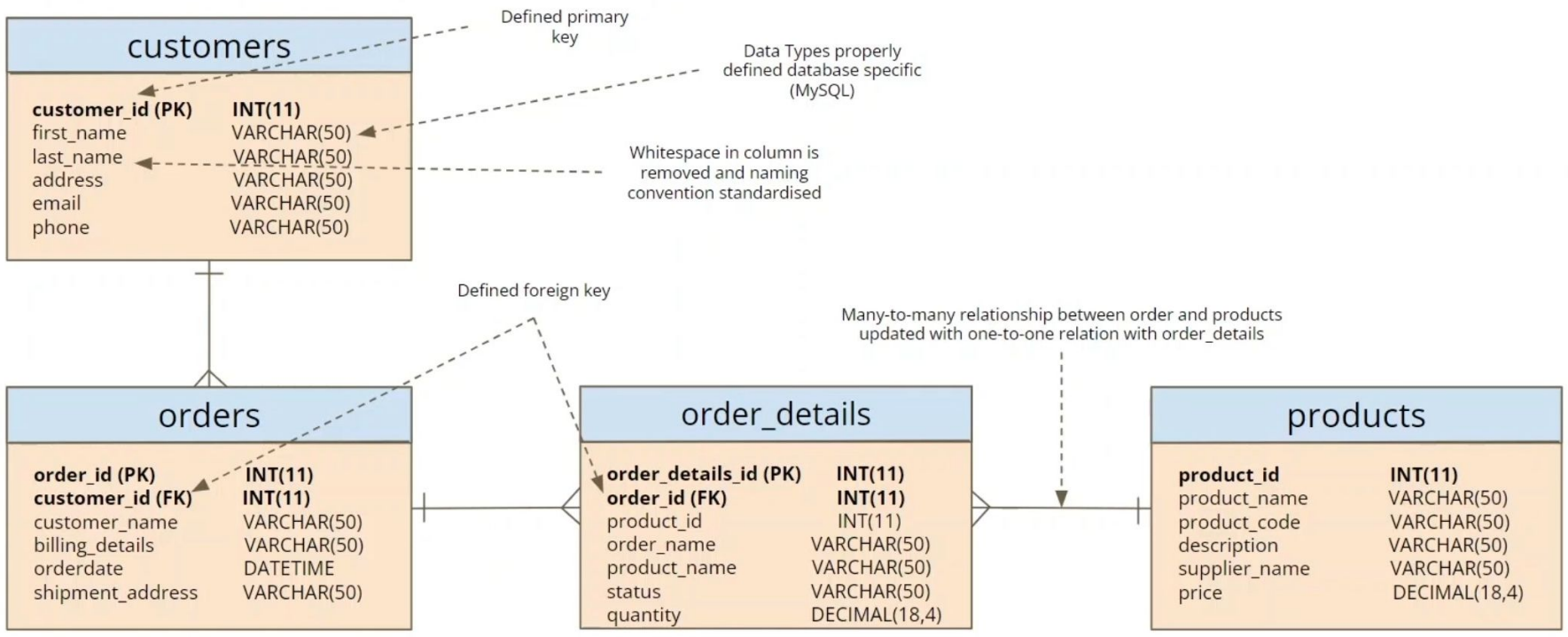
At the logical data model stage, we have identified relationships, primary and foreign keys, data types, and improved cardinality.



Physical Data Model

Identifies how the system will be implemented for the specific DBMS system. It acts as the blueprint for the developer and is therefore the least abstract of the three. Depending on the data modelling choice, some denormalization can occur.

Entities are now tables, attributes become columns, each column has a datatype, length, nullable is specified, and relationship cardinality is defined.



Conceptual vs. Logical vs. Physical

Feature	Conceptual	Logical	Physical
Entity Names	✓	✓	✓
Entity Relationships	✓	✓	✓
Attributes		✓	✓
Table Names			✓
Column Names			✓
Column Data Types		✓	✓
Primary Keys		Optional	✓
Foreign Keys		Optional	✓
Target Audience	Business	Architect	Developer

Recommended Tools for ERD

Lucidchart (most popular), Draw.io, erwin Data Modeller, ER/Studio, MySQL Workbench.

Different Types of Keys

A key is used to `uniquely identify a record in a table`, it’s a vital part of a relational table and is used to establish and identify relationships. There are a few different types of key…

- Primary Key
- Candidate Key
- Composite Key
- Natural Key
- Surrogate Key
- Foreign Key

Primary Key

Used to `uniquely identify each record of an entity in a table`. It cannot contain NULL and it must be unique.

This key is usually a candidate key. It’s very important to have a primary key and it works best if it is numeric or an integer. This makes it efficient for joins, index, and sorting.

An example would be customer_id.

Candidate Key

Any field that can be promoted to be used as a primary key. It should also be unique and not null. You can have more than one candidate key in a table. The developer chooses the most relevant key from the candidate keys, which then becomes a primary key.

An example would be, id_card_number which would be a candidate for a primary key if not for customer_id.

Composite Key

This is a type of primary key, it is created from a combination of multiple fields (attributes) which ensures uniqueness. You would create a composite key when you can't find any single attribute to serve as a primary key in your table.

An (extreme) example would be not having a customer_id or anything like that, and combining name, post code, and phone number, to create a unique field composite_key such as, james_tuner_ab12de_030325851.

We do have to be careful though when using fields that can change, like if a customer moves and his post code changes. A composite key therefore has a high risk of having to make lots of updates.

Natural Key

This is a type of candidate for a primary key. Natural keys have some business meaning and are real data within a table, it's already part of it when a new record is inserted into a table.

An example would be a social security number, and invoice number, an ISBN.

Because these keys are related to business processes, if there is anything that changes in the business regarding how ID card numbers are being used, that means we need to update all of our tables.

Surrogate Key

This is a type of primary key, usually generated at runtime (GUID, sequence, etc.). It has nothing to do with business meaning, it's just a unique identifier. It is typically a numeric value and is guaranteed to be unique.

An example would be just a random unique string of numbers. It has not value outside of the table. These keys have less impact on any business changes and are commonly seen in data warehouse systems.

Foreign Key

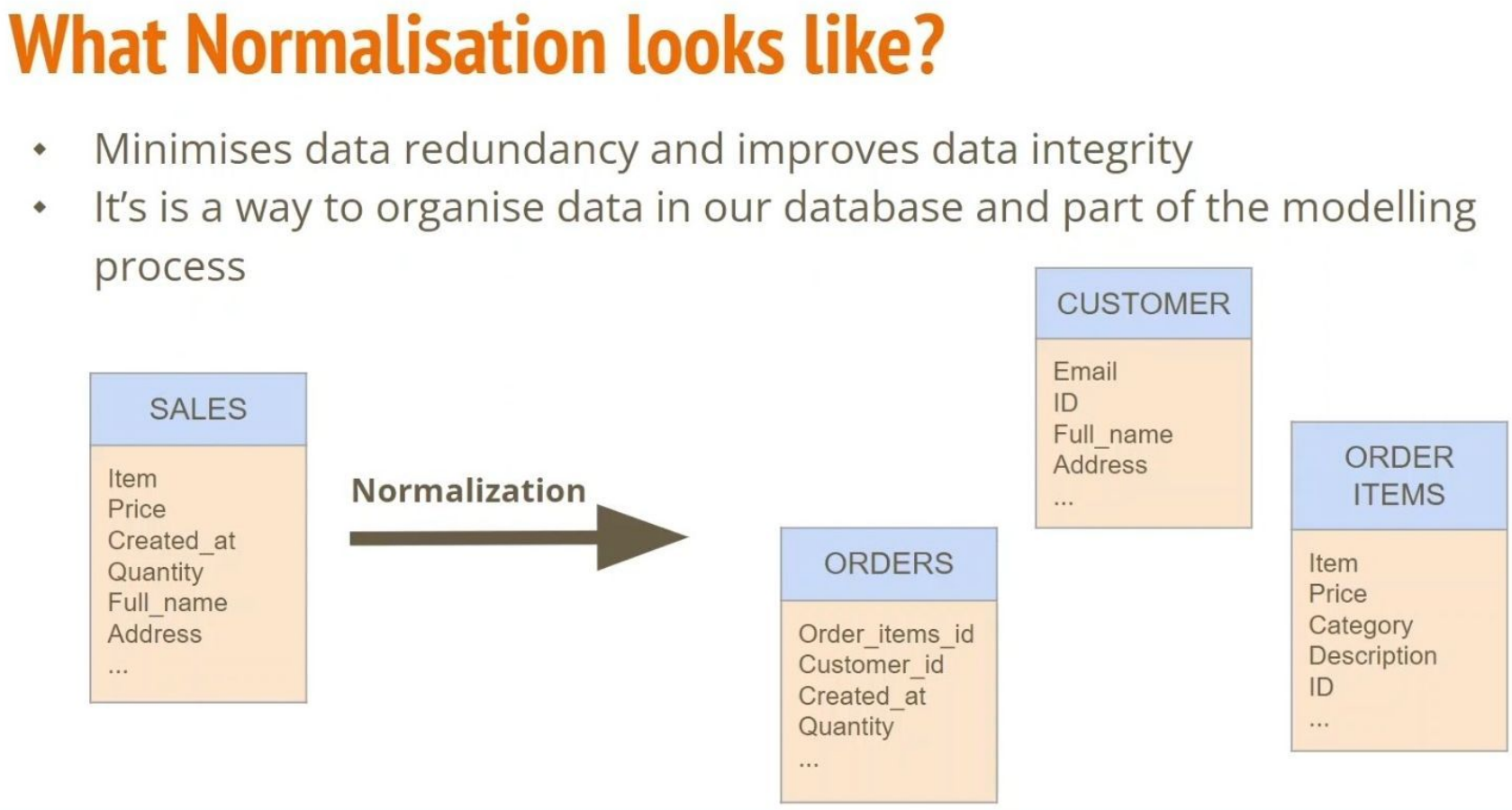
The main purpose of a foreign key is to join two entities by defining a relationship between them. When a key from a parent table exists in another table, it is called a child table and it tells us that records in a child table are related to the parent table.

An example would be a customers table containing customer_id as the primary key, and an orders table, also containing customer_id but with order_id as its primary key. This means customers is the parent table, and orders is the child table.

Normalization and De-Normalization

Normalization is a method of splitting large tables into smaller ones, and joining them using relationships. This improves data integrity, organizes data as part of the modelling process, and reduces data anomalies.

This concept was proposed by Edgar Frank Codd who also invented the relational data model.



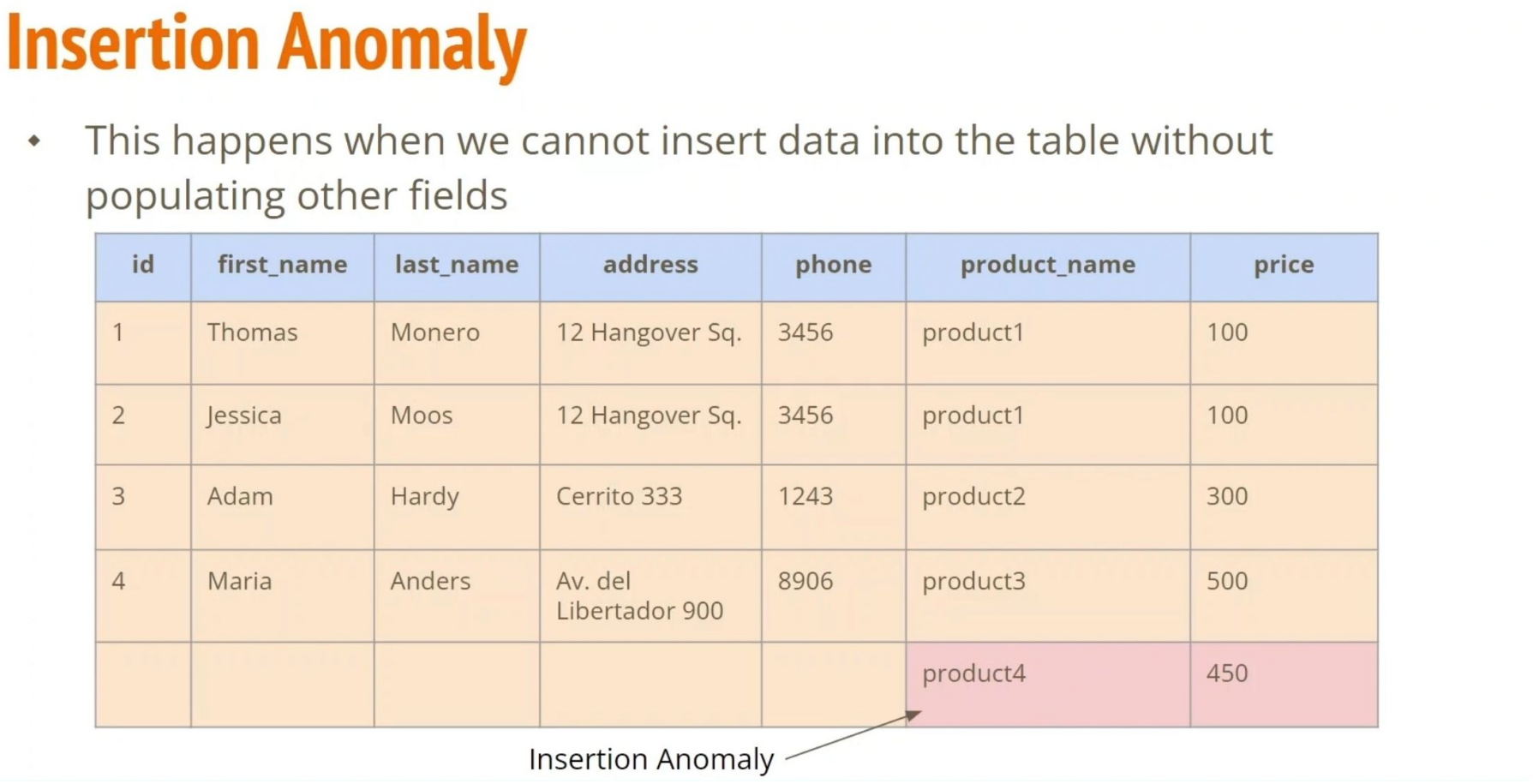
Anomalies

There are three types of anomalies that can occur when normalizing a table.

Insertion Anomaly

This happens when we `cannot insert data into the table without populating other fields`.

For example, if we want to add a new product number to our one big table, we can't add it because there are other attributes that need to be populated as well in order for the new product to exist. Especially the primary key, which cannot be null, but here is related to a customer.



Deletion Anomaly

This happens when there is an `unintended loss of attributes because we deleted another attribute`.

For example, if we discontinued product two and remove it, we also lose the customer details associated with that. Or vice versa, the customer decided to close their account

and so we lose the product information.

Deletion Anomaly

- Unintended loss of attributes due to deletion of another attribute

id	first_name	last_name	address	phone	product_name	price
1	Thomas	Monero	12 Hangover Sq.	3456	product1	100
2	Jessica	Moos	12 Hangover Sq.	3456	product1	100
3	Adam	Hardy	Cerrito 333	1243	product2	300
4	Maria	Anders	Av. del Libertador 900	8906	product3	500

Deletion Anomaly

Modification (Update) Anomaly

This happens when `partial update of the data causes an inconsistency in the database`.

For example, if we wanted to change the price of product one, we would have to do in multiple places.

Modification (Update) Anomaly

- Data redundancy caused by partial update of data causing inconsistency in database

id	first_name	last_name	address	phone	product_name	price
1	Thomas	Monero	12 Hangover Sq.	3456	product1	100
2	Jessica	Moos	12 Hangover Sq.	3456	product1	100
3	Adam	Hardy	Cerrito 333	1243	product2	300
4	Maria	Anders	Av. del Libertador 900	8906	product3	500

Update Anomaly

Normalization solves this problem by splitting the table into a customer table and a product table.

Normalization Levels

There are five normalization levels…

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)
- Fourth normal form (4NF)
- Fifth normal form (5NF)

`For most transaction systems (OLTPs) normalization achieves its best in the third normal form, this is why ER models are often called 3NF or a normalized mode.`

1NF

- Must have a Primary Key.
- Should only hold single valued attributes.
- Those single value attributes must be atomic (it cannot be subdivided). So for example, the name Jane Doe is not atomic because it can be divided into first_name, last_name.
- No repeating groups.

2NF

- Should be in 1NF already.
- Eliminate all redundant data.
- All the non-key columns should be dependent on the table's primary key.
- Should not contain partial dependency (when an attribute depends only on one part of a primary key, for example order_date and shipping_detail only depend on order_id, not product_id).

3NF

- Should be in 2NF already.
- Eliminate columns not dependent on the key.
- Should not have transitive dependency (where non-prime attributes depend on other non-prime attributes).

Pros and Cons of Normalized Model

Pros	Cons
Reduced data redundancy improves data integrity.	Performance issues due to multiple joins.
Helps reduce issues caused by data anomaly.	Slow speed due to multiple tables, but there are workarounds.
Smaller in size saving storage.	Not suitable for analytical queries.
No data duplications.	Complex database structure.
Easy to maintain and conceptually understand.	

De-Normalization

De-normalization is `an optimization technique with the aim of speeding up data retrieval`. It requires adding redundancy to various tables.

When designing your data model, you will need to try and find a balance between normalization and de-normalization. Usually this decision is driven by your business requirements.

De-Normalization Techniques

- Adding redundant columns.
- Storing derived columns.
- Pre-joining tables.

Adding Redundant Columns

When a column from one table is frequently accessed in another large table through joins, we use this technique. Since we are having to join frequently, by just adding them to the table, we avoid doing joins hence improving query processing cost and speed.

Storing Derived Columns

If you have to repeat a calculation again and again during queries, this created a lot of overhead on your system. By storing them, we are reducing the amount of time it requires

to process the query.

Pre-Joining Tables

This is similar to adding redundant columns, but we bring in all the columns. This is generally used when there is a one to many relationship between two or more tables.

Pros and Cons of De-Normalized Model

Pros	Cons
Joins pre-defined in advance, reduced query time and improves performance and speed of retrieval.	Large table sizes, increased storage cost due to large volume of data stored.
Less compute is required.	Possibility of data anomaly issues.
Accelerated reporting.	Less flexible.
Suitable for analytical purposes.	Insert and Updates can be complicated and expensive.

Data Warehouse Design Methodologies

There are various data warehouse design methodologies, including…

- Inmon
- Kimball
- Data Vault
- One Big Table
- Data Modelling – Then, Now, Next

Inmon Methodology

Created by Bill Inmon, the father of data warehousing. In this approach, the data warehouse is at the centre of the Corporate Information Factory (CIF). CIF provides a framework for being an enterprise wide data model, which aims to deliver business intelligence and business management capabilities.

Inmon’s definition of a data warehouse is that it’s subject-oriented, integrated, time-variant, and non-volatile. It starts with identifying the main subject used by the business, such as customer, or products.

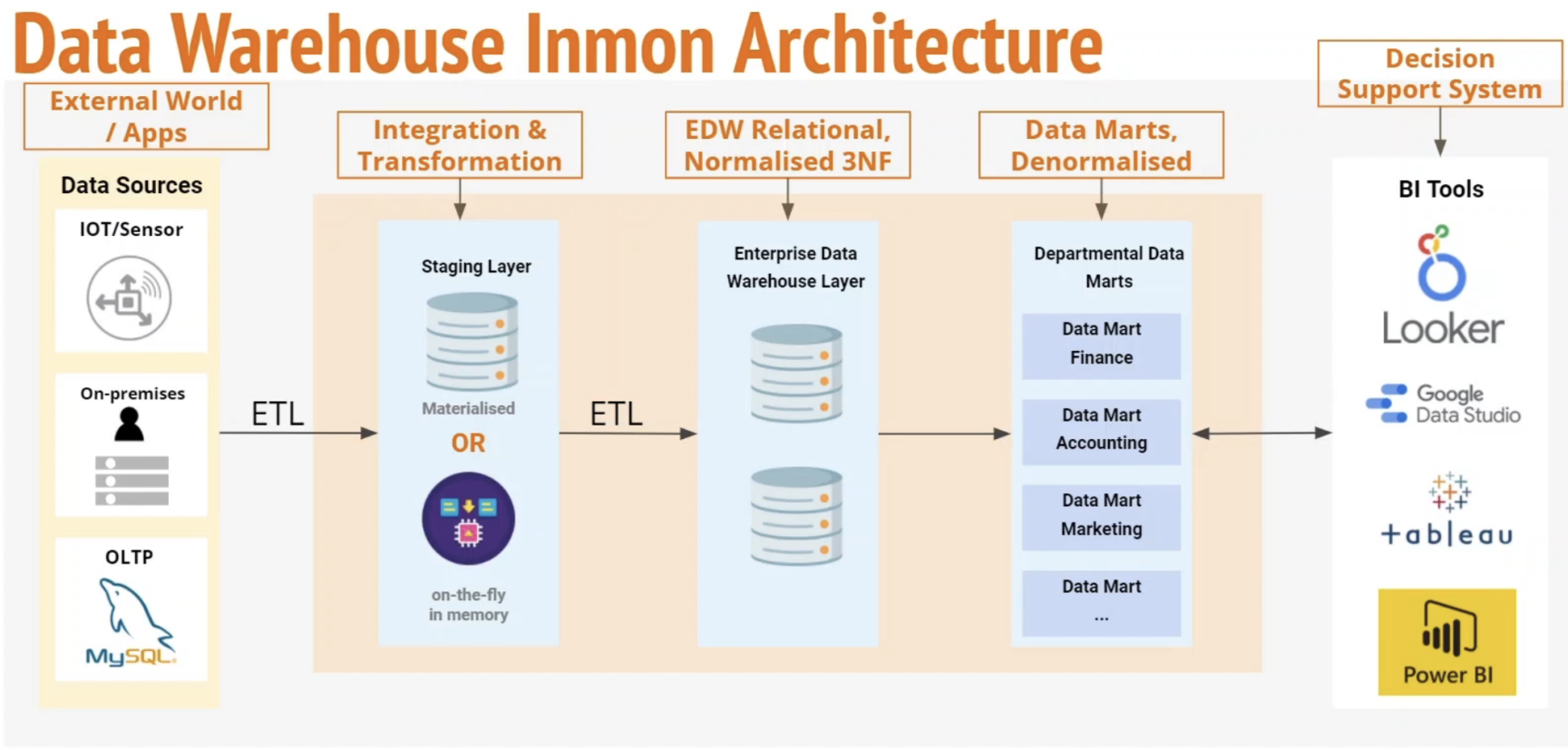
In this approach, Enterprise Data Warehouse (EDW) is the core of this architecture. It acts as the central repository for all business data. The Inmon method proposes creating data marts (usually de-normalized) for each business function, such as finance, sales, operations. It’s referred to as a top-down or data-driven approach.

Inmon CIF Architecture Components

- External World and Applications
- Integration and Transformation
- Operational Data Store (ODS)
- Enterprise Data Warehouse (EDW)
- Data Mart
- Decision Support System (DSS)
- Cross Media Storage Manager

External World and Applications	Integration and Transformation	Operational Data Store	Enterprise Data Warehouse	Data Marts	Decision Support System	Cross Me Storage Manager
Contains all raw data from external world, such as OLTP applications.	Here data is staged and transformed by ETL tools. Data from	Now the same as a data warehouse, but an interim area between	Optimized to store data for analysis and	Gets its data from Enterprise Data Warehouse.	These are different from data marts. DSS applications	As the da grows, la data warehous start to

External World and Applications	Integration and Transformation	Operational Data Store	Enterprise Data Warehouse	Data Marts	Decision Support System	Cross Me Storage Manager
Data can then be in various forms, structured or unstructured. Since it is source data, it contains legacy system data, duplicates, or inconsistencies.	various sources is brought together into a staging area, with the aim of making it corporate friendly. Can be time consuming. Consolidated and transformed data is then loaded to ODS and EDW.	database and data warehouse. It’s subject-oriented (organized around subject or organization), integrated (holds data from all different sources and entire organization), volatile (changes frequently), and current (up to date). It’s read-only, and stores snapshots of data from multiple transaction systems. Used for real-time or near-real-time reporting needs.	reporting purposes. They collect and aggregate data from many sources, handles big data well. It’s subject-oriented, integrated, time-variant, and non-volatile. It is usually using OLAP systems, the data is usually structured and 3NF, optimized for analytics queries, and provides data for data marts.	The data is organized in departments or business process, and de-normalized. There is better control over security as data is organized in small, manageable chunks. BI tools connect to data marts to build reports and unlock business insights.	are focused on a particular decision making process, such as marketing analysis, reporting, or building a KPI dashboard.	contain dormant data. Infreque accessed data is moved ou for archi storage area. Th improves performa and scalabil



Inmon Methodology Pros and Cons

Pros	Cons
Enterprise Data Warehouse acts as the one version of truth, as all the data is integrated for the entire business. The only source for reporting is the Data Marts and it is connected to EDW so it acts as a central repository for all systems.	Tables are normalized, and many joins involved adding complexity to the process, making it a time intensive process.
Since data is stored in 3NF, it has low data redundancy and a lower possibility of irregularities.	Since there are many components within the architecture and it is a top-down approach, it requires a really good understanding of

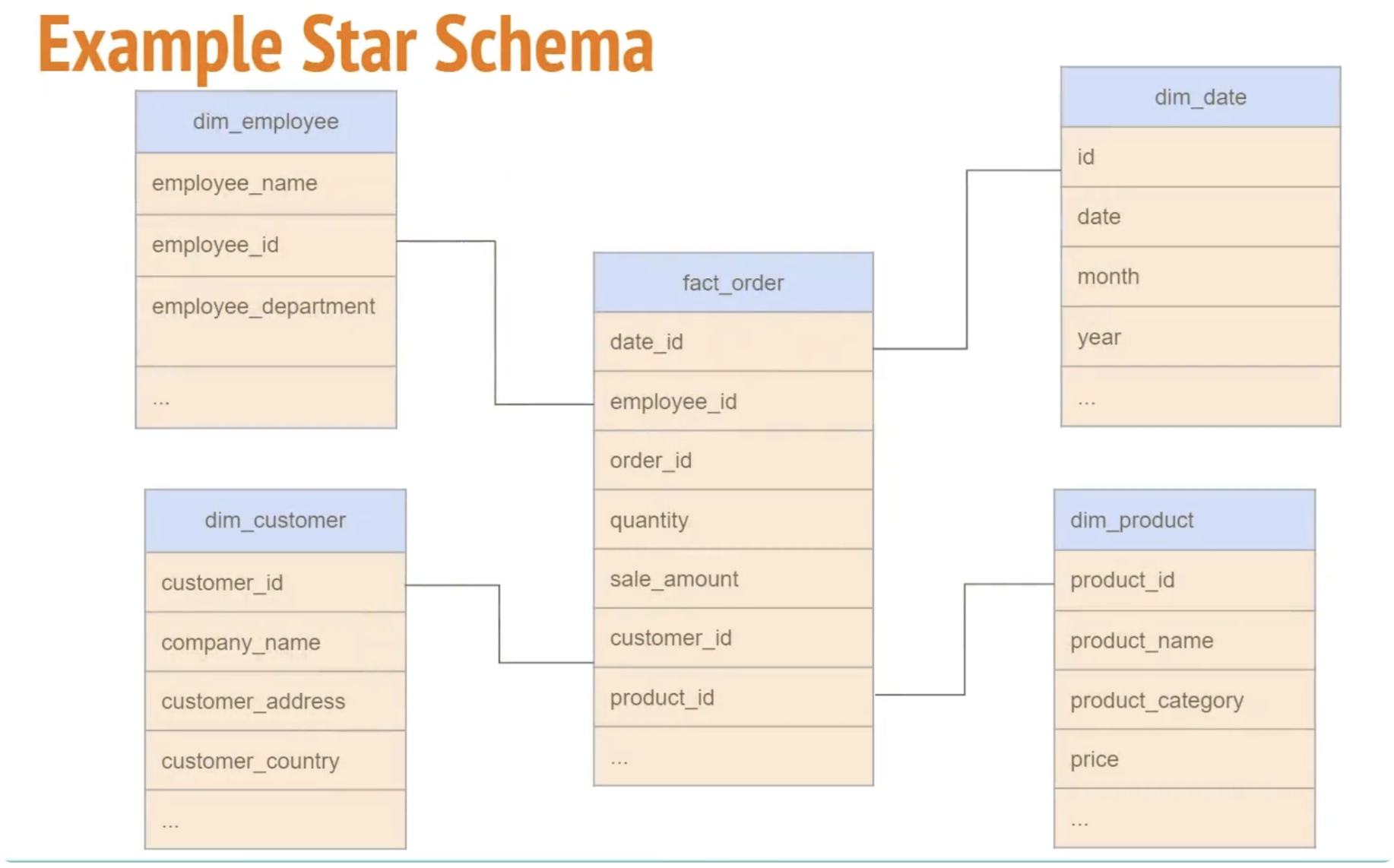
Pros	Cons
	the entire business before and during the modelling process.
Due to normalized nature, it is easier to add new data sources.	Additional ETL works are required due to data marts being built from enterprise data warehouses, so there is more setup and maintenance involved.
This data is easier to understand, making it easy to maintain and simplify business processes.	A little less agile in the approach and not suitable if you wanna be up and running fast.

Kimball Methodology

Created by Ralph Kimball. He defines data warehouse as a copy of transaction data, specifically structured for query and analysis. The Kimball methodology starts by identifying key business processes and end user requirements first. In this approach, a Data Mart is built first, prior to building an Enterprise Data Warehouse, similar to Inman's approach.

The focus here is to enable business intelligence and reporting as soon as possible within the organization. These data marts can later be combined together to create a data warehouse. We know data marts are smaller and easier to create, so this makes the modelling approach very quick and reliable. This modelling concept in Kimball methodology is also known as dimensional modelling. ==Dimensional modelling is denormalized by nature, making it perfect for speedy reporting and analytics. ==Data is loaded into a dimensional model, creating a star schema. The star schema is usually built on facts and dimensions tables.

Star Schema



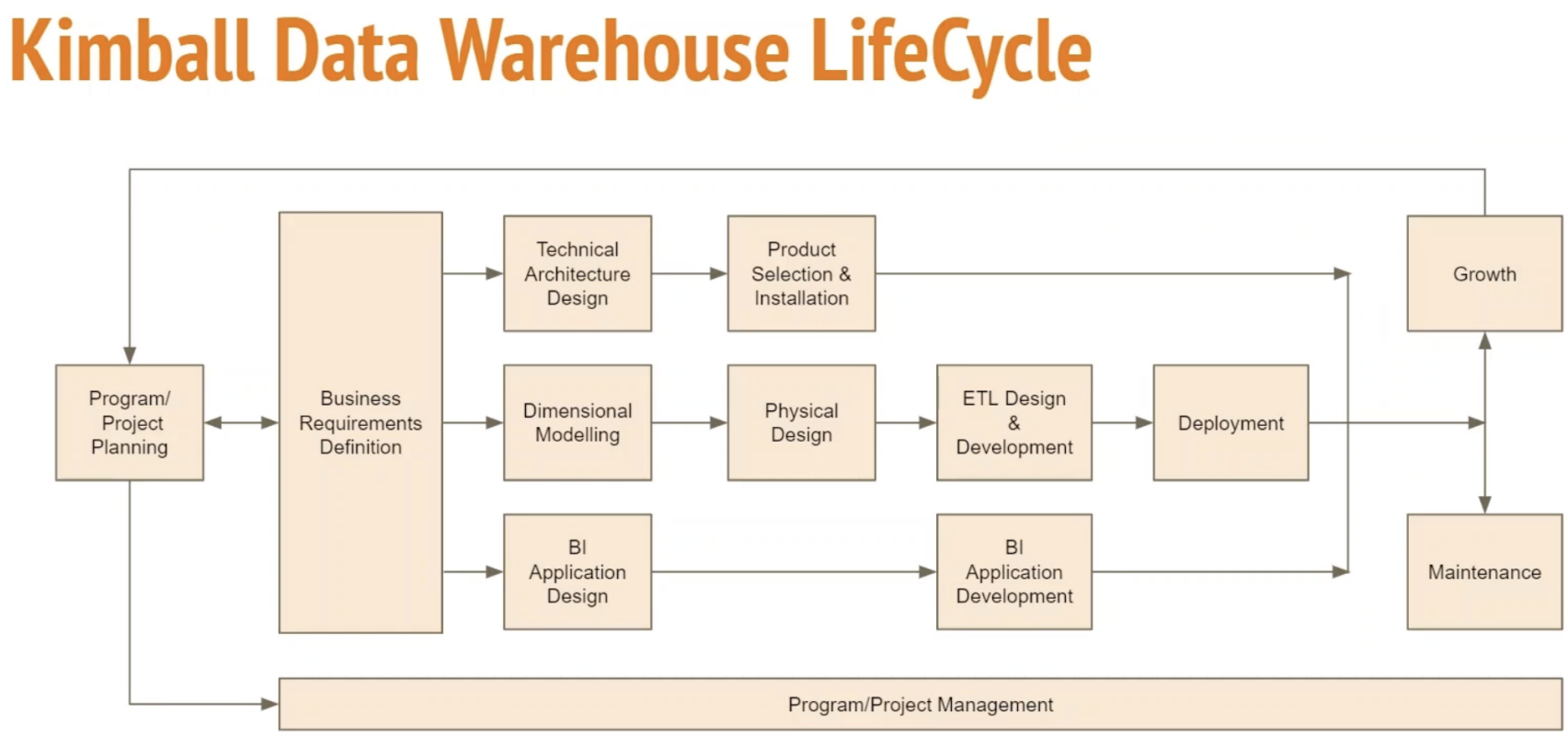
We can see the fact table is surrounded by many other tables, making this look like a star hence the name 'Star Schema.'

Processes of Kimball Methodology

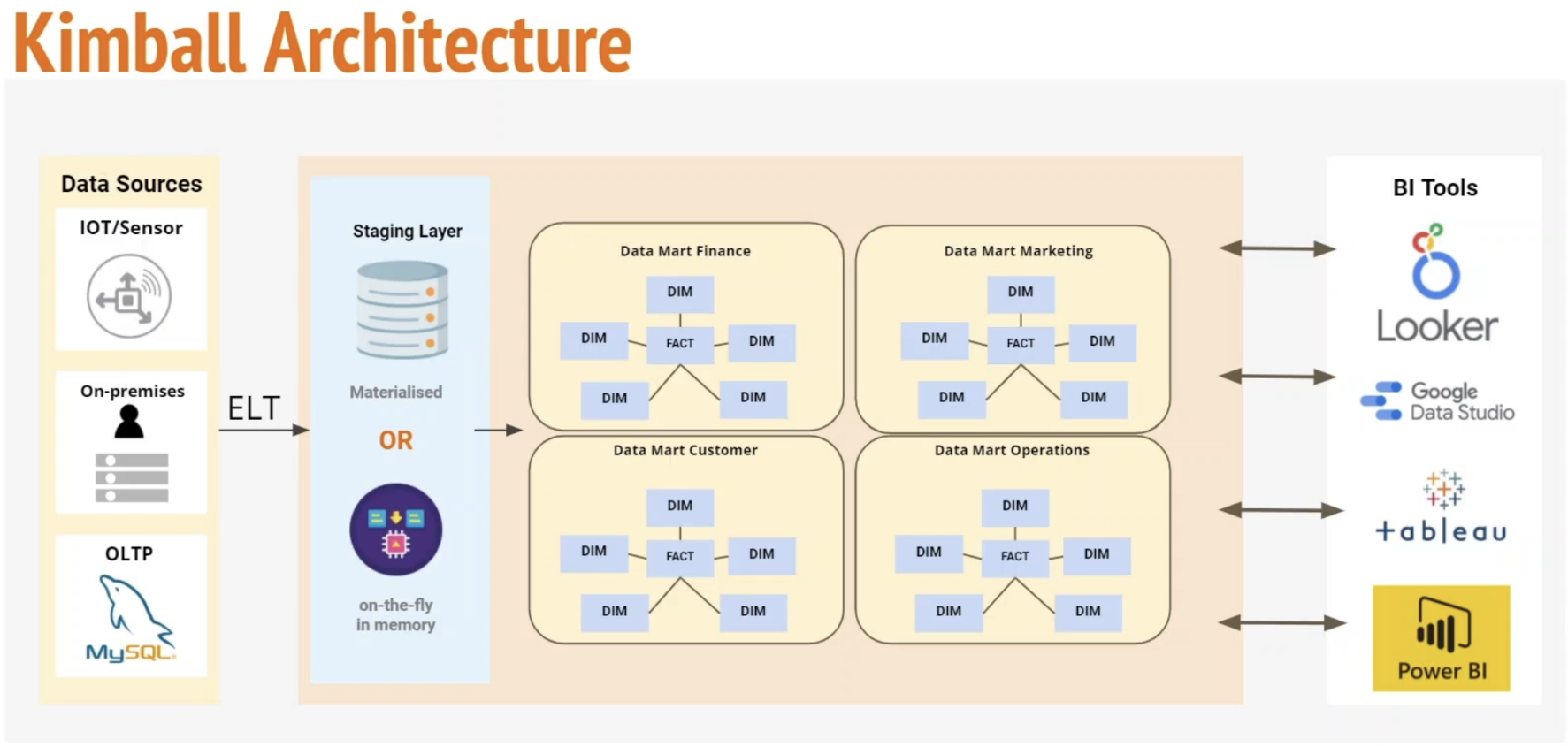
We follow a dimensional modelling technique here. We identify all the data sources and load it into our ETL processes. Kimball Bus Matrix is part of the Kimball Bus Architecture that records the facts and conformed dimensions. It shows how star schemas are constructed. This is used by the development and business team to decide how to implement

the star schema. Data marts are built with the Star Schema being the core element of the dimension model. More than one Star Schema can exist in the Kimball approach. All of these dimensions can share tables within the various data marts, which helps us integrate data through conformed decisions.

Kimball Data Warehouse Cycle



Kimball Architecture



Kimball Methodology Pros and Cons

Pros	Cons
It takes less time to build as data marts are built based on reporting and business requirements. This helps with speedy delivery of business values.	De-normalization introduces redundant data so there is a chance of data anomalies such as duplicates in the warehouse.
Since it’s incrementally built, it allows us to approach data modelling in a more agile way and helps with project management.	Since it is not integrated entirely we cannot really use this as a single source of truth.
Investment requires to build such a model is very low as you can pick and choose what to build and deliver quickly, instead of having to build everything in one go.	Denormalization also adds complexity to the ETL process. Adding new columns to fact tables can cause performance issues as fact tables are already created in depth and additional columns make the tables even bigger.

Pros	Cons
Data is denormalized, making it easier for users to work with, and keeps the model simple with less joins. This also enables faster querying and analysis of data.	Once the tables are created, it is slightly difficult to alter them, making this hard for dimensional models to accommodate changing business requirements.
Star schema allows business intelligence tools to connect and generate reports in a more simple way.	Due to the model being built with business process focused, not all enterprise data is available, making this challenging for all BI reporting needs.

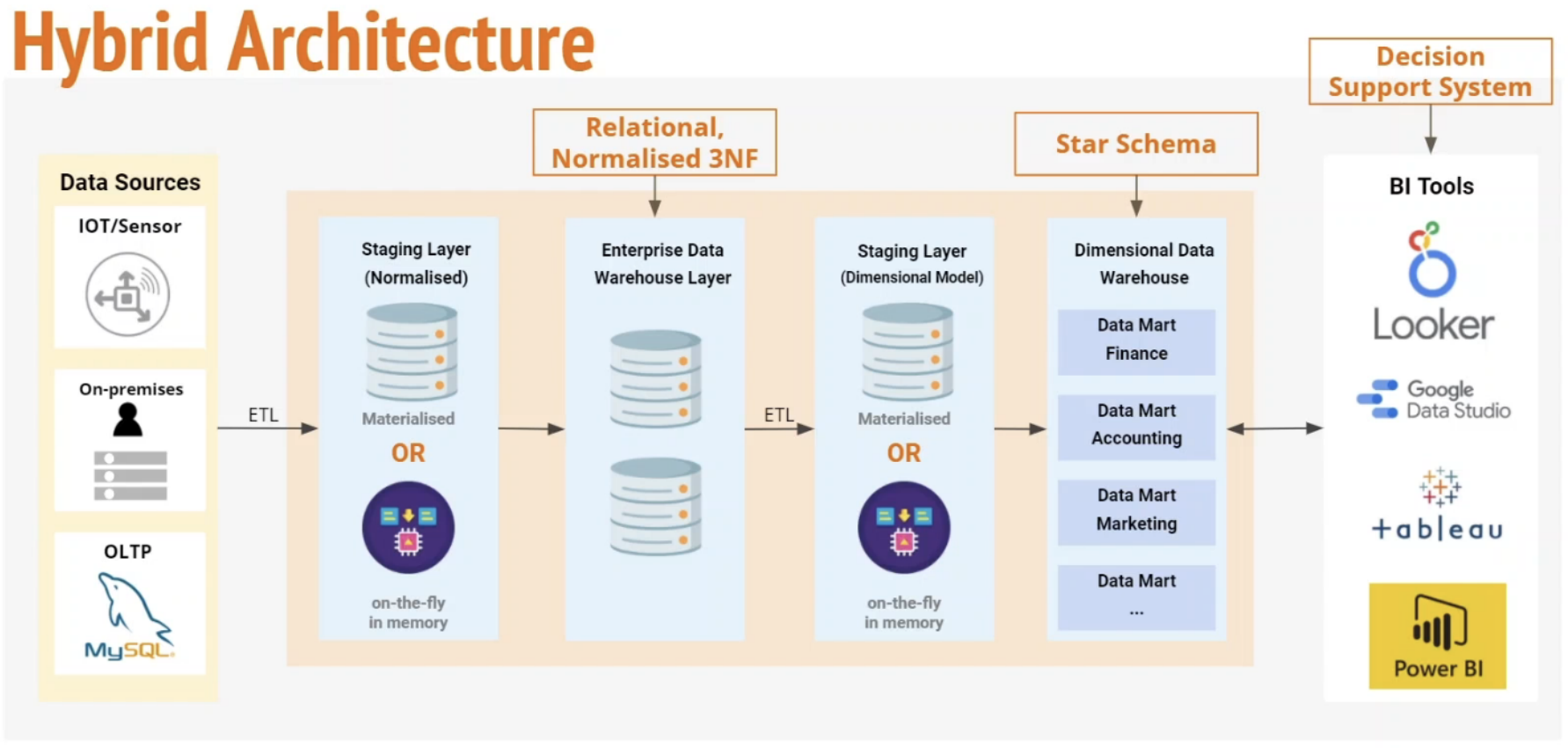
Inmon vs. Kimball

	Inmon	Kimball
Approach	Top-Down Approach which helps ensure the EDW becomes a single source of truth.	Bottom-Up Approach
Key Focus	Focus on Enterprise Wise Data Integration	Focus on individual business areas use-cases.
Data Format	Normalized so good for adding additional data sources.	Denormalized so good for fast query time.
Delivery Time	Takes a lot more effort at the start and requires very deep expertise.	Quick to set up with low initial cost, driving business value faster.
Maintenance	Medium-High	Low-Medium

It is ultimately up to the analytics engineer or an architect to make a decision on which approach to adopt for their projects.

Hybrid Architecture

The idea of this is to get the best of both modelling approaches from Inman and Kimball. Here we have an enterprise data warehouse in a 3NF, acting as our single source of truth for our organization.



Even though this hybrid approach is suitable where an organization has a pre-existing enterprise data warehouse, you shouldn’t build this hybrid approach for a completely new project, without a strong reason behind.

Data Vault Methodology

Created by Dan Linstedt to overcome the disadvantages of Inmon and Kimball’s methodology. It’s a hybrid approach, using 3NF and the Dimensional Model/Star Schema. It stores all of your raw data and then uses it for specific business needs. Data vault is designed to be

flexible and scalable, while adaptable to changing business requirements without having to re-engineer effort.

In data vault methodology, every entity tables (known as hubs) are connected through a link table, in the form of normalized tables. These hubs are also connected to sattelites. In this approach, data is kept and stored for historical tracking purposes, so only inserts are allows. Instead of using sequence numbers as keys, data vault uses hash keys.

Since data vault contains raw data for enterprise data warehouses and many relationships, it is not optimized for reporting by default. For that, we build data marts on top of the data vault, to enable reporting.

Data Vault Components

`Hubs`

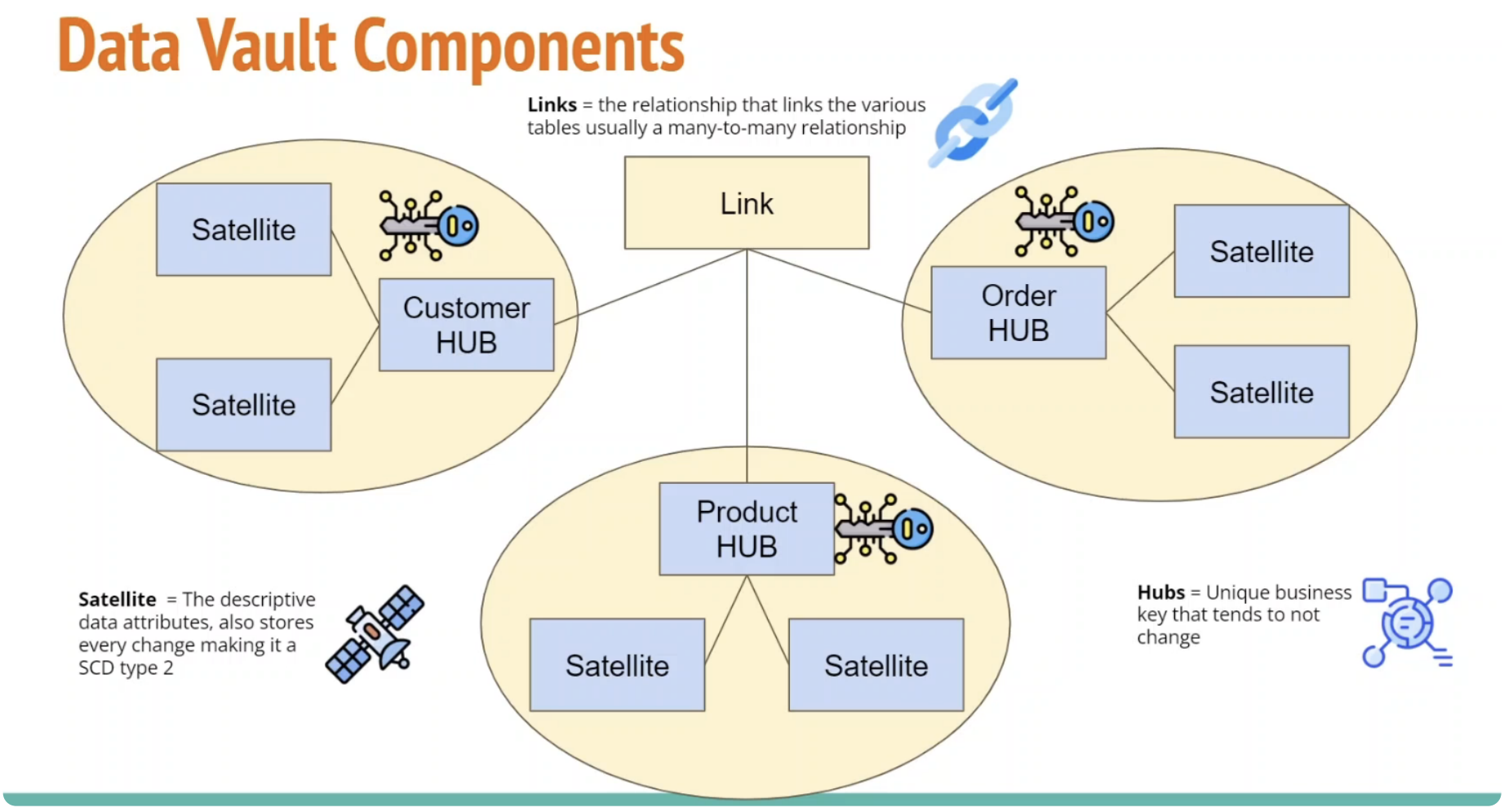
- Represent business entities (customer, products, employee).
- Contain unique business keys (hash) such as invoice numbers, or whatever organizations use day to day.
- Lost business keys result in losing all information associated.

`Link`

- Creates a relationship between different business keys in a way, by creating a link between different hubs.
- They are usually in the form of many-to-many relationships, and table is in 3NF.
- This modelling technique is designed for data warehouse, not for OLTP systems, making it easier to add new sources.

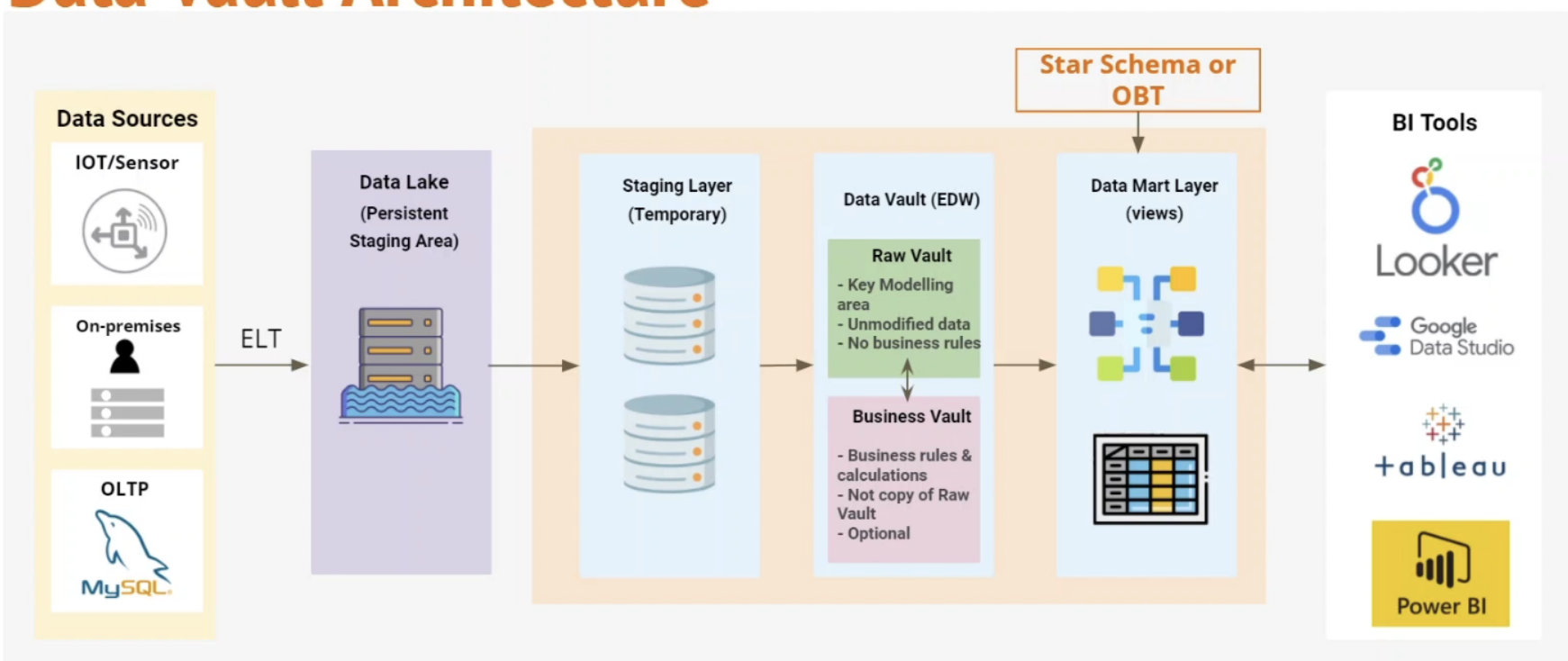
`Satellite`

- Contains attributes or descriptive data of the original tables. This is very similar to dimesnional tables.
- Since this data can change over time, the satellite also evolves as the data changes by storing data as its granular level.
- Satellites contain metadata that links to hub table.



Data Vault Architecture

Data Vault Architecture



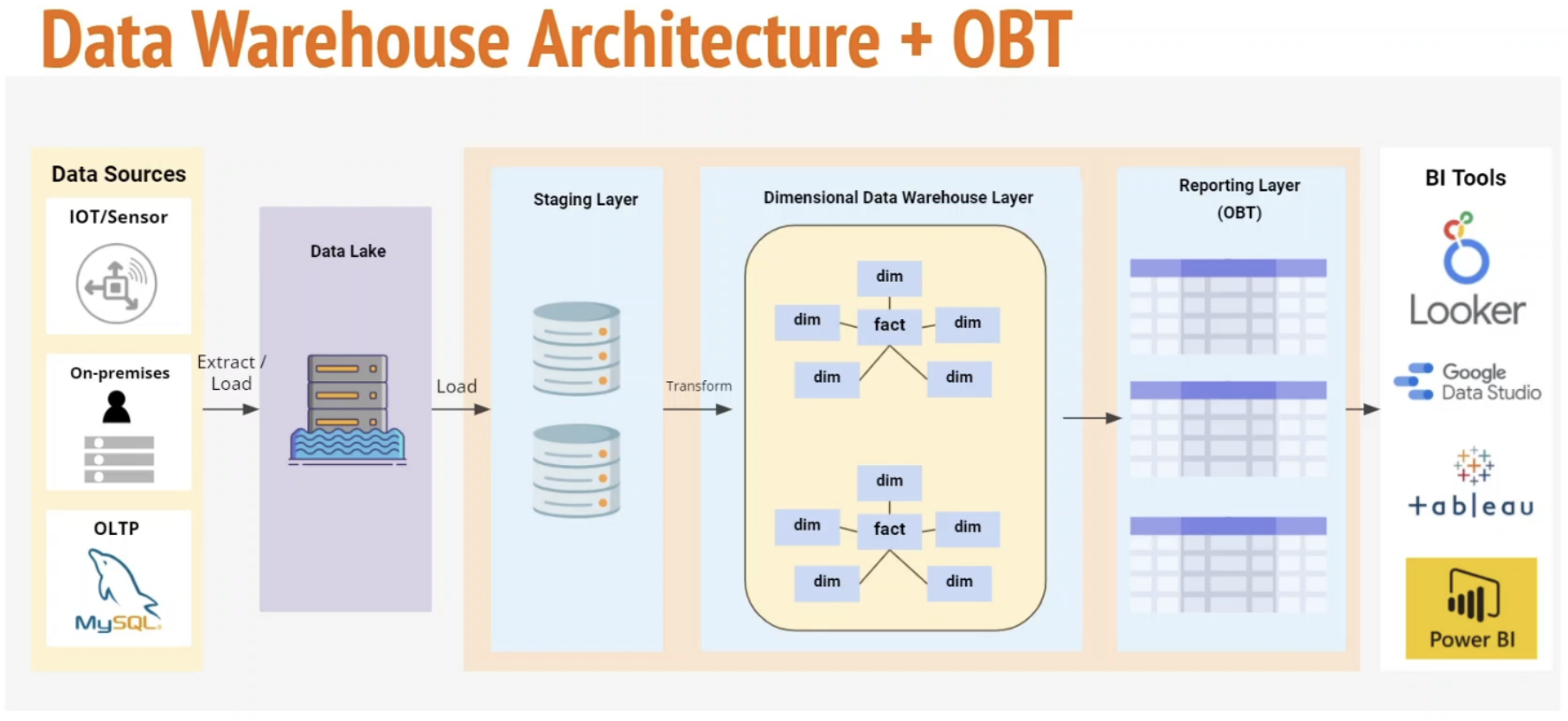
Data Vault Methodology Pros and Cons

Pros	Cons
Satellites in data vaults allow for historical data tracking, including relationships and attributes. We can event track source of data over time.	Data vaults are not suitable for all the data modelling requirements due to complex setup. It has a high learning curve.
Hubs, links, and satellites, allow for incremental builds that handle multiple sources.	It’s not suitable for small data sources.
Since data vault logically separates raw and business, it is able to handle changing business requirements.	Might lead to a loss of a single source of truth.
It’s easy to track and audit your data.	Proper training in data vault methodologies are required.

One Big Table (OBT)

This is essentially one very big denormalized table. All joins are removed and pre-aggregated, which provides faster query performance compared to dimensional modelling. Storage is affordable on cloud, and data warehouses are scalable. It’s also more intuitive to end users (business) since there’s only one table. It’s ideal for getting started and for smaller projects. For complex projects, you can use OBT as part of a star schema.

In the data warehouse architecture, OBT would act as our data warehouse, and become our one true source of truth. We then create data marts on top of our OBT.



One Big Table Pros and Cons

Pros	Cons
Makes data retrieval very quick due to lack of joins during query processing.	It is not suitable for large data warehouse projects with multiple sources.
It is very easy to get started with because you don’t have to apply complex modelling techniques and simple ETL/ELT tools are enough to denormalize tables.	Since these are denormalized tables, there will be redundant data and duplicates, making it hard for these tables to be a source of truth.
It is good for smaller teams who don’t have much data warehouse or data modelling expertise.	It is hard and expensive to handle any update operations since these are large tables.
Since these tables contain everything required by the business, it is quick for building reports.	Using OBT can make data management difficult, however creating views on top of OBT can help with this issue.

Data Modelling, Then, Now, and Next

Then	Now	Next
Inmon (1990), Kimball (1996), Data Vault (early 2000s) were all concerned with storage and compute power. Scaling infrastructure was complicated and process heavy, front designing schema was required, and ETL was used to maintain and built pipelines. All of these required a dedicated team of experts in data modelling techniques.	Due to technological changes and cloud services offering amazing scalability and processing power, we now have more flexibility. Columnar storage removes overhead to performance. ETL/ELT tools are very powerful. Transformation tools such as dbt make modelling very easy.	All of these approaches are valuable. Understanding each means we can make an informed decision on which approach is best for our use-case. Kimball approach for data management, star schema, easy updates. Inmon for EDW principles, enterprise warehouse architecture. Data Vault for agile, raw vault, business vault, data auditing.

Dimensional Modelling

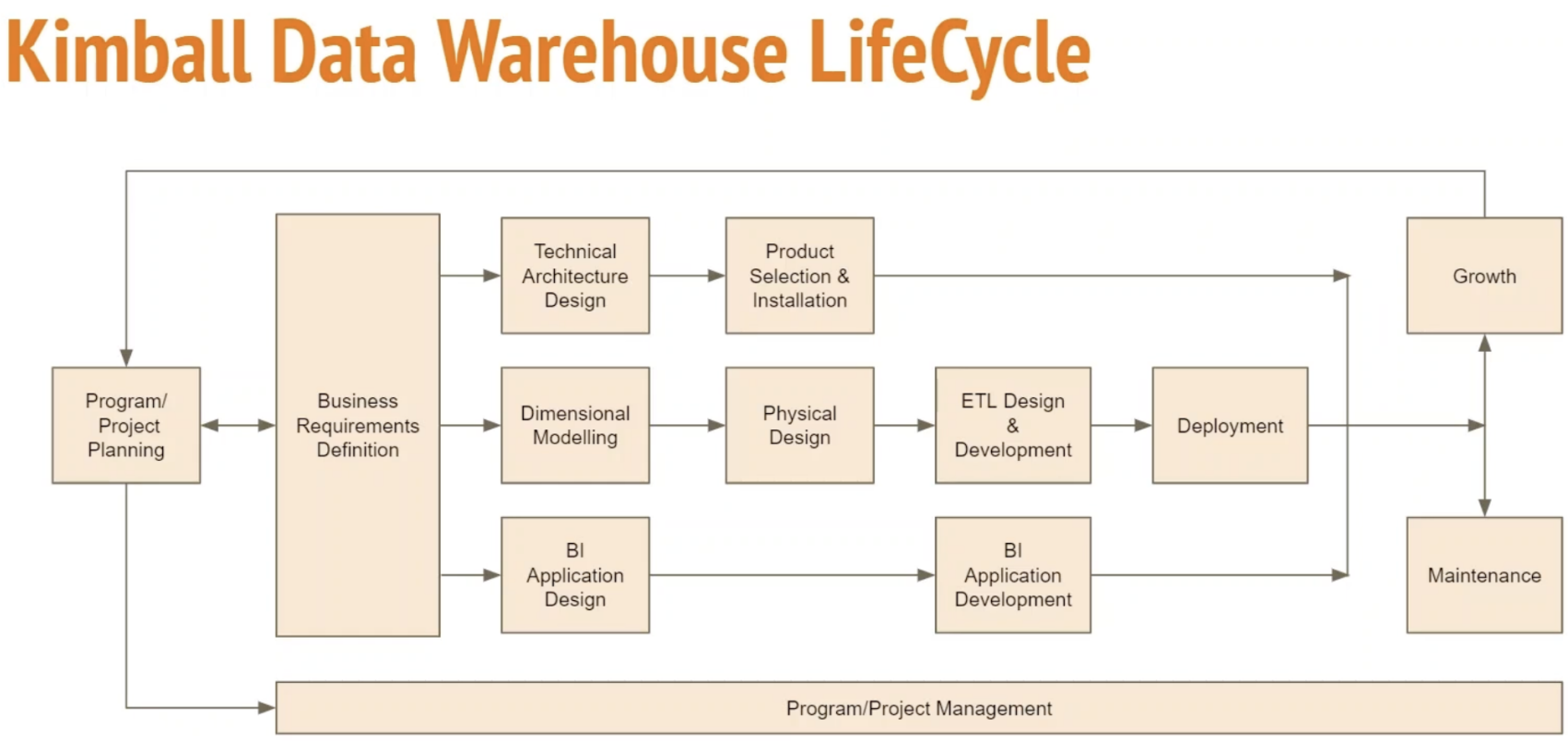
Dimensional modelling is part of the data warehouse lifecycle developed by Ralph Kimball. It's purpose is to enable data warehouse and business intelligence applications. The dimensional model is denormalized by nature, so it is optimized for faster retrieval of data and is also easier to understand and use by business users compared to the relational model—though conceptual model is also very easy to understand.

The reason dimensional modelling is so easy to understand for business users is because it groups data based on business requirements. Data is loaded into the dimensional

modelling process, creating a star schema.

Data Warehouse LifeCycle (Kimball)

This provides an overall `road map` to building data warehouses and business intelligence projects.



Here’s a closer look at each of the steps…

1. `Program and Project Planning`
   We need to get approval of ideas and budgets and assess if there is really a need or requirement for a new data warehouse system.

   What type of project is it? Is it a migration, or an improvement of an existing system?
2. `Business Requirements Definition`
   What are the reporting requirements? Here we assess data readiness and data source availability. These requirements are gathered usually through conducting interviews with stakeholders within the organization. All of our findings here are documented well.
3. `Dimensional Modelling`
   This is where we convert out findings from business requirements into business processes and create logical model. We select our business processes based on priority, and identify the grain of the facts and dimensions
   Then our star schema is built based on this dimensions and facts model. All of these details are noted down.
4. `Physical Design`
   Using the source document and the dim and facts identified earlier, we create SQL and schemas for our tables. At this point we usually have multiple environments such as dev, prod, etc. however for development purpose, we usually do it locally and then these changes are deployed to a test environment. Once tested, they are deployed to production.
5. `ETL Design and Development`
   Deploying the changes and testing them is usually managed here.
6. `Deployment`
   During this stage, a lot is happening at the same time, and other parts of the LifeCycle are taking place, such as `BI Application Design` and `BI Application Development` and `Technical Architecture Design` and `Product Selection and Installation`. Security discussions, which OLAP solution to choose from, etc.
7. `Maintenance and Growth`
   Once our production is deployed, we continue to maintain the system and promote growth and adoption of solutions.

8. `Program and Project Management`
   This is a very agile approach, and it's up to the organization to use this as a base.

Program and Project Planning

It can be difficult to know where to start, since you're going to be changing the organization's data culture.

A program, is `a long term strategic approach`, with no fixed deadline. It contains multiple projects, which are either related or happen sequentially. Due to this there is usually no fixed deadline. It is usually governed by senior stakeholders.

A project has `a very specific aim, to deliver a functional business product`. It's time-bound, has an end-date, and a dedicated project team.

To actually begin planning, you do need a good understanding of business and management concepts. At a high level, we start by `defining a project` by meeting with stakeholders to learn…

1. Why is this data warehouse project is required?
2. How is this going to benefit the whole organization?
3. How will this project impact ROI and give a competitive advantage?
4. Can we assess readiness? How strong is support from our sponsors? Do we have investment and resources? Is our business motivated? What is our data and organizational culture like?

We then `develop the project plan`, including cost, resourcing, start date, and completion date. We see whether we have the right team of people to do this, and if not, how do we hire or bring in consultants to fill in the gaps.

Project team members usually contain at a minimum, a data architect, a data engineer, an analytics engineer, a BI analyst, and a project manager.
Analytics engineers here, handle data transformation and model the data so that it is ready for use. They have expert knowledge in various data modelling technologies, such as BigQuery, Snowflake, or RedShift. They have a good understanding of OLTP systems as well as data modelling techniques. Since their aim is to build the most performant data warehouse, they also use data transformation tools such as dbt.

Business Requirements Definition

In this step, our `focus is to gather all the requirements for the project`, as much as possible. It is a critical step because before we start developing our conceptual or logical model, we want to make sure we have all the information and we're not missing anything critical that might impact the success of the project.

We do this through `interviews` with business users, executives, to find out all the requirements and success criterias.

- We understand the needs of the business. We look for KPIs, existing technologies, and analytical systems.
- We are assessing team skillsets, any pain points and inefficiencies that exist in the current system and the way of working.
- We plan interview questions in advance.

Then we do `data source analysis or data profiling` to understand that we have all the data that is required for building a data warehouse. Interviews are a great first step but we can't rely on them to give us all the information that we need.

- We need to get permission requirements for these data sources.
- We need to meet with data source experts and explain the requirements, finding out if there is any potential risk.

- We need to find out whether the tables we'll be using are indeed correct or are there any new tables we should be considering instead.
- We need to understand what data is coming in, and what data is going out of the system.
- We need to create a data flow diagram to give us better clarity on flow of data between different systems.

Here's an example of a data sources list…

Business Process	Data Sources	Data Source Location
Purchase Orders	Source 1	London
Warehouse Deliveries	Source 2	Munich
Warehouse Inventory	Source 3	Munich
Store Inventory	Source 4	Munich
Retail Sales	Source 5	London
Shipments	Source 6	London

During `data profiling, we need to look out for`…

- Missing data or data sources which can impact timeline.
- Check the relationship of data and ensure all the tables have correct relationships.
- Do a distinct count to surface any duplicate data.
- Quality of data is important here to determine how much time data cleaning will take.
- Volume of data is important so we can build our system as scalable as possible.
- We need to identify if the data will be batch or stream, since this impacts what ETL or ELT tools we use and how we model our data.
- Checking for data redundancies to see if additional transformation is required.

Then, we write up all of our findings in the form of `documentation`. This includes a well defined success criteria, all the data sources that will be used, all the business processes that were mentioned during interviews ordered by priority, user stories, and business processes are also documented in an Enterprise Bus Matrix with high level entities.

This will become `our reference document` and will include…

- Operational activities performed by the organization.
- Captured performance metrics that translate into facts in a fact table.
- Fact tables focus on the results of a single business process.
- Defining a specific design target.
- Allows to define the grain, dimensions, and facts.
- Corresponds to a row in the bus matrix.

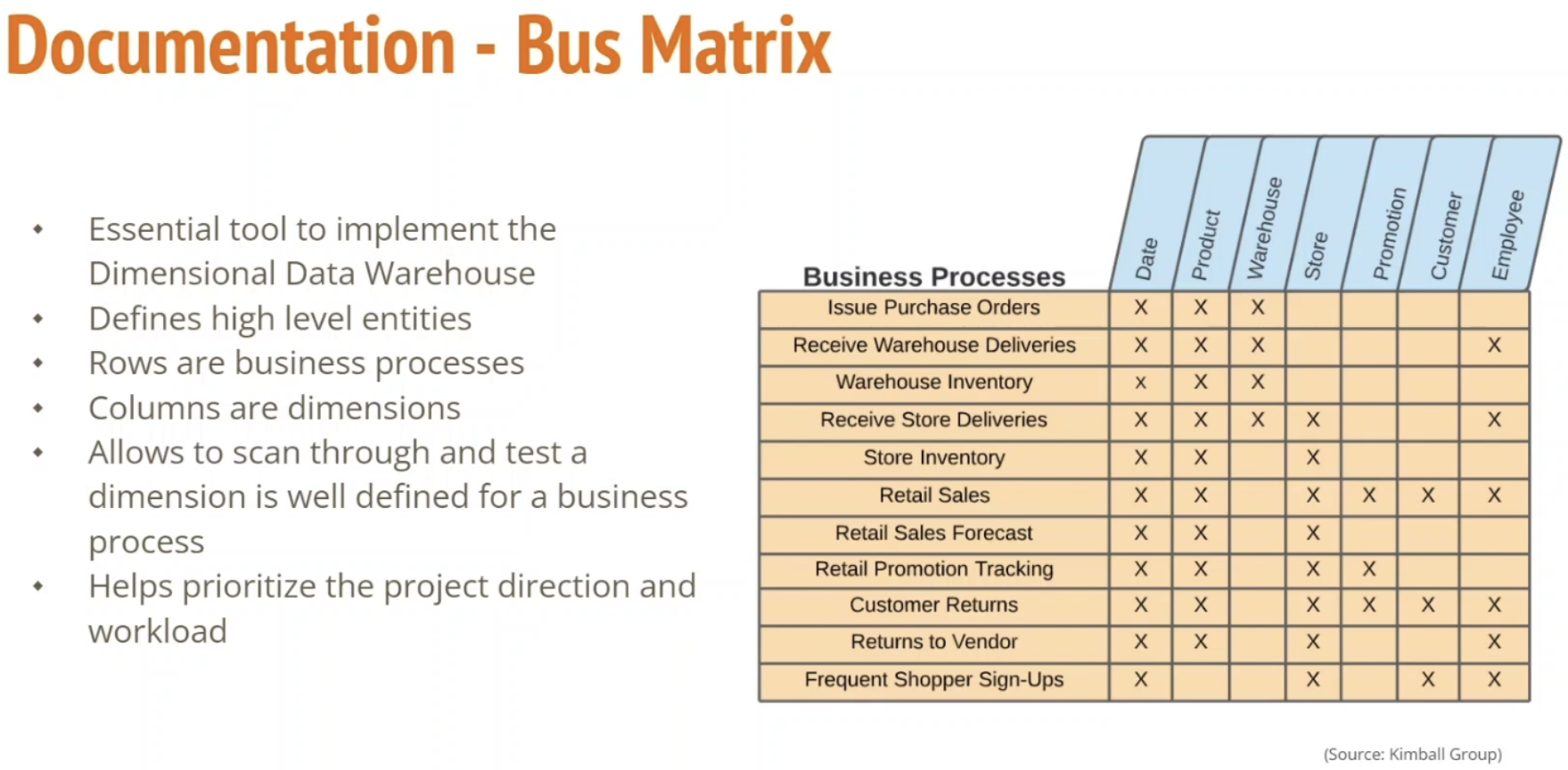
An example of the `business process list` will look like this…

Business Process	Data Readiness	Data Quality	Complexity	Business Priority
Purchase Orders	High	High	Low	High
Warehouse Deliveries	Medium	Medium	Medium	Medium
Warehouse Inventory	Low	High	High	Medium
Store Inventory	Low	Medium	Low	High
Retail Sales	High	High	Low	High
Shipments	Medium	Low	High	Low

An example of `user stories` will look like this…

ID	As a…	I want to…	so that I can.	Priority
1	Business Analyst	Understand total sales quantity for any period of time(day, month, year)	create sales performance report.	High
2	Manager	Find top sales representatives and bottom sales representatives	track their performance and recommend improvements.	Medium
3	Business Analyst	Understand best and worst performing products for any period of time (day, monthly, year)	create product performance analysis.	Medium
4	Business User	View product and sales across country and region for each supplier	product KPI report for stakeholders.	High

An example of a `bus matrix` will look like this…



Dimensional Modelling

`The key concepts of dimensional modelling are dimensions and facts. Fact tables, surrounded by dimension tables, become a star schema in a dimensional model.` We can have multiple star schemas in the same dimensional data warehouse.

Dimensions

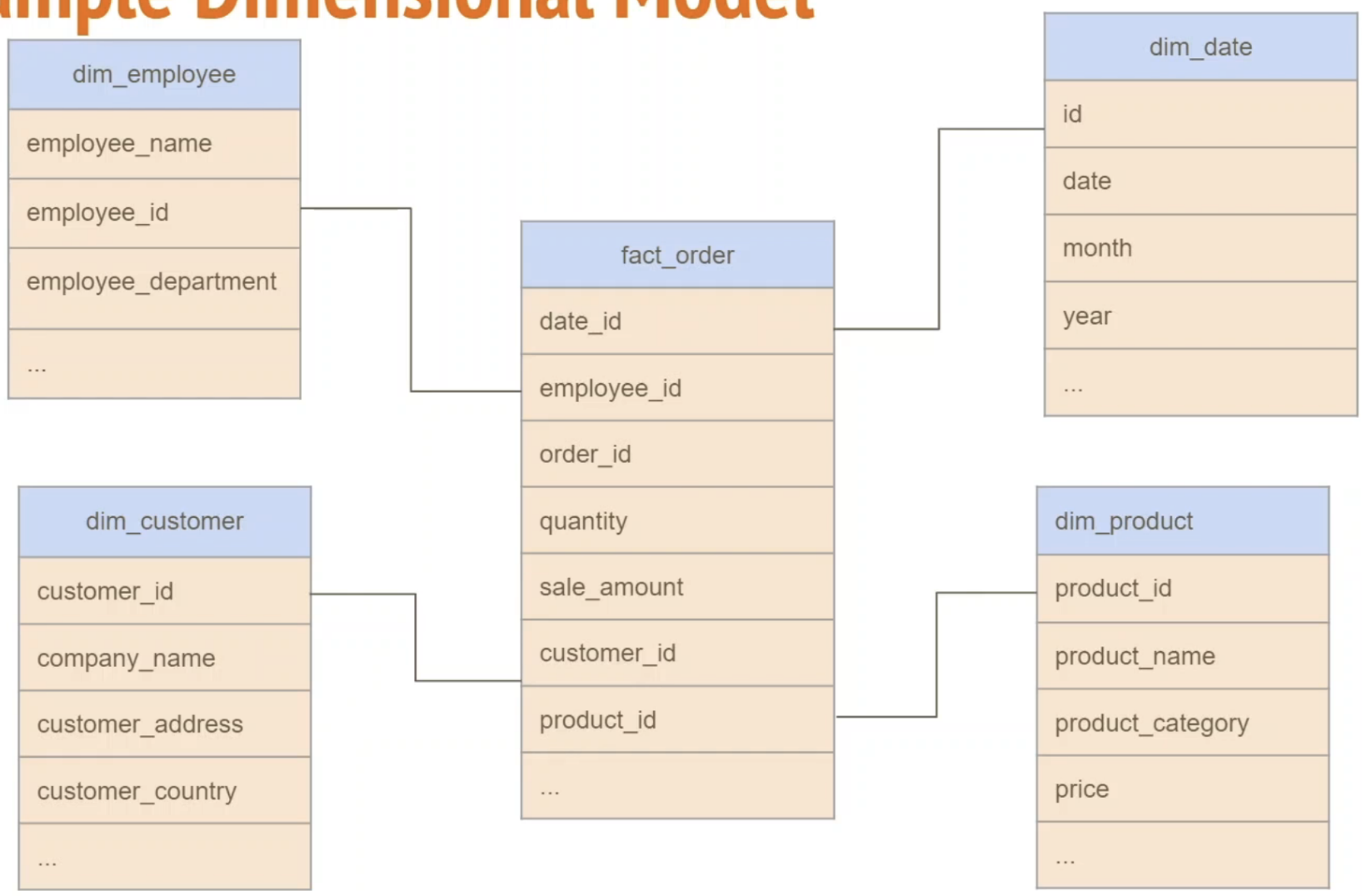
- A dimension is essentially descriptive information about facts.
- Example: Customer, Date Products
- Attributes define characteristics of a dim such as, customer_name, product_description.

Facts

- A fact indicates business measurement or business activity, like transactions and sales.
- Example: a product was sold for $50.
- Measurement in fact defines quantitative values (COUNT, SUM, AVG, etc.)

Example Dimensional Model (Star Schema)

Example Dimensional Model



The tables are related via foreign keys. For example, the order table can be joined to the customer table on customer_id.

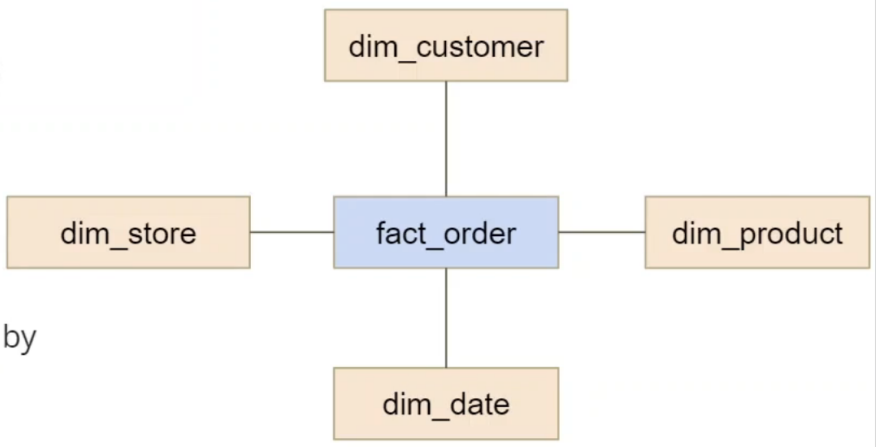
Steps of Dimensional Modelling

1. Select the Business Process
   - Operational activities performed by your organization.
   - Capture performance metrics that will get translated into fact tables.
2. Declare the Grain
   - A critical process that establishes what a single row of a fact table represents.
   - This must be declared before anything else as it dictates the design of dim and fact tables.
3. Identify the Dimensions
   - Who, what, where, when, why, and how context of a business process.
   - Whenever possible, a dim should be a single value when associated with a given fact.
4. Identify the Facts
   - Measurements in numeric values that result from a business process.
   - Only facts that are consistent with the declared grain are allowed.

Steps of Dimensional Modelling Example

Steps of Dimensional Modelling (Summary)

- Business Processes are converted into dimensional model.
- Facts & dimensions are identified from business process
- These dimensional model becomes a star schema or snowflake schema in your data warehouse



How does this works?

Business Process: Create Sales Reporting to understand sales by customer and to increase profit

User Story: As a Business Analyst I want to understand

"How many apples were sold in last 7 days?"

Fact | Dimension/ Product | Business Process | Date Dimension / Duration

Dimensions

Dimension tables store descriptive fields, such as product_name, product_category, price, description. Using dimensions tables, we can group similar attributes into a subject area. Using the dimension attributes, business users can filter, group, and measure the performance of a business process.

Dimension tables are usually denormalized, and since they store attributes in a single place, there are usually no redundancies. Dimension characteristics should be verbose (full words), descriptive, and complete with no missing values. They usually represent a many-to-one hierarchical relationship.

Conformed Dimensions

Any dimension that is joined across multiple fact tables, is called a conformed dimension. The aim of a conformed dimensions is to ensure consistency and quality control of the data, by providing the same structure attributes and meaning in every fact table.

Data is stored at an atomic level, every row is unique, and it is easy to update as all business rules are in one place. They have primary and surrogate keys. Date dimension is a common, conformed dimension (year, month, week, day).

For example, say there are two star schemas, but they both share dim_customer, dim_product, and dim_date. All three of those would be conformed dimensions. It also means we can join those two star schemas.

Junk Dimensions

These are simple dimensions, whose value does not change frequently. They have simple attributes, like flags, yes/no values, true/false. If we create dimensions for each of these listed attributes, we create complexity by introducing additional joins, but when we group together these highly correlated attributes, it is called a junk dimension. We use it to reduce the number of dimensions because we now have one junk dimensions table which will join to one fact table instead of multiple dimension tables. Updating and managing these tables becomes much easier for developers and filtering becomes easier for users.

For example, say we have a fact_shipment table, with additional dimensions such as dim_ship_type, dim_ship_container, and dim_ship_priority and each of those contains a ship_container_code and ship_container_description. We know that if we want to join all these to our fact table, we're going to have to do multiple joins. So instead, we can create a single dimension with all of these attributes, and join to fact table using that single junk dimension. So we create dim_shipment_codes which contains all of the above attributes.

Degenerate Dimensions

These are dimensions without attributes, which exist inside a fact table. These are usually high cardinality or unique, like receipts or invoice numbers, tracking numbers, order numbers. They are stored inside a fact table to reduce duplications. They are used as grouping for business reports instead of aggregations or filters.

For example, say you have your fact_order table and a customer purchases one bar of chocolate, which generates an invoice number. The invoice number here is our business key which we can use to find all the information we want about this transaction at any time. The invoice_number is stored inside the fact_order table, and you might think, should I create an invoice table for this? We can't do this because all the information for that already exists in various dim tables like dim_customer, dim_product, etc. So there's no point in creating another dim table, as an invoice table doesn't have any attributes of its own. That's why it's a degenerate dimension.

Role Playing Dimensions

This happens when the same dimension is used for multiple purposes. The most used role-playing dimension example is the date dimension. A fact_order table contains order_date, due_date, and cancelled_date, and in this case we can create a date dimension, instead of creating multiple date dim tables, we just create dim_date.

Slowly Changing Dimensions (SCD)

These are dimensions that change over time. For example, customers can change their name, address, phone number, and we need to make sure our data warehouse is designed to accommodate this. Managing both current and historical versions of this data is important to track changes over time.

There are the main different types of SCD such as…

- Type 0: Retain original value.
- Type 1: We overwrite the older value with the new value.
- Type 2: We add an additional record.
- Type 3: We add a new attribute column.
- Type 4: We use historical tables to track all the changes.

SCD Type 0

Here there is essentially no change or updates. Even though the attributes change, it never gets updated regardless of any changes.

This approach is not used in most businesses because data changes and we need to track those changes to provide business value.

For example, is a customer changes their number, we do not update it in the table.

SCD Type 1

Here we overwrite the old value and don't maintain any historical content.

So if a customer changes their phone number, we just put their new number in over the old one.

This isn't the most optimal solution however as we lose track of our historical data.

SCD Type 2

Here we create a new additional record, with a new primary key (surrogate key), flag column (Y/N) to identify what is the latest record, and date column to see when the record was last updated.

For example, we would have introduced new columns into our table such as valid_to, valid_from and current_flag. We also have a customer_sk which doesn't have business

meaning, but acts as a unique identifier for that table inside the data warehouse.

So if our customer changes their phone number, we will create a new record. We will update columns like valid_to and current_flag to indicate that this is now an old record. In our new record we will have a new surrogate key, and updated valid_from column and current_flag with Y to indicate this is our latest record.

SCD Type 3

Here we override the records similar to Type 1, but we introduce an additional column to track important business changes. We don't add additional *rows* to track all the changes, just a column that tracks current and previous values, meaning we cannot track historical changes.

For example, instead of just one column for phone_number, we have current_phone and previous_phone.

SCD Type 4

Here we introduce a separate historical dimensional table to track our changes. Sometimes we have data that is changing frequently and quickly, and when this happens our table size can grow very large which impacts performance. That's why we keep our latest or current data in our original table, and introduce a separate historical tracking table.

For example, the current table will contain the new phone number, and the historical table will have columns such as start_date and end_date.

SCD (Store as Snapshots)

This is a fairly modern technique to track SDCs, mostly due to cloud data warehouses which make things scalable and storing redundant data cheap. In this method, instead of following our various SCD types, we store all of our data as snapshots.

For example, in BigQuery we can create partition tables, and all data is added to snapshot daily or weekly. So in our phone number example, the table would contain the phone numbers, and a date_partition column which represents that we have taken a daily snapshot of our entire data, including all the records, both changed and unchanged. To retrieve the current data, we just need to filter the date_partition to today.

Bridge Tables

Bridge tables are used to resolve many-to-many relationships between facts and dimensions. Bridge tables sit between facts and dim tables and only contain the key columns from both the tables they represent. So a bridge_sales_vouchers table that bridges fact_sales and vouchers, will contain product_id and voucher_id. Adding this bridge table solves the many-to-many relationship issue.

Facts

A fact table is usually the centre of the star schema and contains measurements of business activities, created by operational systems. For example, a fact_sale table would contain measures like a daily sale_amount and quantity. Fact tables usually don't have any duplicated data because they are usually normalized. They store data at the lowest level of granularity, making them very deep tables.

Fact tables contain two types of columns, foreign keys (linking to dim tables) and measures (to perform queries and aggregations).

Relationships between fact and dim tables are usually one-to-many. Sometimes there are many-to-many situation, but we use bridge tables here to resolve this issue.

A primary key in a fact table is usually a composite, for example a combination of order_id, employee_id, customer_id, and product_id.

There are two different categories of facts. Types of Facts (Measures) have three sub-types, Additive, Semi-Additive, and Non-Additive. The other category is Types of Facts Tables which have three sub-types, Transaction, Periodic, Accumulative.

Additive Facts

When a measure can be summed up across all of our dimensions within the fact table, the it is called an additive fact.

For example, if the measure quantity_sold can be summed up and grouped across any of our dimensions such as date, order_id, supplier, or product, it is an additive fact. The sum(quantity_sold) makes business sense across all dimensions so quantity_sold is an additive fact.

Semi-Additive Facts

When a measure can be summed up over only some of the dimensions within the fact table, then it is a semi-additive fact.

For example, unit_in_stock tells us how many units are still in stock after every transaction and we can sum this up across product and supplier to understand total units in stock, however it is not additive towards date or order_id as we can't sum up these values. We can do an average for date or order_id however, making this a semi-additive fact.

Non-Additive Facts

These are measured that cannot be summed across any of the dimensions within the fact table.

For example, discounts are stored as percentages in our table, so they cannot be summed. It is not going to give us any additional information, and even if we do try to SUM or AVG it, we will get the wrong information. Same goes for something like unit_price.

Transaction Facts Table

This is the most basic and common table in dimensional modelling. It gives us the most granular view of business operations by storing one row per transaction. Since these are usually stored at date and time level, it also provides the lowest level of granularity. Because of this, the measurements are additive. However this does mean these tables can grow large in size in a very quick amount of time. Usually there are no updates in this table, only inserts.

Periodic Facts Tables

These store snapshots of data for a specific time period such as day, week, month, hour, etc. The grain is one row per time period, and since these are snapshots across time-period, measures stored inside this table are aggregated at some level, making them semi-additive. These tables are usually built from transaction fact tables, since they are large in size. We can use periodic fact tables to build smaller, aggregated tables.

Accumulative Facts Tables

These have one row per record for the entire lifetime of an event or product. It contains a well defined beginning and end date so that we can insert data at any time as a snapshot. This allows us to represent milestones or progress of a business activity through multiple date columns.

Star Schema

This is the most common schema, its a simple structure to organize data within your data warehouse.

There is a fact table at the centre of the star, which connects to the dimension tables. This enables easier querying and understanding.

It usually contains a mix of normalized (facts) and de-normalized (measures) tables. Fact tables are usually large by size as they store a large amount of data, and this is where we usually apply our calculations and aggregations. Dimensions, we use in our reporting for grouping and filtering.

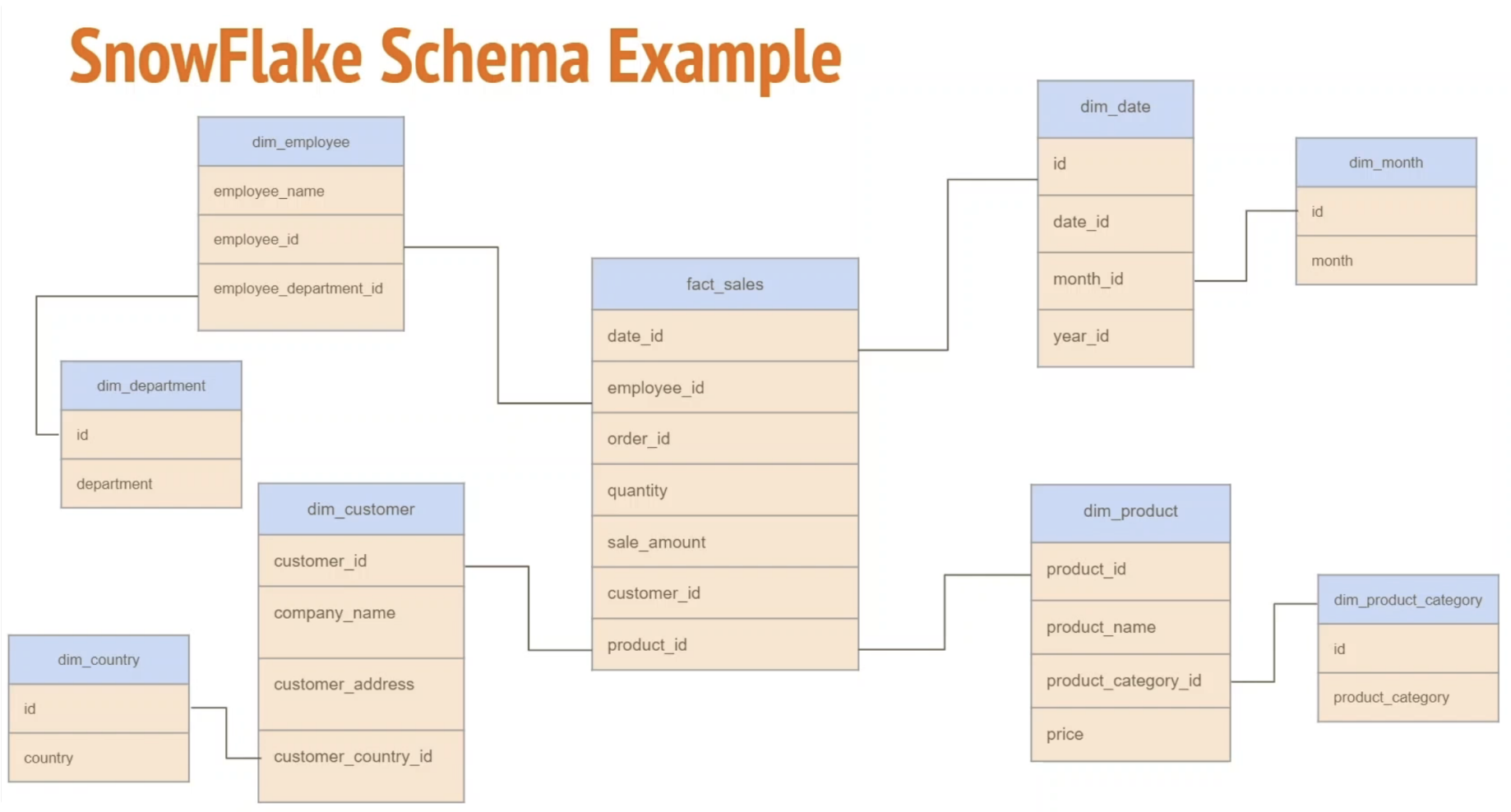
Pros	Cons
Easier to query and gives better performance compared to a normalized model.	Data integrity is not enforced because it is de-normalized.
Simplified business reporting logic.	Does not inform many to many relationships.
Better performing queries.	Dependent on business process.

Snowflake Schema

Similar to star schema as it used the base structure, but the dimension tables are further normalized. It normalizes the denormalized table in a star schema. By doing this our dimension hierarchy becomes its own dimension tables, similar to how we would have done it in a normalized model.
Each dimension table would have its own one or more lookup tables.

Pros	Cons
Improved data quality as data is more structured due to normalization.	Requires for more complex joins due to normalized queries.
Uses less storage space than a denormalized schema.	Increased number of joins can also potentially impact performance.
Suitable for data with deep hierarchies because we don’t need to de-normalize our dimensions.	Level of integrity is still lower than a highly normalized schema.
Easier to design and develop.	Difficult for business users to understand.

Snowflake Schema Example



In conclusion, a star schema is denormalized, requires more storage, is faster, has redundant data, and is better suited for data marts.

A snowflake schema is normalized on top of denormalization, requires less storage, is slower, doesn’t have redundant data, and is better suited for a data warehouse.

Building Dimensional Model with BigQuery & dbt (Hands-On)

We will be building a dimensional data warehouse using BigQuery as our data warehouse and dbt as our data transformation tool.

The database we're using is from Northwind Traders. It's an OLTP database sample created by Microsoft to demonstrate the features of some of its products and for training and tutorial purposes. It contains all the sales data for Northwind Traders, a fictitious specialty foods export-import company. We will be using the MySQL version of Northwind sample database.

Before we start building we need to understand how we're going to approach this from a high-level perspective.

First, we need to understand, `what is our use-case`? Then we need to `gather requirements` to understand business requirements and what we are trying to achieve. Then, we will decide on the `architectural design`. After this, we will start our process of `dimensional modelling` where we will build a high level model of a dimensional mode. Finally, using our dimensional model we will create a `physical design` and `develop a SQL transformation code`, creating a `physical dimensional model` at last.

Use-Case

- What is it we are trying to do?
- Why are we doing it?
- How are we going to achieve it?

`What`

- Northwind Traders are an export/import company who trade specialty foods around the world.
- An initial conversation with the team tells us that their existing architecture is a mix of on-premise and legacy systems, including their MySQL database which is the primary operational database.
- This is where all the sales transactions are recorded between company and customers.
- This same MySQL database is used to generate reports and build analytics solutions.
- They are struggling to keep up with reporting requirements and causing the database to slow down, impacting their day-to-day business.
- Northwind Traders want to modernize their data and reporting solutions and finally move away from on-prem.

`Why`

They want to modernize their existing infrastructure for…

- Better scalability.
- Reduced load on operational systems.
- Improved reporting speed.
- Improved data security.

`How`

- Northwind Traders will migrate their existing database system to the Google Cloud Platform (GCP).
- MySQL On-Prem will be replaced with a fully managed Cloud SQL for MySQL.
- For reporting solutions, we will build a data warehouse on GCP using BigQuery.
- BigQuery will become our main OLAP solution which will support all the reporting requirements.
- Our data warehouse will be a dimensional data warehouse, with dim and fact tables, loosely based on Kimball's approach.

Requirement Gathering

At this point we need to…

- Define business processes (we are loosely following Kimball here).
- Conduct data profiling (data exploration) to understand what our data looks like and to measure the complexity of the project.
- Once we have some understanding of the requirements and tables involved, we can start creating a Bus Matrix with some high level entities.
- During this phase we will also create a Naming Convention Document which will allow us to set standards as early as possible.
- The output of this step is our Conceptual Model which we can use to communicate across the business.

Define Business Processes

During this stage we would have interviews with the business and stakeholders to identify the following reporting requirements.

1. `Sales Overview`
   Overall sales reports to understand customers better, what is being sold, what sells the most, where and what sales the least—a general overview of how the business is going.
2. `Sales Agent Tracking`
   Track sales and performance of each sales agent to adjust commissions, reward high achievers and empower low achievers.
3. `Product Inventory`
   Understand the current inventory levels, how to improve stock management, what suppliers do we have and how much is being purchased. This will allow the business to understand stock management and potentially broker better deals with suppliers.
4. `Customer Reporting`
   Allow customers to understand their purchase orders, how much and when are they buying, empowering them to make data driven decisions and utilize the data to join to their sales data.

Data Profiling — Introduction

Here, we are trying to `understand the system and the data with simple data profiling such as running queries to understand data redundancies`, the complexity of the data, and the relationships between tables. This will help us form our conceptual model, and help speed up design solutions.

Data profiling can range from very simple to very complicated. For now we’re just going to be doing some basic profiling but know that in the future you can use some specialized tool.

You can start by…

1. Clicking on any table and seeing if the data types make sense. Ex. In our dataset, the ID column is nullable which isn’t correct since that will likely be our primary key and so it needs to be unique and always populated. So we need to make a note of that and how it could be a problem.
2. In the same table, take a look at how many rows it has to understand the sample size and make a note of this too.
3. Then, give the actual table a quick scan. Note that this is easier when your table only has 31 rows. Ex. In our dataset, the business_phone column is the same for every person and some columns are null which could be a problem. Make a note of this too.
4. Now we can run some simple queries, for example to check if the ID column is really unique by running a COUNT(DISTINCT).

A very helpful query to run is the one below…

```sql
SELECT
id as column_name,
sum(case when id is null then 0 else 1 end) / cast(count(*) as numeric) as not_null_proportion,
count(distinct id) / cast(count(*) as numeric) as distinct_proportion,
count(distinct id) as distinct_count,
count(distinct id) = count(*) as is_unique,
current_timestamp() as profiled_at
FROM `ae-bootcamp.dl_northwind.customer`
GROUP BY 1
ORDER BY 1
```

Processing location: europe-west2

Query results SAVE RESULTS EXPLORE DATA

Query complete (0.3 sec elapsed, 240 B processed)

Job information | Results | JSON | Execution details

Row	column_name	not_null_proportion	distinct_proportion	distinct_count	is_unique	profiled_at
1	1	1	0.5	1	false	2021-12-17 11:50:41.943387 UTC
2	2	1	1	1	true	2021-12-17 11:50:41.943387 UTC
3	3	1	1	1	true	2021-12-17 11:50:41.943387 UTC
4	4	1	1	1	true	2021-12-17 11:50:41.943387 UTC
5	5	1	1	1	true	2021-12-17 11:50:41.943387 UTC
6	6	1	1	1	true	2021-12-17 11:50:41.943387 UTC
7	7	1	1	1	true	2021-12-17 11:50:41.943387 UTC
8	8	1	1	1	true	2021-12-17 11:50:41.943387 UTC
9	9	1	1	1	true	2021-12-17 11:50:41.943387 UTC

This is because now we can look at this data and see which ID is unique and which ID isn't unique. Now we know exactly where the problem is. If you have hundreds of thousands of records, writing a query like this and putting the filter as is_unique = false will give you the non-unique records only.

So here we see that ID number 1 has duplicates which we will need to handle during our data transformation layer.

As the next step, now that we understand the 'customer' table, we have to repeat this same process for every single table that we think we're going to need to build our dimensional model.

To do this, we should look at the ERD diagram, and identify the tables which we think we would need. By looking quickly, you can tell that tables like 'customer', 'orders', 'products', and 'employees' are important.

Just remember data modelling is an iterative process, it's okay to go back and revise your decision making process and update your model. You won't ever get things 100% right on the first try.

Let's list out any potential issues we find in these big main tables, just at a first glance…

orders	products	employees

orders	products	employees
IDs are NULLABLE	ID is NULLABLE	ID is NULLABLE
ZIP post code is the same for every person.	Discontinued could be a Y/N column?	All phone numbers (business, home, fax) are the same for every person.
Shipping fee doesn't specify currency.	Category for mozzarella needs to be 'Dairy'	

Data Profiling — Completed

After doing our data profiling, we have identified the tables that we are going to need to continue building our dimensional data warehouse. So the existing tables in our OLTP system will be…

- `Customers`—a customer buys food from Northwind.
- `Employees`—people who work for Northwind who may also be customers.
- `Orders`—sales order transactions taking place between the customers and Northwind.
- `Order Details`—order details for the orders placed by customers.
- `Inventory Transaction`—transaction details of each inventory.
- `Products`—contains current Northwind products that customers can purchase.
- `Shippers`—ships orders from Northwind to customers.
- `Suppliers`—supplies Northwind with required items.
- `Invoices`—invoices are created for each order.

Usually once we've identified all the tables we'll be using, it's best to go back to the business and validate the choices, and then go back and work on the data to tweak your understanding of the data and the tables you've selected.

`At this stage we are doing a lot of exploratory work and guess work, it is very difficult to get everything right on the first try and so data modelling is a bit of an iterative approach.`

The next step from here would be to `identify the high level entities`. We do this so we can pick and choose, which entities will go into our bus matrix.

Analytics Engineering Workbook

In this bootcamp we have access to an Analytics Engineering workbook that will contain everything we will be covering in this hands-on tutorial, like the Bus Matrix High Level entities.

In real life there will also be a document of a similar form or structure that we'll be using as a guidebook.

Bus Matrix for NorthWind (High Level Entities)

In real life scenarios, again, note that it might take a while for you to get everything correct.

For example, take the **Sales Overview**. Overall, `sales` reports to understand better our `customers` what is `being sold`, what sells the most, where and `what sells` the least, the goal is to have a general overview of how the business is going.

These `keywords` give us an idea of what dimensions our sales overview report should contain.

So when building our Bus Matrix, we write down the business process (Sales Overview) and the high-level entities we identified earlier as columns.

Next, we `mark which entities we think would be required to build our business processes sales overview report`. Since we're interested in sales and what is being sold, we can assume that we need to have 'product' so we put an X there. Our orders table has quantities sold, so we know we need that as well. And of course the earlier description of

sales has keywords like ‘customers’ so we know we need that too. We also need ‘employees’ because some of our customers can be employees too and we also need some ‘supplier’ information as part of our reports.

We’ll also mark the `date dimension` as a requirement even though it wasn’t in our data lake dimensions earlier, but we’re adding it because without a date dimension, we won’t be able to create reports such as ‘last seven days,’ ‘last month,’ ‘last quarter sales’ etc.

Business Process	product	orders	employees	customers	suppliers	date
Sales Overview	X	X	X	X	X	X

Now let’s do the same thing for **Sales Agent Tracking**. The main purpose of this report is to track `sales` and `performance` of each `sales agent` to adjust commissions, reward high achievers, and empower the rest.

Business Process	product	orders	employees	customers	suppliers	date
Sales Agents	X	X	X	X	X	X

Generally in this scenario we need to know about ‘employees,’ understand sales performance so that’s ‘order’ related, and ‘customers’ are another dimension which will indicate our performance measures. This is similar to our last Bus Matrix so we know we’ll likely need all of these dimensions.

`Whenever you’re building a Bus Matrix, it’s super helpful to go back to your Business Processes list, and look at what each of the processes entails, what are the details (keywords) you can use to build the Bus Matrix.`

Once we have the Bus Matrix we can move on to building our conceptual model.

Conceptual Model

If we look at our Bus Matrix, we can see there are three distinct business processes—sales, inventory, and customer reporting (more to do with understanding purchase orders).

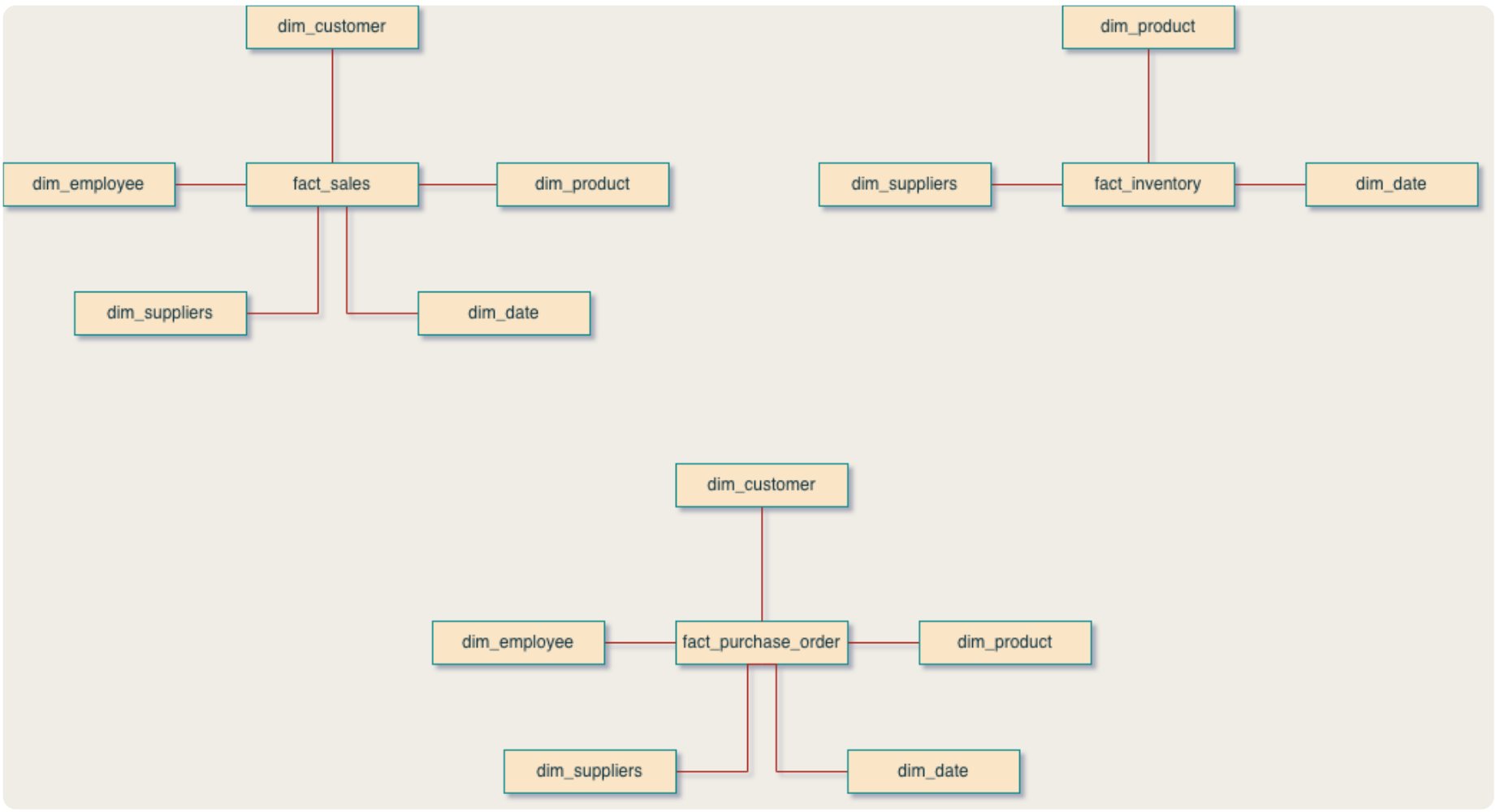
Based on our understanding of the table and the schema and our ERD model, we make a few assumptions about what high-level tables we need to build.

`To build our conceptual model we will be using Draw.io.` We will also be using this for our logical and physical models. So you want to make sure you’re saving everything correctly and creating backups of this file.

Completed Bus Matrix

Business Process	product	orders	employees	customers	suppliers	date
Sales Overview	X		X	X	X	X
Sales Agents	X		X	X	X	X
Product Inventory	X				X	X
Customer Reporting	X		X	X	X	X

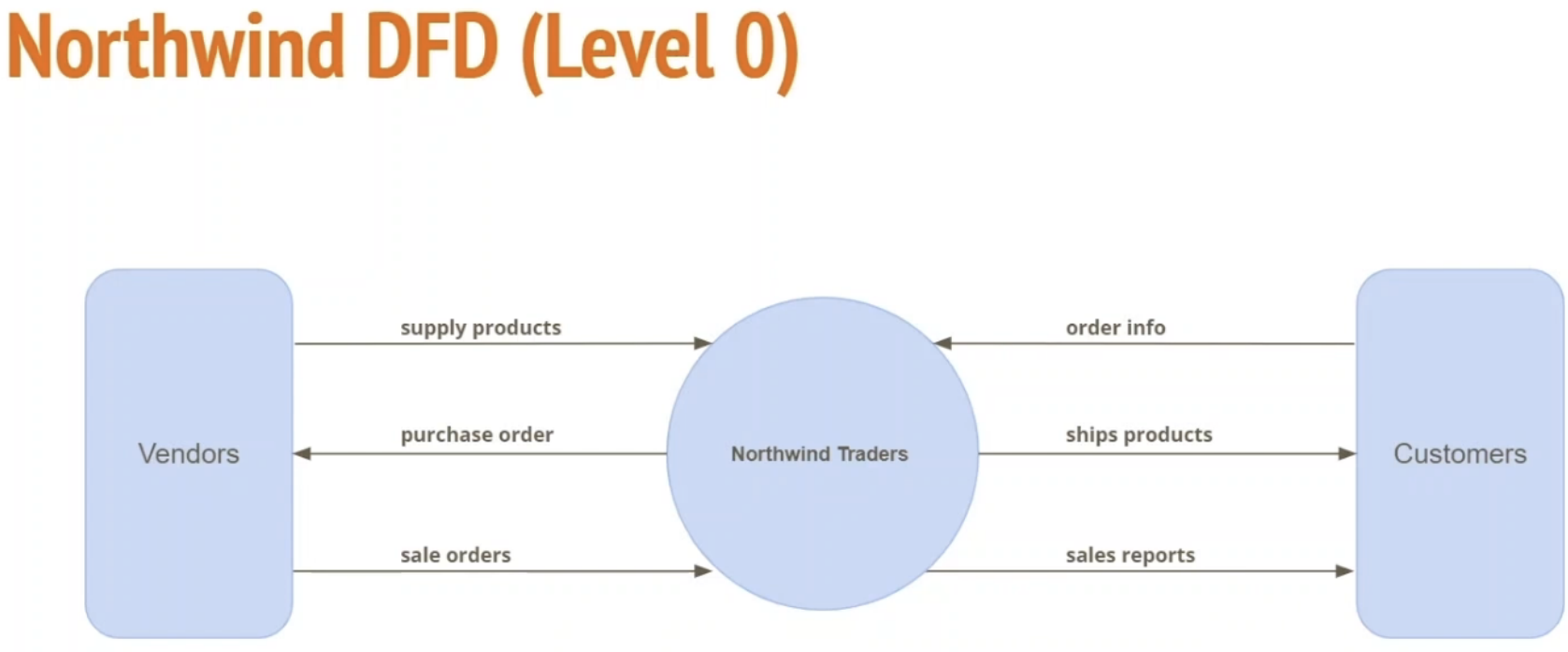
We start by referencing our Bus Matrix. We know we have the three distinct processes to do with sales, inventory, and purchase orders, so let’s create three entities.



Now we have built our Bus Matrix, and our high-level conceptual model.

Data Warehouse Architecture Design

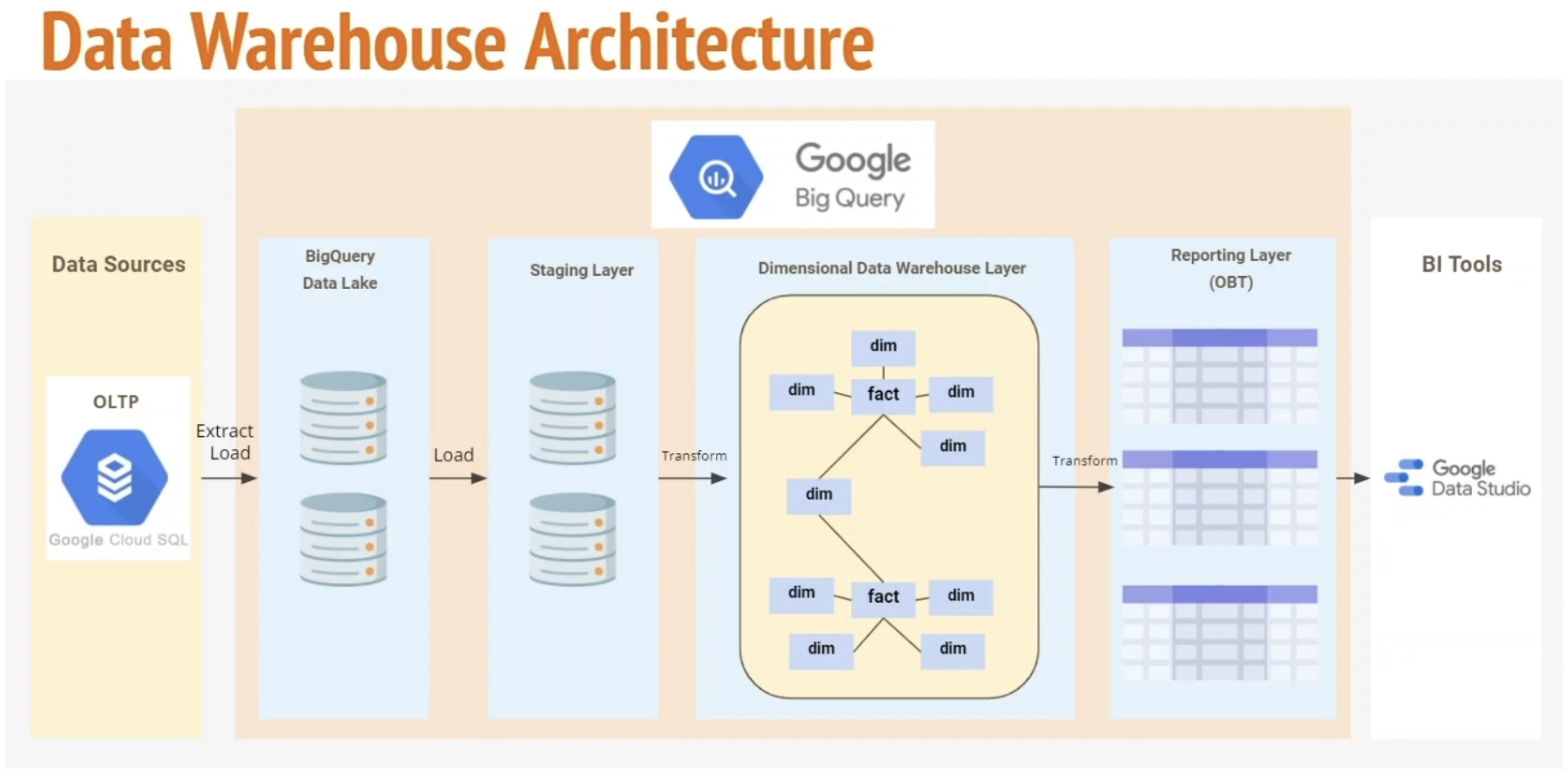
Let’s assume we have been working side by side with our data architect and they’ve created this Data Flow diagram.



We can use this to quickly understand how the data flows between different entities. But note that as the level increases, the diagram becomes more complicated. For now we’ll just stick to level 0 because it’s the easiest and quickest to understand.

Here we can see the relationships between Vendors, Northwind Traders, and Customers. We can see that vendors supply products to Northwind Traders when the company places a purchase order. Sales orders are also processed. A customer places an order and Northwind Traders receive the order info.

Let’s take a look at our data warehouse architecture…



We have our data sources (OLTP) and we can see that our team have already migrated our on-prem MySQL solution to Google Cloud SQL.

This data is then loaded to the BigQuery Data Lake layer. As we saw earlier when we were building our Bus Matrix, our data lake layer is pretty much a replication of our MySQL OLTP data sources. This is so that we don’t query our data sources directly, we have a layer in between where we can materialize our OLTP data which reduced the load on your transactional database systems.

Then this data is loaded to the staging layer where we will do some data transformations and cleansing.

Then, we will build our dim and fact models in our dimensional data warehouse layer.

Once our data is available in our dimensional data warehouse layer, we create a one big table (OTB) to connect our data to BI tools.

Connecting this to BI tools speeds up reporting and showcases how we would use different modelling styles in combination to build such solutions.

Dimensional Modelling

Previously we completed building our Bus Matrix, mapped out our High-Level Entities, and our Conceptual Model. Now, we will be using all of that to continue with our process of dimensional modelling.

We will…

1. Follow four steps of dimensional modelling to create our dimensional model.
2. Once this is completed we should have the grain of our Bus Matrix defined.
3. Based on this we will create a logical model.
4. Then we will create a source to target document.

4 Steps of Dimensional Modelling

1. Select the business process.
2. Declare the grain.
3. Identify the dimensions.
4. Identify the facts.

So take our Bus Matrix again, but notice that we have some new columns added to the table.

Business Process	Fact Tables	Fact Type of Grain	Granularity	Facts	product	orders	employees	customers	suppliers	date	F
Sales Overview					X		X	X	X	X	H
Sales Agents					X		X	X	X	X	M
Product Inventory					X				X	X	L
Customer Reporting					X		X	X	X	X	H

We also added a priority column, which are defined by our business, based on which reports will bring the highest value to the business first.

If you remember, we said we would be taking the previous Bus Matrix with High Level Entities back to the business, and discussing which reports to build first. The outcome of that process is that table we are seeing here with the new columns.

Bus Matrix Detailed

At this point we would clarify what tables, grain, granularity, and facts exist for each of our business processes.

Business Process	Fact Tables	Fact Type of Grain	Granularity	Facts	product	orders	employees	customers	s
Sales Overview	fact_sales	Transaction	One Row per order detail line item.	Refer Dimensional Model (Attributes and Measures)	X		X	X	X
Sales Agents					X		X	X	X
Product Inventory					X				X
Customer Reporting					X		X	X	X

For the facts column, this is easier to do as part of your source to target mapping document which will act as the main reference document between developers and it will be critical to use during the implementation phase.

Source to Target Mapping (Source to BigQuery Data Lake)

In order to understand how to do this, we should reference our Data Warehouse Architecture diagram.

We know the process goes, Cloud SQL to Data Lake, Data Lake to Staging Layer, Staging Layer to Dimensional Data Warehouse Layer (dim and fact tables), and that layer to Reporting Layer (OBT). So we need to create four source to target documents.

So first of all, source to target mapping basically allows us to understand where the data is coming from and what it will become once it lands at its destination. Because during each transfer of data, data transformation can happen, and we need to keep track of all of these transformations, which is why this source to target mapping document will be a reference point for that.

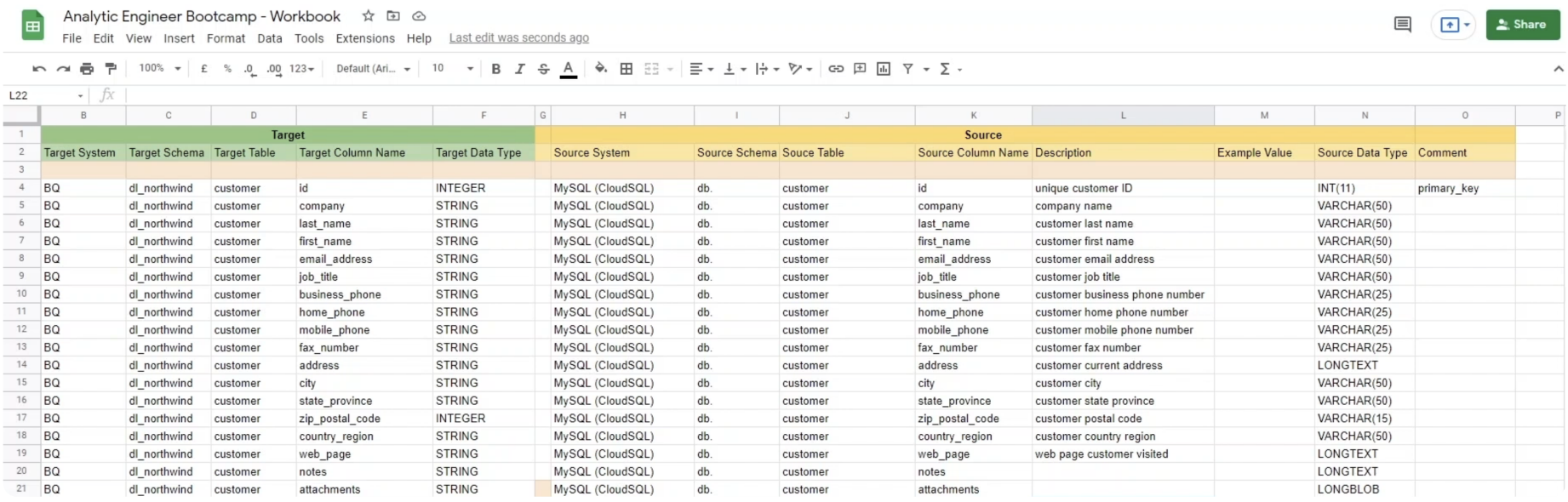
The easiest way to start populating our source to target table is by looking at our BigQuery tables. Note: the table itself lists ‘Target’ first because when we’re actually

building the tables, it is helpful to see target first.

So the tables we now have in BigQuery are our Data Lake datasets, and they're a reflection of what's in our OLTP or Cloud SQL database. It is also helpful to frequently reference the ERD diagram because it is a representation of what the data looks like in the Cloud SQL environment.

We can start by looking at the 'customer' table. If we look at the schema we can see all the columns that we need which is convenient since we can just copy and paste those into the source document. This is the easiest way to do things since currently we don't have a MySQL instance running, we are basically replicating what it will look liken in reality. Usually we would take this directly from our OLTP system.

So then we can populate our table…



Analytic Engineer Bootcamp - Workbook

File Edit View Insert Format Data Tools Extensions Help Last edit was seconds ago

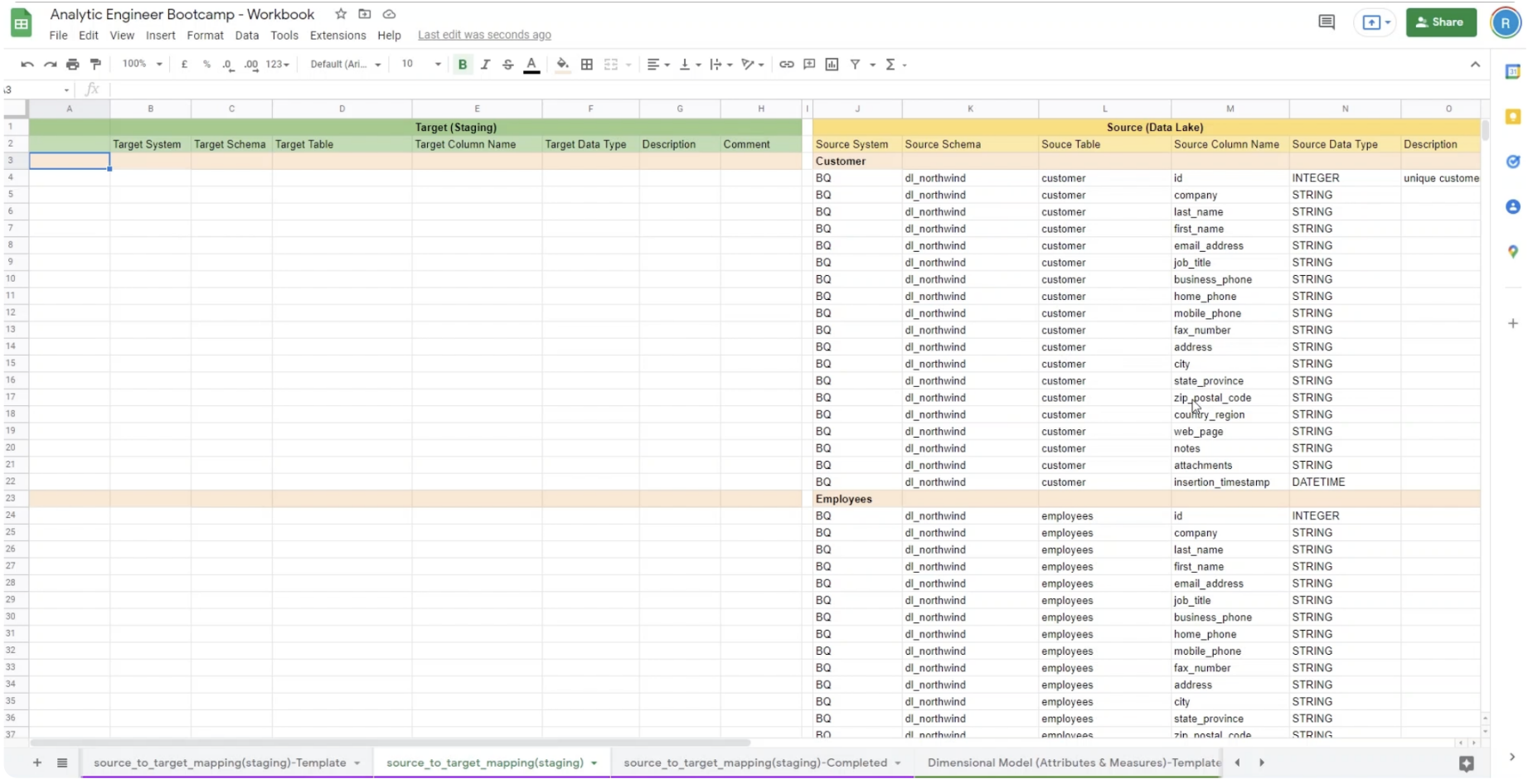
	Target					Source							
	Target System	Target Schema	Target Table	Target Column Name	Target Data Type	Source System	Source Schema	Souce Table	Source Column Name	Description	Example Value	Source Data Type	Comment
	BQ	dl_northwind	customer	id	INTEGER	MySQL (CloudSQL)	db.	customer	id	unique customer ID		INT(11)	primary_key
	BQ	dl_northwind	customer	company	STRING	MySQL (CloudSQL)	db.	customer	company	company name		VARCHAR(50)	
	BQ	dl_northwind	customer	last_name	STRING	MySQL (CloudSQL)	db.	customer	last_name	customer last name		VARCHAR(50)	
	BQ	dl_northwind	customer	first_name	STRING	MySQL (CloudSQL)	db.	customer	first_name	customer first name		VARCHAR(50)	
	BQ	dl_northwind	customer	email_address	STRING	MySQL (CloudSQL)	db.	customer	email_address	customer email address		VARCHAR(50)	
	BQ	dl_northwind	customer	job_title	STRING	MySQL (CloudSQL)	db.	customer	job_title	customer job title		VARCHAR(50)	
	BQ	dl_northwind	customer	business_phone	STRING	MySQL (CloudSQL)	db.	customer	business_phone	customer business phone number		VARCHAR(25)	
	BQ	dl_northwind	customer	home_phone	STRING	MySQL (CloudSQL)	db.	customer	home_phone	customer home phone number		VARCHAR(25)	
	BQ	dl_northwind	customer	mobile_phone	STRING	MySQL (CloudSQL)	db.	customer	mobile_phone	customer mobile phone number		VARCHAR(25)	
	BQ	dl_northwind	customer	fax_number	STRING	MySQL (CloudSQL)	db.	customer	fax_number	customer fax number		VARCHAR(25)	
	BQ	dl_northwind	customer	address	STRING	MySQL (CloudSQL)	db.	customer	address	customer current address		LONGTEXT	
	BQ	dl_northwind	customer	city	STRING	MySQL (CloudSQL)	db.	customer	city	customer city		VARCHAR(50)	
	BQ	dl_northwind	customer	state_province	STRING	MySQL (CloudSQL)	db.	customer	state_province	customer state province		VARCHAR(50)	
	BQ	dl_northwind	customer	zip_postal_code	INTEGER	MySQL (CloudSQL)	db.	customer	zip_postal_code	customer postal code		VARCHAR(15)	
	BQ	dl_northwind	customer	country_region	STRING	MySQL (CloudSQL)	db.	customer	country_region	customer country region		VARCHAR(50)	
	BQ	dl_northwind	customer	web_page	STRING	MySQL (CloudSQL)	db.	customer	web_page	web page customer visited		LONGTEXT	
	BQ	dl_northwind	customer	notes	STRING	MySQL (CloudSQL)	db.	customer	notes			LONGTEXT	
	BQ	dl_northwind	customer	attachments	STRING	MySQL (CloudSQL)	db.	customer	attachments			LONGBLOB	

Here we've added what the source data types should be from our ERD diagram, identified the primary key, and filled in the columns. We have identified the source system, and for now put in db. for the source schema as we don't know what it will be.

We would then fill out this table for every table in our ERD diagram.
Check Analytics Engineering (Bootcamp) Document

Source to Target Mapping (BigQuery Data Lake to Staging)

We are now at the source to target mapping for the staging layer. There is a template for this in the Analytics Engineering workbook too, but generally when you're doing source to target mapping and moving one step after another, it is best to duplicate your completed version, and then do the mapping from there. We're now doing the mapping from our dat lake layer to our staging layer, meaning the 'Target' column becomes our 'Source.' Meaning you can copy and paste that entire section and leave it as is because we've done this already. So we start with this…



Analytic Engineer Bootcamp - Workbook

File Edit View Insert Format Data Tools Extensions Help Last edit was seconds ago

	Target (Staging)							Source (Data Lake)					
	Target System	Target Schema	Target Table	Target Column Name	Target Data Type	Description	Comment	Source System	Source Schema	Souce Table	Source Column Name	Source Data Type	Description
								Customer					
								BQ	dl_northwind	customer	id	INTEGER	unique custome
								BQ	dl_northwind	customer	company	STRING	
								BQ	dl_northwind	customer	last_name	STRING	
								BQ	dl_northwind	customer	first_name	STRING	
								BQ	dl_northwind	customer	email_address	STRING	
								BQ	dl_northwind	customer	job_title	STRING	
								BQ	dl_northwind	customer	business_phone	STRING	
								BQ	dl_northwind	customer	home_phone	STRING	
								BQ	dl_northwind	customer	mobile_phone	STRING	
								BQ	dl_northwind	customer	fax_number	STRING	
								BQ	dl_northwind	customer	address	STRING	
								BQ	dl_northwind	customer	city	STRING	
								BQ	dl_northwind	customer	state_province	STRING	
								BQ	dl_northwind	customer	zip_postal_code	STRING	
								BQ	dl_northwind	customer	country_region	STRING	
								BQ	dl_northwind	customer	web_page	STRING	
								BQ	dl_northwind	customer	notes	STRING	
								BQ	dl_northwind	customer	attachments	STRING	
								BQ	dl_northwind	customer	insertion_timestamp	DATETIME	
								Employees					
								BQ	dl_northwind	employees	id	INTEGER	
								BQ	dl_northwind	employees	company	STRING	
								BQ	dl_northwind	employees	last_name	STRING	
								BQ	dl_northwind	employees	first_name	STRING	
								BQ	dl_northwind	employees	email_address	STRING	
								BQ	dl_northwind	employees	job_title	STRING	
								BQ	dl_northwind	employees	business_phone	STRING	
								BQ	dl_northwind	employees	home_phone	STRING	
								BQ	dl_northwind	employees	mobile_phone	STRING	
								BQ	dl_northwind	employees	fax_number	STRING	
								BQ	dl_northwind	employees	address	STRING	
								BQ	dl_northwind	employees	city	STRING	
								BQ	dl_northwind	employees	state_province	STRING	
								BQ	dl_northwind	employees	zip_postal_code	STRING	

source_to_target_mapping(staging)-Template | source_to_target_mapping(staging) | source_to_target_mapping(staging)-Completed | Dimensional Model (Attributes & Measures)-Template

At this stage it’s important to note that the source to target document is not a static document, you will be revisiting it and making updates frequently as you discover more things throughout your development process.

In this case we can copy and paste the source and the target in the staging layer to look the same, because we are happy with the column names and data types so the only thing that actually needs to be changed is the ‘Target Schema’ and ‘Target Table’ Instead of dl_northwind it would be stg_northwind (staging_northwind). This naming convention is very important because it helps us identify during our development process, which tables are we working together with. We’ve also added a column name into our Source(Data Lake) called insertion_timestamp so we can track when the data was loaded into our source systems. Because we added this column, we would then need to add it into our Target(Staging Layer) too. Our final version looks like this…

Analytic Engineer Bootcamp - Workbook

File Edit View Insert Format Data Tools Extensions Help Last edit was seconds ago

Share

		Target (Staging)							Source (Data Lake)					
	Target System	Target Schema	Target Table	Target Column Name	Target Data Type	Description	Comment	Source System	Source Schema	Souce Table	Source Column Name	Source Data Type	Description
Customer								Customer					
1	BQ	stg_northwind	stg_customer	id	INTEGER	unique customer ID		BQ	dl_northwind	customer	id	INTEGER	unique custome
2	BQ	stg_northwind	stg_customer	company	STRING			BQ	dl_northwind	customer	company	STRING	
3	BQ	stg_northwind	stg_customer	last_name	STRING			BQ	dl_northwind	customer	last_name	STRING	
4	BQ	stg_northwind	stg_customer	first_name	STRING			BQ	dl_northwind	customer	first_name	STRING	
5	BQ	stg_northwind	stg_customer	email_address	STRING			BQ	dl_northwind	customer	email_address	STRING	
6	BQ	stg_northwind	stg_customer	job_title	STRING			BQ	dl_northwind	customer	job_title	STRING	
7	BQ	stg_northwind	stg_customer	business_phone	STRING			BQ	dl_northwind	customer	business_phone	STRING	
8	BQ	stg_northwind	stg_customer	home_phone	STRING			BQ	dl_northwind	customer	home_phone	STRING	
9	BQ	stg_northwind	stg_customer	mobile_phone	STRING			BQ	dl_northwind	customer	mobile_phone	STRING	
10	BQ	stg_northwind	stg_customer	fax_number	STRING			BQ	dl_northwind	customer	fax_number	STRING	
11	BQ	stg_northwind	stg_customer	address	STRING			BQ	dl_northwind	customer	address	STRING	
12	BQ	stg_northwind	stg_customer	city	STRING			BQ	dl_northwind	customer	city	STRING	
13	BQ	stg_northwind	stg_customer	state_province	STRING			BQ	dl_northwind	customer	state_province	STRING	
14	BQ	stg_northwind	stg_customer	zip_postal_code	STRING			BQ	dl_northwind	customer	zip_postal_code	STRING	
15	BQ	stg_northwind	stg_customer	country_region	STRING			BQ	dl_northwind	customer	country_region	STRING	
16	BQ	stg_northwind	stg_customer	web_page	STRING			BQ	dl_northwind	customer	web_page	STRING	
17	BQ	stg_northwind	stg_customer	notes	STRING			BQ	dl_northwind	customer	notes	STRING	
18	BQ	stg_northwind	stg_customer	attachments	STRING			BQ	dl_northwind	customer	attachments	STRING	
19	BQ	stg_northwind	stg_customer	insertion_timestamp	DATETIME			BQ	dl_northwind	customer	insertion_timestamp	DATETIME	
Employees								Employees					
1	BQ	stg_northwind	stg_employees	id	INTEGER			BQ	dl_northwind	employees	id	INTEGER	
2	BQ	stg_northwind	stg_employees	company	STRING			BQ	dl_northwind	employees	company	STRING	
3	BQ	stg_northwind	stg_employees	last_name	STRING			BQ	dl_northwind	employees	last_name	STRING	
4	BQ	stg_northwind	stg_employees	first_name	STRING			BQ	dl_northwind	employees	first_name	STRING	
5	BQ	stg_northwind	stg_employees	email_address	STRING			BQ	dl_northwind	employees	email_address	STRING	
6	BQ	stg_northwind	stg_employees	job_title	STRING			BQ	dl_northwind	employees	job_title	STRING	
7	BQ	stg_northwind	stg_employees	business_phone	STRING			BQ	dl_northwind	employees	business_phone	STRING	
8	BQ	stg_northwind	stg_employees	home_phone	STRING			BQ	dl_northwind	employees	home_phone	STRING	
9	BQ	stg_northwind	stg_employees	mobile_phone	STRING			BQ	dl_northwind	employees	mobile_phone	STRING	
10	BQ	stg_northwind	stg_employees	fax_number	STRING			BQ	dl_northwind	employees	fax_number	STRING	
11	BQ	stg_northwind	stg_employees	address	STRING			BQ	dl_northwind	employees	address	STRING	
12	BQ	stg_northwind	stg_employees	city	STRING			BQ	dl_northwind	employees	city	STRING	
13	BQ	stg_northwind	stg_employees	state_province	STRING			BQ	dl_northwind	employees	state_province	STRING	
14	BQ	stg_northwind	stg_employees	zip_postal_code	STRING			BQ	dl_northwind	employees	zip_postal_code	STRING	

source_to_target_mapping(staging)-Template | source_to_target_mapping(staging) | source_to_target_mapping(staging)-Completed | Dimensional Model (Attributes & Measures)-Template | Explore

At this point things are gonna get a little more complex because it won’t be completely one-to-one mapping between sources and targets. Earlier we created that Bus Matrix, identified grains, and which facts and dimensions we needed. Instead of defining our dimensions and facts in detail, we mentioned that we will be doing this during our source to target mapping process.

Dimensional Model (Attributes and Measures)

Let’s take a look at our completed Bus Matrix again…

Business Process	Fact Tables	Fact Type of Grain	Granularity	Facts	product	employee	customer	suppliers	date	Priority
Sales Overview	fact_sales	Transaction	One Row per order / detail line item	refer: Dimensional Model (Attributes & Measures)	x	x	x	x	x	High
Sales Agent	fact_sales	Transaction	One Row per order / detail line item	refer: Dimensional Model (Attributes & Measures)	x	x	x	x	x	Medium
Product Inventory	fact_inventory	Transaction	One Row per inventory transaction	refer: Dimensional Model (Attributes & Measures)	x			x	x	Low
Customer Reporting	fact_purchase_order	Transaction	One Row per order /detail line item	refer: Dimensional Model (Attributes & Measures)	x	x	x	x	x	High

First we need to create a dimensional model attributes and measures document, then once we have this we create our source to target mapping for the dimensional database layer.

The priority according to our Bus Matrix is the Sales Overview Report because it’s ranked as High so we will focus on getting facts and dimensions for this report.

So let’s start with our customer table. Observing the table, we can use all these fields as our dimensions, so let’s make a note of that in our workbook. Now we know we’re happy with the content we have in our customer table, we can populate our table and call it dim_customer. Then let’s bring in all those fields we saw earlier by looking at the table schema in BigQuery.

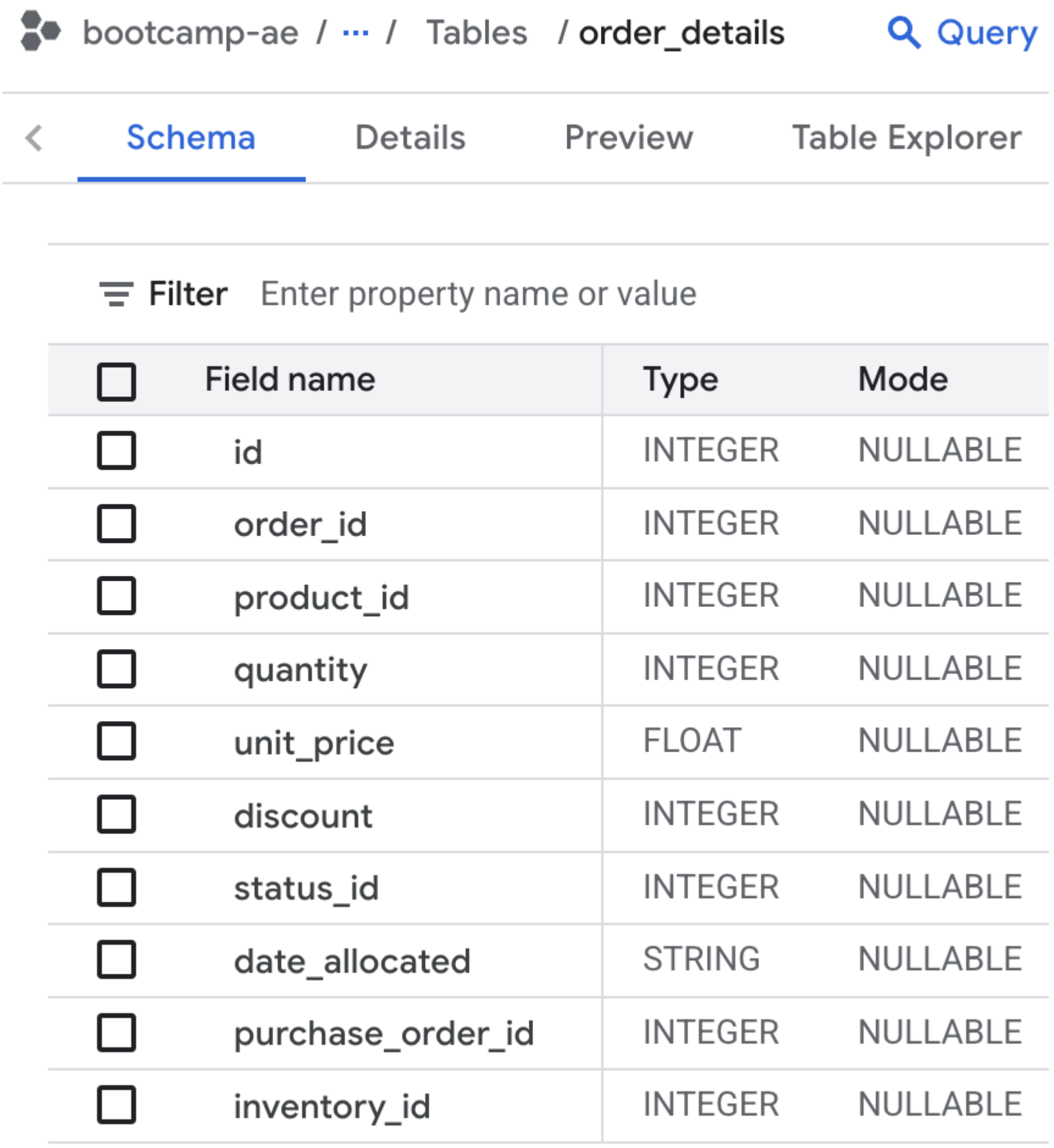
Column Position	Table Name	Column	Data Type	Mode	Key	Description
1	dim_customer	unique_customer_id	INTEGER	NO	PK	
2	dim_customer	company	STRING	NULLABLE		
3	dim_customer	last_name	STRING	NULLABLE		
4	dim_customer	first_name	STRING	NULLABLE		
5	dim_customer	email_address	STRING	NULLABLE		
6	dim_customer	job_title	STRING	NULLABLE		
7	dim_customer	business_phone	STRING	NULLABLE		
8	dim_customer	home_phone	STRING	NULLABLE		
9	dim_customer	mobile_phone	STRING	NULLABLE		
10	dim_customer	fax_number	STRING	NULLABLE		
11	dim_customer	address	STRING	NULLABLE		
12	dim_customer	city	STRING	NULLABLE		
13	dim_customer	state_province	STRING	NULLABLE		
14	dim_customer	zip_postal_code	STRING	NULLABLE		
15	dim_customer	country_region	STRING	NULLABLE		
16	dim_customer	web_page	STRING	NULLABLE		
17	dim_customer	notes	STRING	NULLABLE		
18	dim_customer	attachments	STRING	NULLABLE		

What we’ve done is update the table name to dim_customer, and added a new column ‘Mode’ which is important because it’s where we’ll be telling our system if a field can be empty or not. In this case we want to ensure the unique_customer_id column is always populated since it’s a primary key.

According to our Bus Matrix, the next one in our list is the ‘employees’ table. So we would repeat the process again. Go back to BigQuery, select ‘employees’ and look at the table schema. For us, the employees table has an almost identical structure to the ‘customers’ table. The process is the same as before. Bring in the columns, update the ID from id to unique_employee_id, make it not nullable.

As the next step, let’s create a fact table. According to our Bus Matrix we need to create a fact_sales table which is going to be based on one row per order detail. Let’s have a look at order_details then in BigQuery. By looking there, we see we have the foreign key

that we need and the measures such as quantity and unit_price. Let’s keep all of that.



But we need more. If we look at our dimensions table we know we have the dim_employee and dim_customer, which there isn’t a key for in our order_details table. This is also noticeable in our ERD diagram where we have our customer and employees table, but those don’t exist as foreign keys inside the order_details table. We need to join our order_details table to orders to access our customers and employees information. So we need to bring these fields into our fact table, so let’s add all of that there.

Column Position	Table Name	Column	Data Type	Mode	Key	Description
1	fact_sales	order_id	INTEGER	NO		
2	fact_sales	product_id	INTEGER	NO		
3	fact_sales	customer_id	INTEGER	NO		
4	fact_sales	employee_id	INTEGER	NO		
5	fact_sales	shipper_id	INTEGER	NO		
6	fact_sales	quantity	FLOAT	NULLABLE		
7	fact_sales	unit_price	FLOAT	NULLABLE		
8	fact_sales	discount	FLOAT	NULLABLE		
9	fact_sales	status_id	INTEGER	NO		

Column Position	Table Name	Column	Data Type	Mode	Key	Description
10	fact_sales	date_allocated	DATETIME	NULLABLE		
11	fact_sales	purchase_order_id	INTEGER	NO		
12	fact_sales	inventory_id	INTEGER	NO		
13	fact_sales	order_date	DATETIME	NULLABLE		
14	fact_sales	shipped_date	DATETIME	NULLABLE		
15	fact_sales	paid_date	DATETIME	NULLABLE		
16	fact_sales	comments	STRING	NULLABLE		

So now we've successfully created the fact_sales table and we have all the foreign keys that are associated with it. We have also updated the 'Mode' type to 'NO' for all the foreign keys because we can't have them empty.

We would then repeat this process for all the remaining dimensions and fact tables.

Importantly, neither in the ERD diagram or BigQuery do we have a date table, it doesn't exist. This is because a dim_table is something we'll have to create by ourselves by running a SQL query in BigQuery. We can use BigQuery's built-in generate date array function to populate the dim_date table.

```sql
SELECT
  FORMAT_DATE('%F', d) as id,
  d AS full_date,
  EXTRACT(YEAR FROM d) AS year,
  EXTRACT(WEEK FROM d) AS year_week,
  EXTRACT(DAY FROM d) AS year_day,
  EXTRACT(YEAR FROM d) AS fiscal_year,
  FORMAT_DATE('%Q', d) AS fiscal_qtr,
  EXTRACT(MONTH FROM d) AS month,
  FORMAT_DATE('%B', d) AS month_name,
  FORMAT_DATE('%W', d) AS week_day,
  FORMAT_DATE('%A', d) AS day_name,
  (CASE WHEN FORMAT_DATE('%A', d) IN ('Sunday', 'Saturday') THEN 0 ELSE 1 END) AS day_is_weekday,
FROM (
  SELECT
  *
  FROM
    UNNEST (GENERATE_DATE_ARRAY('2014-01-01', '2050-01-01', INTERVAL 1 DAY)) AS d)
```

Query completed

Query results

Save results Open in

Job information | Results | Visualization | JSON | Execution details | Execution graph

Row	id	full_date	year	year_week	year_day	fiscal_year	fiscal_qtr	month	month_name	week_day	day_name	day_is_weekday
1	2014-01-01	2014-01-01	2014	0	1	2014	1	1	January	00	Wednesday	1
2	2014-01-02	2014-01-02	2014	0	2	2014	1	1	January	00	Thursday	1
3	2014-01-03	2014-01-03	2014	0	3	2014	1	1	January	00	Friday	1
4	2014-01-04	2014-01-04	2014	0	4	2014	1	1	January	00	Saturday	0
5	2014-01-05	2014-01-05	2014	1	5	2014	1	1	January	00	Sunday	0
6	2014-01-06	2014-01-06	2014	1	6	2014	1	1	January	01	Monday	1
7	2014-01-07	2014-01-07	2014	1	7	2014	1	1	January	01	Tuesday	1
8	2014-01-08	2014-01-08	2014	1	8	2014	1	1	January	01	Wednesday	1
9	2014-01-09	2014-01-09	2014	1	9	2014	1	1	January	01	Thursday	1
10	2014-01-10	2014-01-10	2014	1	10	2014	1	1	January	01	Friday	1
11	2014-01-11	2014-01-11	2014	1	11	2014	1	1	January	01	Saturday	0

This query looks complicated but it's not. All it's doing is creating a date_array using this function, which creates a bunch of date values based on start and end date. Once we have the data from this SELECT date query, we're using a bunch of data formatting to create the date_column fields that we need. So now we can go populate our workbook with dim_date fields.

Column Position	Table Name	Column	Data Type	Mode	Key	Description	
1	dim_date	id	STRING	NO			
2	dim_date	full_date	DATE	NULLABLE			
3	dim_date	year	INTEGER	NULLABLE			
4	dim_date	year_week	INTEGER	NULLABLE			

Column Position	Table Name	Column	Data Type	Mode	Key	Description	
5	dim_date	year_day	INTEGER	NULLABLE			
6	dim_date	fiscal_year	INTEGER	NULLABLE			
7	dim_date	fiscal_qtr	STRING	NULLABLE			
8	dim_date	month	INTEGER	NULLABLE			
9	dim_date	month_name	STRING	NULLABLE			
10	dim_date	week_day	STRING	NULLABLE			
11	dim_date	day_name	STRING	NULLABLE			

We would then write up all the dimension and fact tables for the rest of our tables. Check Analytics Engineering (Bootcamp) Document

Then using the complete version of the dimension and fact tables, we're going to be creating our source to target mapping for the data warehouse layer. Because so far we have only created what fields we want from our dim and fact models. For example, we have dim_customer which only tells us all the columns that are going to be part of the dim_customer table, but it doesn't tell us where this column comes from.

Source to Target Mapping (BigQuery Staging Layer to Data Warehouse)

This document should look familiar by now. On the left we have the target which is our Data Warehouse and on the right we have the source, which is our Staging Layer. Since we have this Staging Layer filled out from earlier when it was our target, we can take it and copy paste it directly into the source section of the table.

Now we can go back to our dimensional model with attributes and measures and copy what's there, pasting it into the target section of the source to target mapping table. All we're adding here is the target schema which, say for customer, will be dwh_northwind (data_warehouse_northwind). We'll also add in our insertion_timestamp, however it's important to note that that's generated at the point of insertion so to reduce confusion, we need to get rid of that insertion_timestamp row from the source since that's misleading. Instead we'll just add a comment and say 'updated by transformation tool.'

Target (Dimensional Data Warehouse)							Source (Staging)						
Target System	Target Schema	Target Table	Target Column Name	Target Data Type	Description	Comment	Source System	Source Schema	Souce Table	Source Column Name	Source Data Type	Description	Comment
BQ	dwh_northwind	dim_customer	customer_id	INTEGER	PK		BQ	stg_northwind	stg_customer	id	INTEGER	unique customer ID	
BQ	dwh_northwind	dim_customer	company	STRING			BQ	stg_northwind	stg_customer	company	STRING		
BQ	dwh_northwind	dim_customer	last_name	STRING			BQ	stg_northwind	stg_customer	last_name	STRING		
BQ	dwh_northwind	dim_customer	first_name	STRING			BQ	stg_northwind	stg_customer	first_name	STRING		
BQ	dwh_northwind	dim_customer	email_address	STRING			BQ	stg_northwind	stg_customer	email_address	STRING		
BQ	dwh_northwind	dim_customer	job_title	STRING			BQ	stg_northwind	stg_customer	job_title	STRING		
BQ	dwh_northwind	dim_customer	business_phone	STRING			BQ	stg_northwind	stg_customer	business_phone	STRING		
BQ	dwh_northwind	dim_customer	home_phone	STRING			BQ	stg_northwind	stg_customer	home_phone	STRING		
BQ	dwh_northwind	dim_customer	mobile_phone	STRING			BQ	stg_northwind	stg_customer	mobile_phone	STRING		
BQ	dwh_northwind	dim_customer	fax_number	STRING			BQ	stg_northwind	stg_customer	fax_number	STRING		
BQ	dwh_northwind	dim_customer	address	STRING			BQ	stg_northwind	stg_customer	address	STRING		
BQ	dwh_northwind	dim_customer	city	STRING			BQ	stg_northwind	stg_customer	city	STRING		
BQ	dwh_northwind	dim_customer	state_province	STRING			BQ	stg_northwind	stg_customer	state_province	STRING		
BQ	dwh_northwind	dim_customer	zip_postal_code	STRING			BQ	stg_northwind	stg_customer	zip_postal_code	STRING		
BQ	dwh_northwind	dim_customer	country_region	STRING			BQ	stg_northwind	stg_customer	country_region	STRING		
BQ	dwh_northwind	dim_customer	web_page	STRING			BQ	stg_northwind	stg_customer	web_page	STRING		
BQ	dwh_northwind	dim_customer	notes	STRING			BQ	stg_northwind	stg_customer	notes	STRING		
BQ	dwh_northwind	dim_customer	attachments	STRING			BQ	stg_northwind	stg_customer	attachments	STRING		
BQ	dwh_northwind	dim_customer	insertion_timestamp	DATETIME		Updated by transformation tool.							
BQ	dwh_northwind	dim_employee	employee_id	INTEGER	PK		BQ	stg_northwind	stg_employees	id	INTEGER		
BQ	dwh_northwind	dim_employee	company	STRING			BQ	stg_northwind	stg_employees	company	STRING		
BQ	dwh_northwind	dim_employee	last_name	STRING			BQ	stg_northwind	stg_employees	last_name	STRING		
BQ	dwh_northwind	dim_employee	first_name	STRING			BQ	stg_northwind	stg_employees	first_name	STRING		
BQ	dwh_northwind	dim_employee	email_address	STRING			BQ	stg_northwind	stg_employees	email_address	STRING		
BQ	dwh_northwind	dim_employee	job_title	STRING			BQ	stg_northwind	stg_employees	job_title	STRING		
BQ	dwh_northwind	dim_employee	business_phone	STRING			BQ	stg_northwind	stg_employees	business_phone	STRING		
BQ	dwh_northwind	dim_employee	home_phone	STRING			BQ	stg_northwind	stg_employees	home_phone	STRING		
BQ	dwh_northwind	dim_employee	mobile_phone	STRING			BQ	stg_northwind	stg_employees	mobile_phone	STRING		
BQ	dwh_northwind	dim_employee	fax_number	STRING			BQ	stg_northwind	stg_employees	fax_number	STRING		
BQ	dwh_northwind	dim_employee	address	STRING			BQ	stg_northwind	stg_employees	address	STRING		
BQ	dwh_northwind	dim_employee	city	STRING			BQ	stg_northwind	stg_employees	city	STRING		
BQ	dwh_northwind	dim_employee	state_province	STRING			BQ	stg_northwind	stg_employees	state_province	STRING		
BQ	dwh_northwind	dim_employee	zip_postal_code	STRING			BQ	stg_northwind	stg_employees	zip_postal_code	STRING		
BQ	dwh_northwind	dim_employee	country_region	STRING			BQ	stg_northwind	stg_employees	country_region	STRING		
BQ	dwh_northwind	dim_employee	web_page	STRING			BQ	stg_northwind	stg_employees	web_page	STRING		
BQ	dwh_northwind	dim_employee	notes	STRING			BQ	stg_northwind	stg_employees	notes	STRING		
BQ	dwh_northwind	dim_employee	attachments	STRING			BQ	stg_northwind	stg_employees	attachments	STRING		
BQ	dwh_northwind	dim_employee	insertion_timestamp	DATETIME		Updated by transformation tool.							

For the dim_date table, because there is no source we have just left a comment in the source section to the side saying 'Created from 'GENERATE_DATE_ARRAY.'

When mapping our fact_sales table, we're going to be using dimensions from the order_details and orders table. We can see that here as well.

BQ	dwh_northwind	fact_sales	order_id	INTEGER			BQ	stg_northwind	stg_order_details	order_id	INTEGER
BQ	dwh_northwind	fact_sales	product_id	INTEGER			BQ	stg_northwind	stg_order_details	product_id	INTEGER
BQ	dwh_northwind	fact_sales	customer_id	INTEGER			BQ	stg_northwind	stg_orders	customer_id	INTEGER
BQ	dwh_northwind	fact_sales	employee_id	INTEGER			BQ	stg_northwind	stg_orders	employee_id	INTEGER
BQ	dwh_northwind	fact_sales	shipper_id	INTEGER			BQ	stg_northwind	stg_orders	shipper_id	INTEGER
BQ	dwh_northwind	fact_sales	quantity	FLOAT64			BQ	stg_northwind	stg_order_details	quantity	FLOAT64
BQ	dwh_northwind	fact_sales	unit_price	FLOAT64			BQ	stg_northwind	stg_order_details	unit_price	FLOAT64
BQ	dwh_northwind	fact_sales	discount	FLOAT64			BQ	stg_northwind	stg_order_details	discount	FLOAT64
BQ	dwh_northwind	fact_sales	status_id	INTEGER			BQ	stg_northwind	stg_order_details	status_id	INTEGER
BQ	dwh_northwind	fact_sales	date_allocated	DATETIME			BQ	stg_northwind	stg_order_details	date_allocated	DATETIME
BQ	dwh_northwind	fact_sales	purchase_order_id	INTEGER			BQ	stg_northwind	stg_order_details	purchase_order_id	INTEGER
BQ	dwh_northwind	fact_sales	inventory_id	INTEGER			BQ	stg_northwind	stg_order_details	inventory_id	INTEGER
BQ	dwh_northwind	fact_sales	order_date	DATETIME			BQ	stg_northwind	stg_orders	order_date	DATETIME
BQ	dwh_northwind	fact_sales	shipped_date	DATETIME			BQ	stg_northwind	stg_orders	shipped_date	DATETIME
BQ	dwh_northwind	fact_sales	paid_date	DATETIME			BQ	stg_northwind	stg_orders	paid_date	DATETIME
BQ	dwh_northwind	fact_sales	insertion_timestamp	DATETIME							

Again, remember this is an iterative process. You will always find little errors and have to go back to fix them and that's okay, that's the point.

If we look at our Bus Matrix again we know we originally created it from our high-level entities, defined the grain and the granularity, and later on created a detailed document on facts and dimensions. Sometime at this stage, it is useful to make note of the types of dimensions. For example, dim_date is a role-playing dimension.

Now we're going into our last mapping stage which is to build a source to target mapping document for our analytical layer.

Source to Target Mapping (Data Warehouse to One Big Table)

At this point we know exactly what fact and dim tables are required to build our sales overview report. All we need to do is to bring our dim and fact table that we need to build our OBT for sales overview report, and find the sources that they belong to. So we repeat the same process, we go to our source to target mapping document for data warehouse to OBT and fill out the table.

One Big Table is just what it sounds like, it's a big denormalized table containing all the columns that are required to build reports. By denormalizing all the tables into one big table, it speeds up our reporting purposes and makes access of the data easier for the business.

So say we start by bringing in our 'dim_customer' table from the Data Warehouse and paste that into the source, but then in the target we change the schema name from dim_customer to obt_sales_overview because our one big table needs to be in a different schema or dataset. We repeat the same process for employees, paste the original on the right in source, then on the left do the same but change the schema name to obt_sales_overview. We do this for every table we have, and then last we bring in our dim_date table, with the comment on the right in source saying 'Created from 'GENERATE_DATE_ARRAY.'

	Target (Analytical Layer - OBT)						Source (Dimensional Data Warehouse)						
Target System	Target Schema	Target Table	Target Column Name	Target Data Type	Description	Comment	Source System	Source Schema	Souce Table	Source Column Name	Source Data Type	Description	Comment
1	obt_sales_overview	unique_customer_id	INTEGER	NO			BQ	dim_customer	unique_customer_id	INTEGER	NO	PK	
2	obt_sales_overview	company	STRING	NULLABLE			BQ	dim_customer	company	STRING	NULLABLE		
3	obt_sales_overview	last_name	STRING	NULLABLE			BQ	dim_customer	last_name	STRING	NULLABLE		
4	obt_sales_overview	first_name	STRING	NULLABLE			BQ	dim_customer	first_name	STRING	NULLABLE		
5	obt_sales_overview	email_address	STRING	NULLABLE			BQ	dim_customer	email_address	STRING	NULLABLE		
6	obt_sales_overview	job_title	STRING	NULLABLE			BQ	dim_customer	job_title	STRING	NULLABLE		
7	obt_sales_overview	business_phone	STRING	NULLABLE			BQ	dim_customer	business_phone	STRING	NULLABLE		
8	obt_sales_overview	home_phone	STRING	NULLABLE			BQ	dim_customer	home_phone	STRING	NULLABLE		
9	obt_sales_overview	mobile_phone	STRING	NULLABLE			BQ	dim_customer	mobile_phone	STRING	NULLABLE		
10	obt_sales_overview	fax_number	STRING	NULLABLE			BQ	dim_customer	fax_number	STRING	NULLABLE		
11	obt_sales_overview	address	STRING	NULLABLE			BQ	dim_customer	address	STRING	NULLABLE		
12	obt_sales_overview	city	STRING	NULLABLE			BQ	dim_customer	city	STRING	NULLABLE		
13	obt_sales_overview	state_province	STRING	NULLABLE			BQ	dim_customer	state_province	STRING	NULLABLE		
14	obt_sales_overview	zip_postal_code	STRING	NULLABLE			BQ	dim_customer	zip_postal_code	STRING	NULLABLE		
15	obt_sales_overview	country_region	STRING	NULLABLE			BQ	dim_customer	country_region	STRING	NULLABLE		
16	obt_sales_overview	web_page	STRING	NULLABLE			BQ	dim_customer	web_page	STRING	NULLABLE		
17	obt_sales_overview	notes	STRING	NULLABLE			BQ	dim_customer	notes	STRING	NULLABLE		
18	obt_sales_overview	attachments	STRING	NULLABLE			BQ	dim_customer	attachments	STRING	NULLABLE		
19	obt_sales_overview	unique_employee_id	INTEGER	NO			BQ	dim_employee	unique_employee_id	INTEGER	NO	PK	
20	obt_sales_overview	company	STRING	NULLABLE			BQ	dim_employee	company	STRING	NULLABLE		
21	obt_sales_overview	last_name	STRING	NULLABLE			BQ	dim_employee	last_name	STRING	NULLABLE		
22	obt_sales_overview	first_name	STRING	NULLABLE			BQ	dim_employee	first_name	STRING	NULLABLE		
23	obt_sales_overview	email_address	STRING	NULLABLE			BQ	dim_employee	email_address	STRING	NULLABLE		
24	obt_sales_overview	job_title	STRING	NULLABLE			BQ	dim_employee	job_title	STRING	NULLABLE		
25	obt_sales_overview	business_phone	STRING	NULLABLE			BQ	dim_employee	business_phone	STRING	NULLABLE		
26	obt_sales_overview	home_phone	STRING	NULLABLE			BQ	dim_employee	home_phone	STRING	NULLABLE		
27	obt_sales_overview	mobile_phone	STRING	NULLABLE			BQ	dim_employee	mobile_phone	STRING	NULLABLE		
28	obt_sales_overview	fax_number	STRING	NULLABLE			BQ	dim_employee	fax_number	STRING	NULLABLE		
29	obt_sales_overview	address	STRING	NULLABLE			BQ	dim_employee	address	STRING	NULLABLE		
30	obt_sales_overview	city	STRING	NULLABLE			BQ	dim_employee	city	STRING	NULLABLE		
31	obt_sales_overview	state_province	STRING	NULLABLE			BQ	dim_employee	state_province	STRING	NULLABLE		
32	obt_sales_overview	zip_postal_code	STRING	NULLABLE			BQ	dim_employee	zip_postal_code	STRING	NULLABLE		
33	obt_sales_overview	country_region	STRING	NULLABLE			BQ	dim_employee	country_region	STRING	NULLABLE		
34	obt_sales_overview	web_page	STRING	NULLABLE			BQ	dim_employee	web_page	STRING	NULLABLE		
35	obt_sales_overview	notes	STRING	NULLABLE			BQ	dim_employee	notes	STRING	NULLABLE		
36	obt_sales_overview	attachments	STRING	NULLABLE			BQ	dim_employee	attachments	STRING	NULLABLE		
37	obt_sales_overview	unique_product_id	INTEGER	NO			BQ	dim_products	unique_product_id	INTEGER	NO		
38	obt_sales_overview	product_code	STRING	NO			BQ	dim_products	product_code	STRING	NO		
39	obt_sales_overview	product_name	STRING	NO			BQ	dim_products	product_name	STRING	NO		
40	obt_sales_overview	description	STRING	NULLABLE			BQ	dim_products	description	STRING	NULLABLE		
41	obt_sales_overview	supplier_company	STRING	NO			BQ	dim_products	supplier_company	STRING	NO		
42	obt_sales_overview	standard_cost	FLOAT64	NULLABLE			BQ	dim_products	standard_cost	FLOAT64	NULLABLE		
43	obt_sales_overview	list_price	FLOAT64	NULLABLE			BQ	dim_products	list_price	FLOAT64	NULLABLE		
44	obt_sales_overview	reorder_level	INTEGER	NULLABLE			BQ	dim_products	reorder_level	INTEGER	NULLABLE		
45	obt_sales_overview	target_level	INTEGER	NULLABLE			BQ	dim_products	target_level	INTEGER	NULLABLE		
46	obt_sales_overview	quantity_per_unit	STRING	NULLABLE			BQ	dim_products	quantity_per_unit	STRING	NULLABLE		
47	obt_sales_overview	discontinued	INTEGER	NULLABLE			BQ	dim_products	discontinued	INTEGER	NULLABLE		
48	obt_sales_overview	minimum_reorder_quantity	INTEGER	NULLABLE			BQ	dim_products	minimum_reorder_quantity	INTEGER	NULLABLE		
49	obt_sales_overview	category	STRING	NULLABLE			BQ	dim_products	category	STRING	NULLABLE		
50	obt_sales_overview	attachments	STRING	NULLABLE			BQ	dim_products	attachments	STRING	NULLABLE		

By joining all of these together we would be creating our one big table for the sales_overview_report. If you go back to the Bus Matrix, you can see we still have a couple of more reports left that were outlined based on the business processes, that we still need to build. You would do this for all of those as well.

Next, we have to create a logical model by going back to Draw.io.

Logical Model

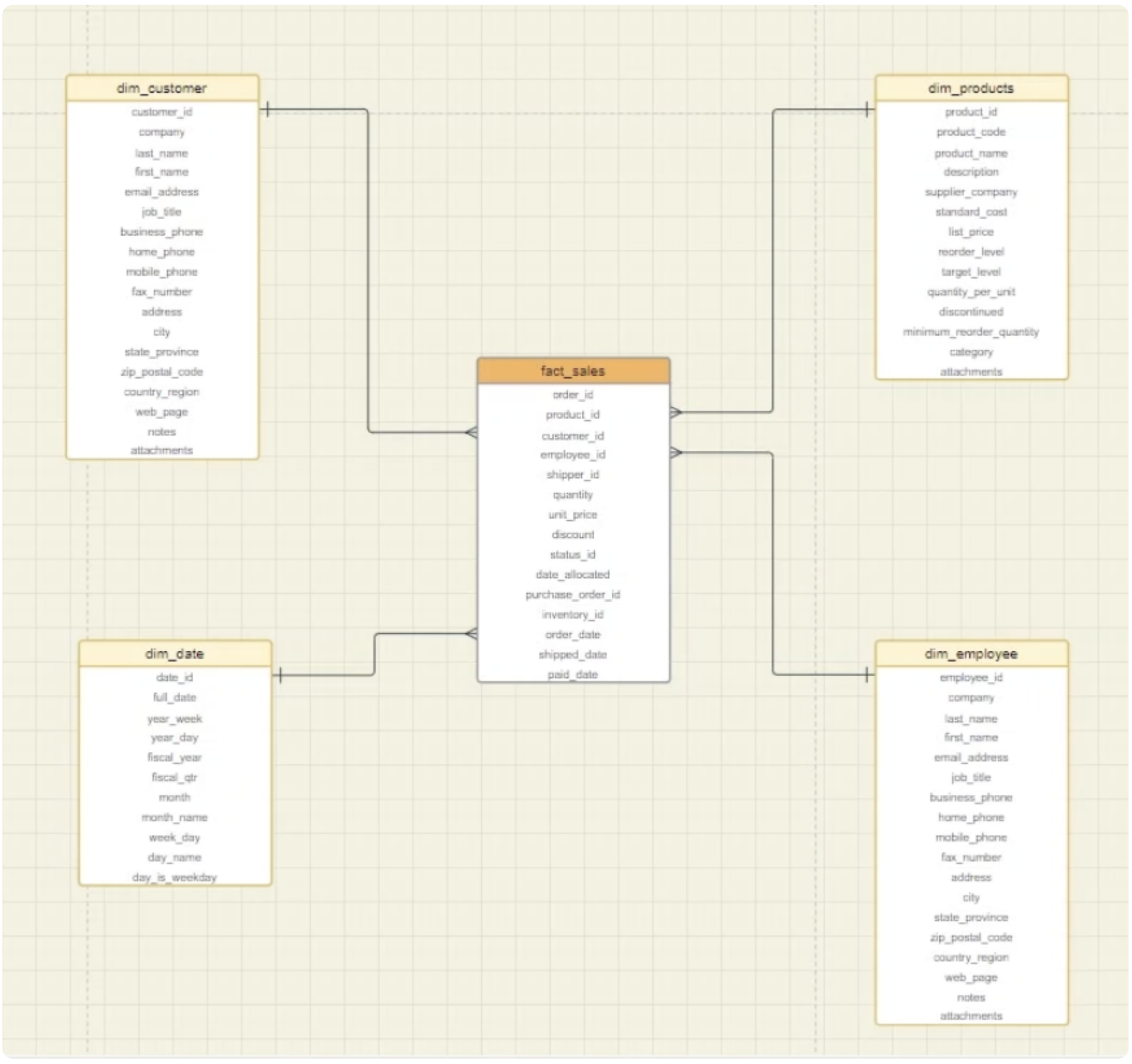
For us to build our logical model we need to reference our Bus Matrix once again. We'll start with our first fact table which is fact_sales and all the dimensions that are associated with it.

How do we know what fields go into the table? `We reference our attributes and measures document`.

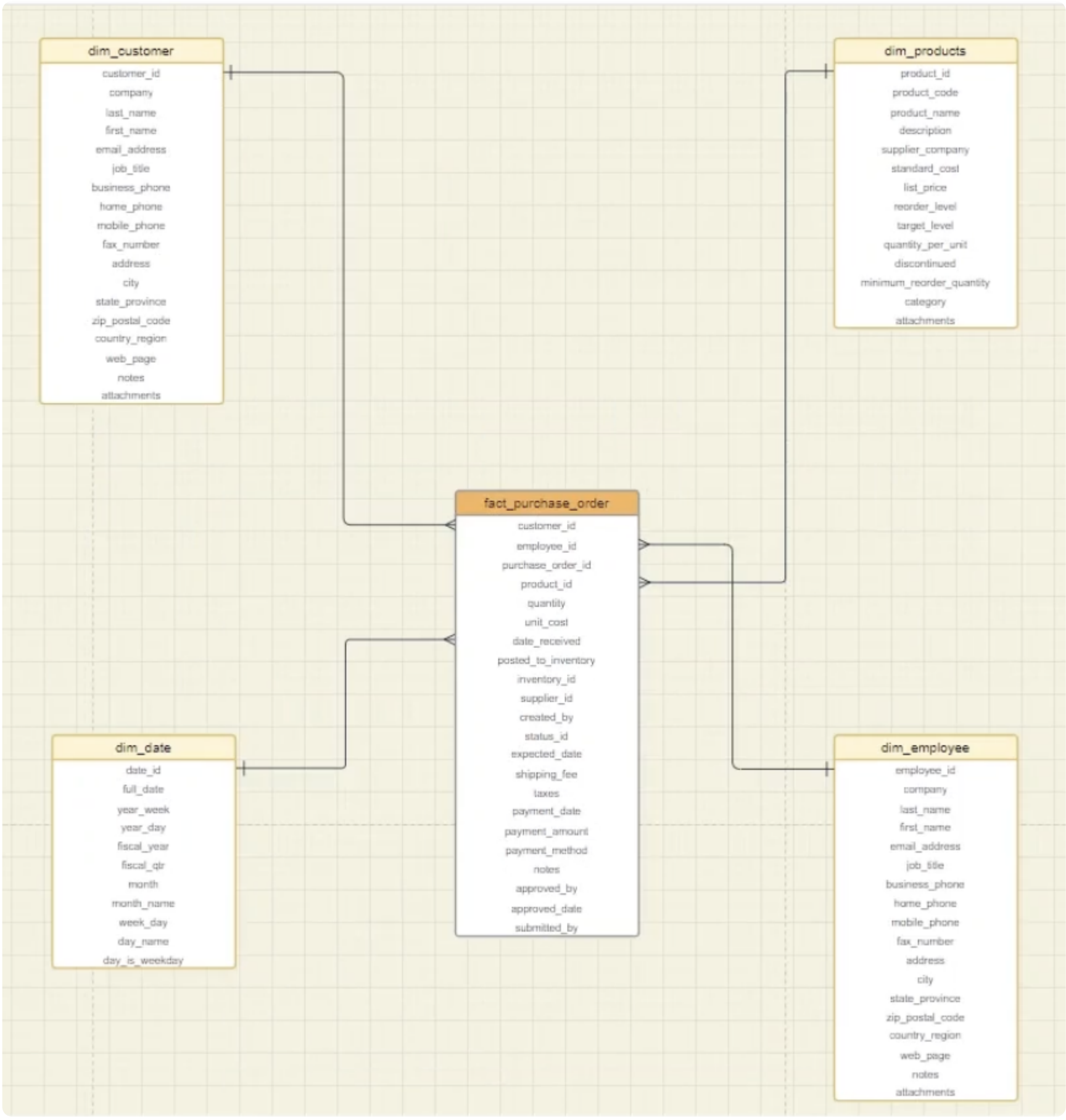
So it would look something like this…

Then we need to create our dim tables. We know we need four dim tables, dim_date, dim_employee, dim_customer, and dim_products.

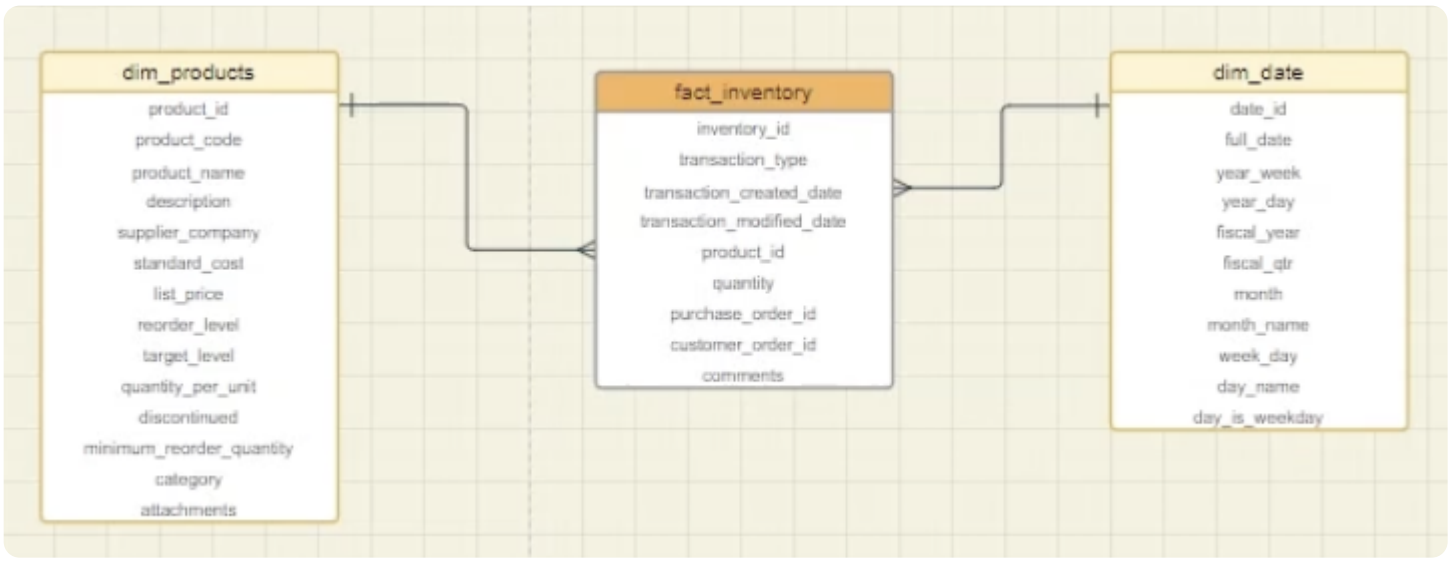
Once those four dim tables are created and the relationships between them are outlined (by indicating one-to-one or one-to-many), it will look something like this…



Next we will do this for our other fact tables, like fact_purchase_order…



Or fact_inventory…



If you're wondering, okay, I went to the Bus Matrix, I see the Xs say I need product, suppliers, and date, but my diagram only has product and date, why is that? Because if you remember, we have combined supplier information inside our dim_product table.

Physical Model

Now that we have the logical model, we can build our physical model. Then we'll use the source to target document we created previously as our reference point to setup our data warehouse (dbt).

During this dbt phase we will…

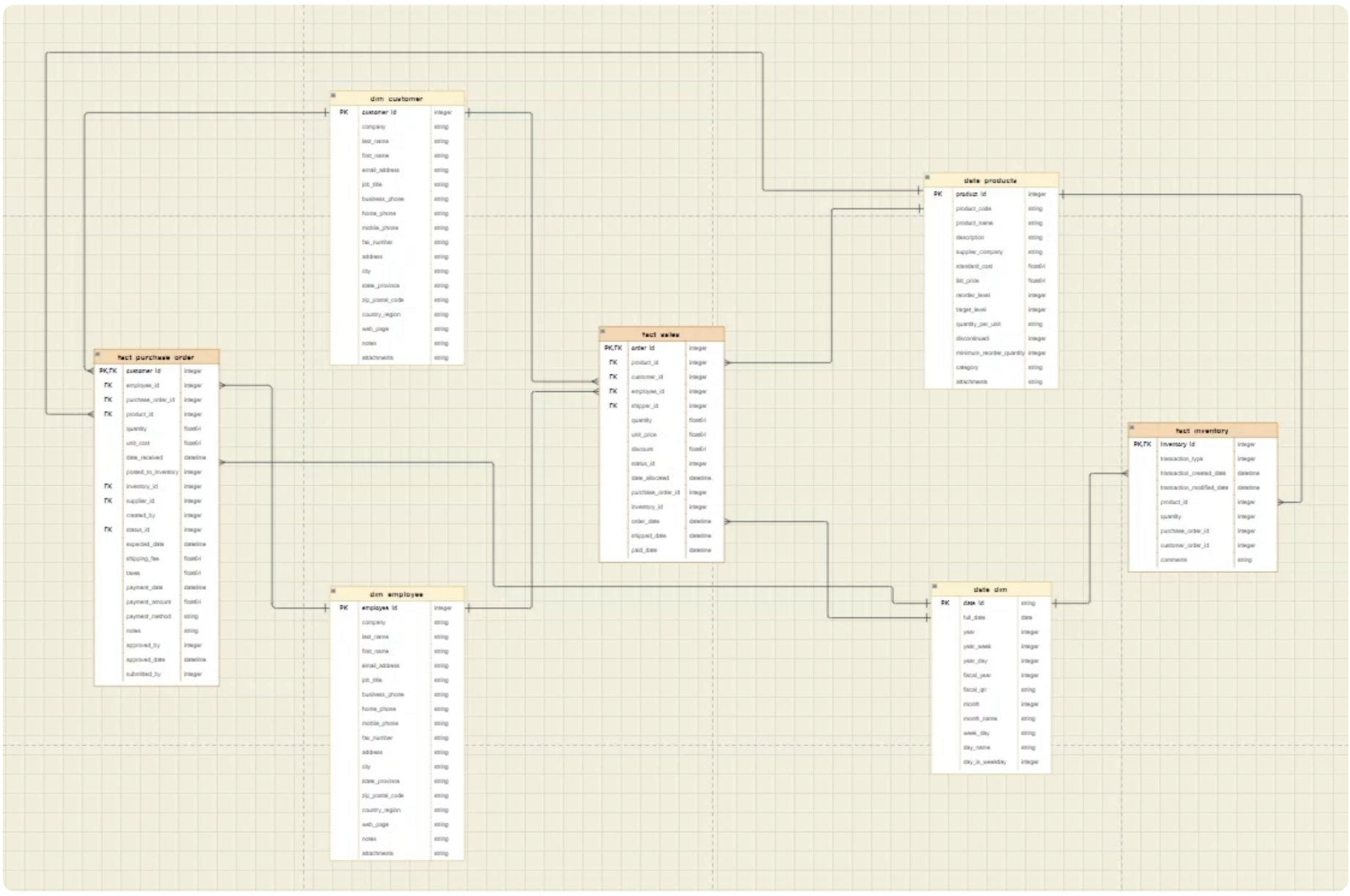
1. Create Data Warehouse Layers
2. Create Data Warehouse Tables
3. Populate Tables using SQL

So first let’s go back to Draw.io to create the physical model. If we take a look at our logical model we just created, the first thing we notice is how the tables are replicated accross our fact tables. For example, the dim_date table appears in fact_inventory, fact_sales, and fact_purchase_orders. The same thing goes for dim_product, dim_customer, dim_employee. All of these dimensions are shared across multiple facts so what we can do in this case is create a conformed dimension in our physical design. We’ll also be adding data types, and primary keys and foreign keys information to our design model.

So to start, we move the information from our logical model into the physical model, mainly the column names. Then we will add one column on either side of that. On the left, we will specify what is the primary key, what are the foreign keys, and on the right we will specify data types (which we get from our source to target mapping document). So our original fact_sales table will end up looking something like…

We would do the same for our dim tables, by adding date types, and specifying primary and foreign keys.

After creating the other fact and dimension tables, we end up with something like this…



Here we only have conformed dimensions, just four dim tables for the three fact tables.

Now we'll be using dbt to do the physical implementation of the model.

dbt Overview

Let's remind ourselves what we are actually building.

1. Using dbt we will be getting the data from our data lake to the staging layer.
2. We'll do the data transformation in our staging layer.
3. Then we'll be using our physical design model to build our dimensional data warehouse layer.
4. After this we'll be building our reporting layer.

Throughout all of these steps we'll be using and referencing our source to target mapping document.

If you're using dbt for the first time it can seem a little confusing, so let's go through it one by one.

What is a dbt project?

A dbt project is a collection of SQL files and YML files which dbt uses to do the data transformation. A dbt project contains a `dbt_project.yml` file and there is a `profiles.yml` file which you will find under .dbt/profiles.yml.

So project.yml contains all the information about a dbt project, and the profiles.yml file contains all the information about your database and the target. The target in this case basically defines all of your database settings.

Imagine you have a developer environment, a test environment, a production environment— profiles.yml gives you a very easy way to switch between the various environments.

Macros

The macros folder are basically a piece of code that you can reuse many times. This basically allows you to keep your SQL code dry. Dry basically means don't repeat yourself.

To understand how macros work, we need to get familiar with Jinja. Jinja is basically a templating language. Using Jinja, you can get rid of your static code and you can make your code more dynamic. Jinja variables are usually replaced by the values which are parsed in when the dbt compiles a SQL code.

Here's an example of a dbt model that leverages Jinja:

/models/order_payment_method_amounts.sql

```
{% set payment_methods = ["bank_transfer", "credit_card", "gift_card"] %}

select
    order_id,
    {% for payment_method in payment_methods %}
    sum(case when payment_method = '{{payment_method}}' then amount end) as {{payment_method}}_amount,
    {% endfor %}
    sum(amount) as total_amount
from app_data.payments
group by 1
```

This query will get compiled to:

/models/order_payment_method_amounts.sql

```
select
    order_id,
    sum(case when payment_method = 'bank_transfer' then amount end) as bank_transfer_amount,
    sum(case when payment_method = 'credit_card' then amount end) as credit_card_amount,
    sum(case when payment_method = 'gift_card' then amount end) as gift_card_amount,
    sum(amount) as total_amount
from app_data.payments
group by 1
```

So for example, we can have a variable…
(% set payment_methods = [“bank_transfer”, “credit_card”, “gift_card”] %)

And then a SELECT statement which has a loop. Usually without dbt we cannot write a SQL code like that, but we can do that now because of Jinja templating language. So what this code will do is it will loop through the payment_methods in the variable when running the query, and will produce one neat output. Normally without Jinja code you’d have to repeat it three different times for each payment_method but using Jinja you only have to write once.

So now macros are very similar in that you don’t have to repeat code. You create a macro once and you can use that macro across dbt projects. Let’s take a look at this macro…

Macro files can contain one or more macros — here's an example:

macros/cents_to_dollars.sql

```
{% macro cents_to_dollars(column_name, scale=2) %}
    ({{ column_name }} / 100)::numeric(16, {{ scale }})
{% endmacro %}
```

A model which uses this macro might look like:

models/stg_payments.sql

```
select
  id as payment_id,
  {{ cents_to_dollars('amount') }} as amount_usd,
  ...
from app_data.payments
```

This would be *compiled* to:

target/compiled/models/stg_payments.sql

```
select
  id as payment_id,
  (amount / 100)::numeric(16, 2) as amount_usd,
  ...
from app_data.payments
```

This macro does cents_to_dollars. It receives two parameter values, column_name and precision. And once this macro receives a column name, it does a calculation. And how do we use this macro in our SQL code? Well, you would write your SQL as normal, but instead of hard-coding your value, you would use a macro and parse the value.

Models

These model folders contain SQL files, and these SQL files are where your transformation happens. We can order and create our models in a very granular way here.

Snapshots

These provide a type 2 slowly changing dimensions, so if you have a requirement to create your table as a type 2 slowly changing dimension, you will be using this snapshot folder to create all of those configurations and SQL files here.

Test

These are basically types of assertions that allow you to test your assumptions about your data or SQL file. dbt provides two ways of testing, singular and generic. Singular testing

is usually done by writing a SQL query which will return failing records. Generic tests are something we can keep on reusing over and over again, this could be something like not-null test or uniqueness test.

Physical Implementation (Staging Layer)

In dbt we should try our best to use Jinja templating and make sure our model is as clean as possible, so writing regular SQL where you do SELECT and twenty things after it, is discouraged.

So instead of writing…

```
SELECT
id,
company,
last_name,
first_name,
email_address,
job_title,
business_phone,
home_phone,
mobile_phone,
fax_number,
address,
city,
state_province,
zip_postal_code,
country_region,
web_page,
notes,
attachments,
insertion_timestamp
FROM 'ae_bootcamp.dl_northwind_customers'
```

A more dbt friendly way would be…

```
with source as (
select from {{ source('northwind', 'customers') }}
)
select from source
```

So we started our SQL code with a CTE, named our CTE source, within the CTE we have a SELECT statement which is selecting from the customer table, and the source is called Northwind. Then finally we are selecting everything from our CTE source. So now the question is, `where is this ‘northwind’ coming from?` How do we know what is inside this Northwind?

Northwind here is defined under something called source.yml file which we need to create. Inside this document we define our sources which looks something like this…

```
version: 2
sources:
```

- name: northwind
  schema: dl_northwind
  tables:
    - name: customer
    - name: employee_privileges
    - name: employees
    - name: inventory_transaction_types
    - name: inventory_transactions

- name: invoices
- name: order_details
- name: order_details_status
- name: order_status
- name: orders
- name: orders_tax_status
- name: privileges
- name: products
- name: purchase_order_details
- name: purchase_order_status
- name: purchase_orders
- name: shippers
- name: suppliers
- name: strings

Here we have the name of the source which is northwind, the schema which is referencing the data lake tables, and all the tables that exist under our dl_northwind. By defining this all as part of our source.yml file, we can now reference inside our SQL file.

After typing ‘dbt run’ in the terminal, we have successfully created our staging customer table! We can check this by going into BigQuery and refreshing to see if it comes up. We now need to do the same for the rest of the tables, so employees, employee_priviliges, etc.

Sometimes a staging layer is created as a view layer, so since we are using dbt it’s as simple as just changing the ‘materialized’ configuration in the dbt_project.yml file from ‘table’ to ‘view.’ Then if we run ‘dbt run –full-refresh’ it will create all of our existing tables in our staging layer, as views. You can see how powerful dbt is, with a single configuration change you can easily change from a table to a view. We’ll switch it back to tables since we’d like to retain it as a table for now. You can check this worked by refreshing the database in BigQuery.

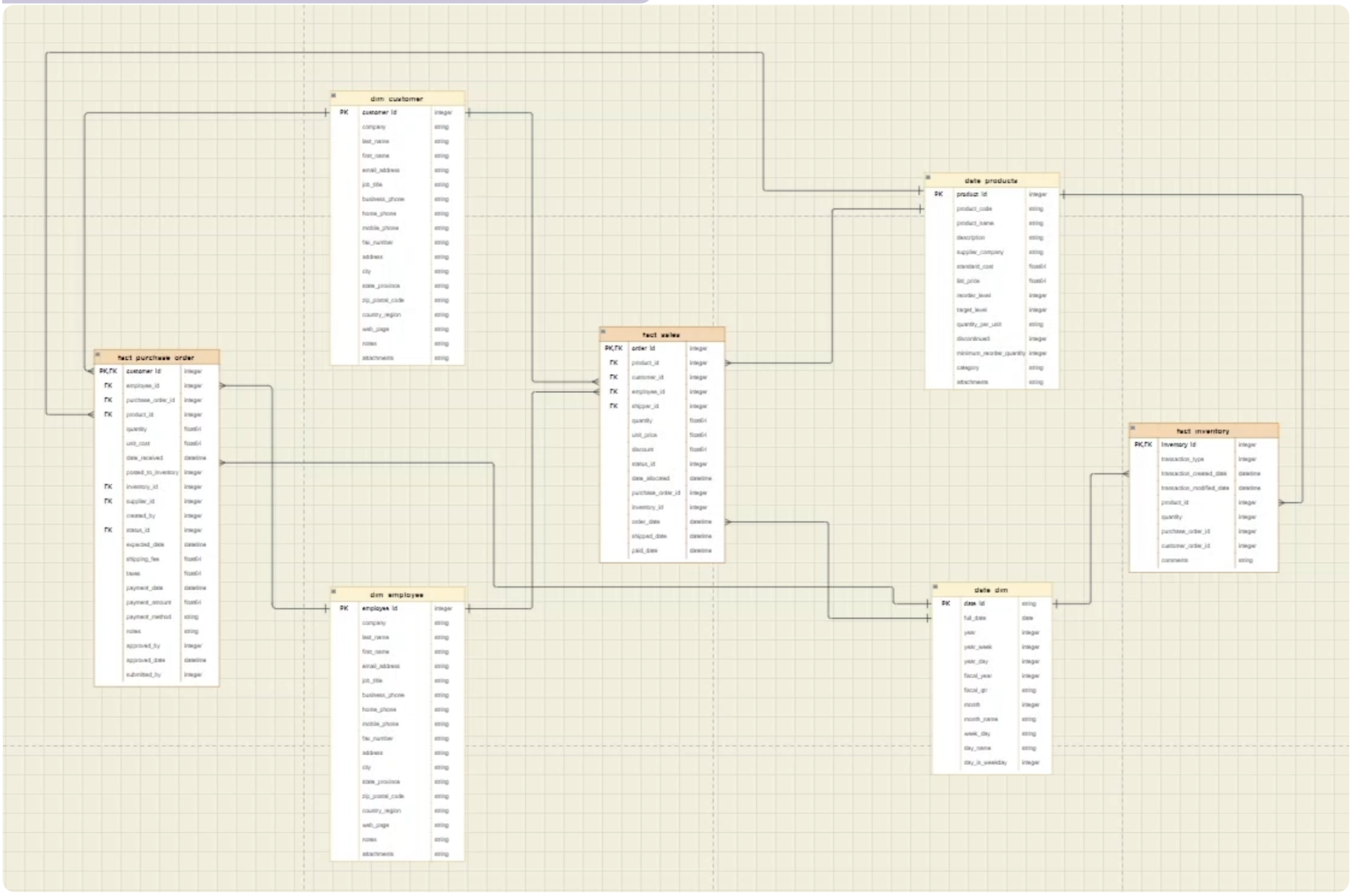
Now, we have successfully created the staging layer! Now we just have to save everything and push to Git so we don’t lose anything. It’s important to note here, we don’t usually push to the main branch on Git. That’s usually saved for production purposes. The only reason we are pushing our changes directly to our main branch for now is for tutorial purpose, and to keep things a bit more simple. Usually what we would do is create different feature branches, so for example, we would create a new branch and call it staging-layer and make all of our code changes in this branch and then push this branch to our git repository. For now we’ll skip that to keep things very clear.

If we go back to our staging layer source to target mapping document however, we notice we have the insertion_timestamp row missing, so let’s see how we can improve our existing staging layer.

Physical Implementation Dim Tables (Data Warehouse Layer)

Now that we’ve built our staging layer, we’re going to be building our warehouse layer. Let’s first go back to our Draw.io and take a look at our physical ERD diagram to refresh

our memory of what exactly we need to build.



We can see here we need to build three fact tables and four dimension tables. This is our conformed data warehouse.

dim_customer

So let’s start by building one of our dim tables first, say dim_customer. If we zoom in, we can see we have all the data types and everything that’s associated and required to build our dim_customer table.

	dim_customer	
PK	customer_id	integer
	company	string
	last_name	string
	first_name	string
	email_address	string
	job_title	string
	business_phone	string
	home_phone	string
	mobile_phone	string
	fax_number	string
	address	string
	city	string
	state_province	string
	zip_postal_code	string
	country_region	string
	web_page	string
	notes	string
	attachments	string

So once we’ve created our dim_customer.sql table under warehouse in dbt, we can go back to our source to target mapping document to just copy all our column names—this will make writing our SQL query a lot easier. That list will become our base for our SELECT statement.

Now let’s try to understand how we’re going to select it from the staging layer by going back to our source to target mapping document again. So our dim_customer table is coming directly from our stg_customer layer so that’s easier for us since we don’t have to do too many things here, just get everything from the staging layer and make a few changes to it.

Previously when building our staging layer we were selecting from ‘source’ and this source information is identified inside our source.yml file. Now how do we do this for our dim_customer? We can’t just do ‘FROM {{ source('northwind’, ‘stg_customer’) }} because if you remember, we selected that source for our staging layer because the data lake layer is built outside of our dtb framework. Once we start building tables within our dbt framework, we should not be using source, we should be using ref as our reference to our tables within dbt. So instead of using source, we should write: FROM {{ ref('stg_customer') }} because behind the scenes, this ‘ref’ is already a macro and dbt already has an understanding of where to get the staging customer table.

But once we have created this it’s not complete yet, we still need to set up our dbt project. Initially, we have only set up our dtb to create data within the staging layer. Now we need to create for our warehouse layer. So we need to go back to dbt_project.yml and paste…

```
staging:
+enabled: true
+materialized: table
+schema: stg_northwind
warehouse:
+enabled: true
+materialized: table
+schema: dwh_northwind
```

What we did is copied the staging template, and changed it to be suitable for the warehouse layer.

Now we don’t want to run dbt run here because we don’t want to run everything, we’re happy with the staging layer and don’t want to make any more changes to it. We only want to run our warehouse layer so the command instead is: dbt run -m warehouse.dim_customer.

At this point we might run into an error. dbt might tell us customer_id doesn’t exist and if we go back to our source to target mapping document we can see that in the staging layer that row was called id, and in warehouse we named it customer_id. So we need to update that to reflect those changes. We go back to our dbt project and instead of saying customer_id, we alias it and write id AS customer_id. Then we do dbt run -m warehouse.dim_customer.

If you remember, we originally had a duplicated entry in the dim_customer table. We need to find a way to remove duplicates in our dimensional model, our dim_customer table. So we can do this by slightly changing the query like this…

```
WITH source AS (
SELECT
id AS customer_id,
company,
last_name,
first_name,
email_address,
job_title,
business_phone,
home_phone,
mobile_phone,
fax_number,
address,
city,
state_province,
zip_postal_code,
country_region,
web_page,
notes,
```

```
attachments,
current_timestamp() AS insertion_timestamp,
FROM {{ ref('stg_customer') }}
),
unique_source AS (
SELECT ,
row_number() over(partition by customer_id) AS row_number
FROM source
)
SELECT
FROM unique_source
WHERE row_number = 1
```

Now this looks complicated but it’s not really. All we’re doing is selecting everything from our staging customer table using the ref macro. Then once we have all that selected, we’re applying an analytic function. Basically the row_number goes over the analytic functions to calculate the row number, so for each row it is partitioning that by customer_id. Then we are basically selecting all the fields from our unique_source CTE. So if we run this query and then look at BigQuery again, we’ll still see duplicate records, but we will also see a new column called row number, and a little 2 next to the duplicate entry. This means the row number function went over each row and identified that it’s already seen one of these rows before, and therefore it’s a duplicate. If there’s a 1 it means it’s unique, if there’s a 2 it means it’s not unique. So in our SQL query, by filtering out number 1, we’re only getting the unique records. We also don’t want the row_number in our table so we’re going to add a line to filter it out, so the last section of our SQL query will instead be…

```
SELECT *
EXCEPT
(row_number),
FROM unique_source
WHERE row_number = 1
```

Now if we do dbt run -m warehouse.dim_customer again and then check BigQuery, we’ll see we only have 29 records instead of 30, so the duplicate record has been removed.

dim_employee

At this point we still need to create our other dim tables so let’s create dim_employee because we know it’s very similar to dim_customer. So we can just duplicate our completed dim_customer.sql file and change the customer fields to employee. The reason we’re still including our de-duplication query is because we want to make sure that we are not bringing in non-unique data. This is important since sometimes you do actually want to bring in duplicated data that was created over periods of time, like slowly changing dimensions. In that case it would be important since you would need to be able to identify the change in your source data.

dim_date

Now let’s move on to dim_date. To create dim_date we need to use this query…

```
SELECT
FORMAT_DATE ('%F', d) AS id,
d AS full_date,
EXTRACT(YEAR FROM d) AS year,
EXTRACT(WEEK FROM d) AS year_week,
EXTRACT(DAY FROM d) AS year_day,
EXTRACT(YEAR FROM d) AS fiscal_year,
FORMAT_DATE ('%Q', d) AS fiscal_qtr,
EXTRACT(MONTH FROM d) AS month,
FORMAT_DATE (%B, d) AS month_name,
```

```
FORMAT_DATE ('%W', d) AS week_day,
FORMAT_DATE ('%A', d) AS day_name,
(CASE WHEN FORMAT_DATE('%A', d) IN ('Sunday', 'Saturday') THEN 0 ELSE 1 END) AS
day_is_weekday,
FROM (
SELECT
*
FROM
UNNEST(GENERATE_DATE_ARRAY('2014-01-01', '2050-01-01', INTERVAL 1 DAY)) AS d)
```

Again, this query was mentioned earlier when we were building our source to target mapping document. Basically this query is being generated from BigQuery generate data array function. We give this function two parameters, a start and end date, and based on the value this function returns we are changing the format of that date to meet our requirement, and this becomes our dim_date table. So we dbt run -m warehouse.dim_date and check BigQuery to make sure it worked.

At this stage, know that usually dim_date is created as a view table. Especially in BigQuery this dim_table is very fast due to the nature of the generate_array function so how do we change our table from a materialized table to view table? We can do that in our dbt_project.yml. We want to make sure only the dim_date table has the materialized view configuration, so we can do that by just adding…

```
dim_date:
+materialized: view
```

Under the +schema in the warehouse section.

dim_product

Now all we need to do is create our dim_product table—our final dimension table. After copying the column names from our source to target document and formatting our query, we might notice that in the source to target document, supplier_company actually comes from stg_suppliers not stg_products. To fix this we can just join products onto suppliers, like this…

```
FROM {{ ref('stg_products') }} p
LEFT JOIN {{ ref('stg_suppliers') }} s
ON s.id = p.supplier_id
```

However when we run this we receive an error, that there is no matching signature for operator. That is because in stg_suppliers, the id column is an integer, but in products, the supplier_ids is a string. So let's try and understand why that field is a string instead of an integer. If we preview the data we can straight away notice that there seems to be some error in our supplier_ids field which is causing the data type to be incorrect. Some ids are coming up as two numbers with a semicolon between them, ex. 2;6. As a next step, it is difficult for us to make a decision as developers as to where these errors are coming from so we have gone back to the business, and double checked with them how we need to handle this. Do we split this into two different columns or should we just remove the semi-colon and everything beyond that? The business has confirmed that we should just get rid of those rows because they are an error. Before we do that, we need to find out how we can ignore these errors.

So first we stay in BigQuery and run some queries in the stg_products table. We start by filtering everything with a semicolon in the supplier_ids column…

```
SELECT supplier_ids FROM bootcamp-ae.dbt_dariya_stg_northwind.stg_products
WHERE supplier_ids LIKE "%;%"
```

Running this query shows us where exactly the issue is. And if we reverse this query, and write NOT LIKE “%;%” we should only get the rows with correctly inputted ids. Now importantly, we should not be doing this type of work in our dim_products table because

the dim_product focus is to build a dimensional model. This type of data cleaning work we should be doing in the staging layer. So we need to go back to our stg_products file in dbt and make some changes there.

So now in that stg_products file we have updated our SQL to…

```
with source as (
select cast(supplier_ids as integer) as supplier_id,
id,
product_code,
description,
standard_cost,
list_price,
reorder_level,
target_level,
quantity_per_unit,
discontinued,
minimum_reorder_quantity,
category,
attachments,
from {{ source('northwind', 'products') }}
where supplier_ids not like "%;%"
)
select
*,
current_timestamp() AS ingestion_timestamp
from source
```

Instead of doing select star, we have brought in all the columns because we need to make a few changes to our supplier id. First, we cast it as an integer instead of string, and we have removed the s from the end and called it supplier_id instead of supplier_ids which is more standardized. We have also added a where clause to filter out the incorrect records. This should basically clean up our products table and re-create it in the way we want it.

Now we run dbt run -m staging.stg_products --full-refresh. The reason we run a full refresh here is because now we want to recreate this table from scratch.

If we check BigQuery, we can see the supplier_id is an integer now, and all the ids look normalized when we preview our table.

Now we just need to update our field name from supplier_ids to supplier_id in our dim_products file, and then we can run the dbt command to create that table.

Now we have created all of our dimension tables! Let’s move on to creating our fact tables. So we can see we have fact_purchase_order, fact_sales, and fact_inventory. Let’s start with fact_sales.

fact_sales

Now let’s try to understand what fields we need to build this table. Going back to Draw.io or our ERD diagram, we can see all the fields and data types we need, but it doesn’t tell us what source it’s coming from. So we need to go back to the source to target mapping document again for our warehouse layer, and find our fact_sales table. There we can see that fact_sales is actually a combination of stg_orders and stg_order_details.

This is the query we have written to create this table…

```
WITH source AS (
SELECT
od.order_id,
od.product_id,
o.customer_id,
```

```
o.employee_id,
o.shipper_id,
od.quantity,
od.unit_price,
od.discount,
od.status_id,
od.date_allocated,
od.purchase_order_id,
od.inventory_id,
o.order_date,
o.shipped_date,
o.paid_date,
current_timestamp() AS insertion_timestamp,
FROM {{ ref('stg_orders') }} o
LEFT JOIN {{ ref('stg_order_details') }} od
ON od.order_id = o.id
WHERE od.order_id IS NOT NULL
),
unique_source AS (
SELECT ,
row_number() over(partition by customer_id, employee_id, order_id, product_id, shipper_id,
purchase_order_id, shipper_id, order_date) AS row_number
FROM source
)
SELECT
EXCEPT
(row_number),
FROM unique_source
WHERE row_number = 1
```

Next we have fact_purchase_order to create.

fact_purchase_order

So we do the same thing, go back to our source to target mapping document for our warehouse, and see what sources we need to create our fact_purchase_order table. We can see that we need quite a few tables here, mainly stg_customer, stg_employees, stg_purchase_order_details, stg_products, and stg_order. So let’s have a look at how we can build this.

Since our fact_purchase_order is a slightly complicated one, let’s review our ERD table again. We know we need our purchase_orders table and our purchase_order_details. We know from our source to target mapping document that we need customer_id, employee_id, and product_id which don’t exist within our purchase_orders or purchase_order_details. So going back to ERD let’s try to get our heads around how we can get those customer, employee, and product IDs. So purchase_orders has a relation to employees (meaning they share some kind of key) so that’s easy because we can just join the tables. Now to get to customers, we don’t have any relationship between purchase_orders and customers, so we have to figure out a path there. From purchase_orders we can get to purchase_order_details, and from there we can get to products, so that’s one issue solved. Then from products we can get to orders and from orders we can join to customers. Using this logic let’s build our table.

```
WITH source as(
SELECT
c.id AS customer_id,
e.id AS employee_id,
pod.purchase_order_id,
pod.product_id,
```

```
pod.quantity,
pod.unit_cost,
pod.date_received,
pod.posted_to_inventory,
pod.inventory_id,
po.supplier_id,
po.created_by,
po.submitted_date,
po.creation_date,
po.status_id,
po.expected_date,
po.shipping_fee,
po.taxes,
po.payment_date,
po.payment_amount,
po.payment_method,
po.notes,
po.approved_by,
po.approved_date,
po.submitted_by,
current_timestamp() AS insertion_timestamp
FROM {{ ref('stg_purchase_orders') }} po
LEFT JOIN {{ ref('stg_purchase_order_details') }} pod
ON pod.purchase_order_id = po.id
LEFT JOIN {{ ref('stg_products') }} p
ON p.id = pod.product_id
LEFT JOIN {{ ref('stg_order_details') }} od
ON od.product_id = p.id
LEFT JOIN {{ ref('stg_orders') }} o
ON o.id = od.order_id
LEFT JOIN {{ ref('stg_employees') }} e
ON e.id = po.created_by
LEFT JOIN {{ ref('stg_customer') }} c
ON c.id = o.customer_id
),
unique_source AS (
SELECT ,
row_number() over(partition by customer_id, employee_id, purchase_order_id, product_id,
inventory_id, supplier_id, creation_date) AS row_number
FROM source
)
SELECT
EXCEPT
(row_number),
FROM unique_source
WHERE row_number = 1
```

Let’s try to understand this. We have our stg_purchase_orders and we left joined that to stg_purchase_order_details. From there we joing to stg_products to retrieve the product_id. Then we join to stg_order_details and then to stg_orders to retrieve our customer_id.

Once we’ve created our table and looked at it in BigQuery, we might notice that some of the IDs are null. We need to correct this in our SQL query because we don’t want null values in our fact table. We can do this by just adding “WHERE o.customer_id IS NOT NULL” under the last ON c.id = o.customer_id. And re-run the dbt run prompt, and you’re done!

fact_inventory

This is the last fact table we need to create. The process is similar. We go back to our source to target mapping document for warehouse, and see which fields, and where do we need to build our table from. In this case it’s simple, all of our fact_inventory information is coming from. stg_inventory_transactions. So let’s go ahead and create that…

```
WITH source AS (
SELECT
id AS inventory_id,
transaction_type,
transaction_created_date,
transaction_modified_date,
product_id,
quantity,
purchase_order_id,
customer_order_id,
comments,
current_timestamp() AS insertion_timestamp,
FROM {{ ref('stg_inventory_transactions') }}
),
unique_source AS (
SELECT ,
row_number() over(partition by inventory_id) AS row_number
FROM source
)
SELECT
EXCEPT
(row_number),
FROM unique_source
WHERE row_number = 1
```

Then do dbt run -m warehouse.fact_inventory and boom, you’re done.

Partitioning

We have now successfully created our dimensional data warehouse layer.

But it’s not all done yet, because usually these fact tables are really large in a real-life scenario, so you want to make sure these fact tables are partitioned correctly.

So if we go into our fact_sales table and add this configuration before our WITH source clause…

```
{( config(
partition_by={
"field": "order_date",
"data_type": "date"
}
))}
```

What this tells our dbt fact_sales model, is to partition this table based on the order_date field. And that the data type for that field is ‘date.’ Then in our SELECT statement we’ve changed o.order_date to DATE(o.order_date) AS order_date meaning we’ve casted our order_date as a date field and aliased it as order_date. Then we re-create that table with our dbt run command.

So how do we know what to partition by? It’s very important to go back to the business and discuss the fields they will be using more frequently for reporting purposes and this is the field which should be applied in the SQL query filter. If you know a field the user will be filtering frequently by, that field should be used as a partition. It will help you save cost and speed up your queries.

We can do the exact same thing for fact_inventory by partitioning by transaction_created_date, and for fact_purchase_order by partitioning by creation_date.

Now we have successfully created our dimensional data warehouse layer and optimized our fact tables.

Physical Implementation (Analytics OBT)

So far we have created our data warehouse staging layer and our dimensional data warehouse layer, and now we need to create our reporting layer in the form of One Big Table (OBT). So let’s see how we can build that.

Again we come back to our source to target mapping document, this time for OBT. Here we can see, to build our OBT sales overview report, we need dim_customer, dim_employee, dim_products, fact_sales, and dim_date. So let’s see how we can build our OBT sales overview by using all of these tables.

Before we create anything though, let’s make sure our changes are pushed and saved to GitHub.

1. git status
2. git add models/staging/stg_products.sql
3. git commit -m “Removed ; issue from supplier_ids”
4. git add models/warehouse/
5. git add dbt_project.yml
6. git commit -m “Created Dimensional Data Warehouse Layer”
7. git push

You’re all done. Now we can start on the OBT layer.

obt_sales_overview

We start by creating a new file under analytics_dbt and calling it obt_sales_overview.sql.

Let’s now populate this table with everything we need based on our source to target mapping document for our OBT. By looking at the spreadsheet we see all the fields that we need. We can copy those and use them as our starting point for now. We can also see where that data is coming from, like dim_customer, dim_employee, dim_products, fact_sales, and dim_date. This is going to be our from field.

So we’re going to write our SQL query but we don’t be including dim_date for now because we’re not going to be doing any filtering such as filtering on fiscal year, etc. so we’ll skip that for now. We will also alias our customer and employee fields because they are basically the same.

```
WITH source AS (
SELECT
c.customer_id,
c.company AS customer_company,
c.last_name AS customer_last_name,
c.first_name AS customer_first_name,
c.email_address AS customer_email_address,
c.job_title AS customer_job_title,
c.business_phone AS customer_business_phone,
c.home_phone AS customer_home_phone,
c.mobile_phone AS customer_mobile_phone,
c.fax_number AS customer_fax_number,
c.address AS customer_address,
c.city AS customer_city,
c.state_province AS customer_state_province,
c.zip_postal_code AS customer_zip_postal_code,
c.country_region AS customer_country_region,
```

```
c.web_page AS customer_web_page,
c.notes AS customer_notes,
c.attachments AS customer_attachments,
e.employee_id,
e.company AS employee_company,
e.last_name AS employee_last_name,
e.first_name AS employee_first_name,
e.email_address AS employee_email_address,
e.job_title AS employee_job_title,
e.business_phone AS employee_business_phone,
e.home_phone AS employee_home_phone,
e.mobile_phone AS employee_mobile_phone,
e.fax_number AS employee_fax_number,
e.address AS employee_address,
e.city AS employee_city,
e.state_province AS employee_state_province,
e.zip_postal_code AS employee_zip_postal_code,
e.country_region AS employee_country_region,
e.web_page AS employee_web_page,
e.notes AS employee_notes,
e.attachments AS employee_attachments,
p.product_id,
p.product_code,
p.product_name,
p.description,
p.supplier_company,
p.standard_cost,
p.list_price,
p.reorder_level,
p.target_level,
p.quantity_per_unit,
p.discontinued,
p.minimum_reorder_quantity,
p.category,
s.order_id,
s.shipper_id,
s.quantity,
s.unit_price,
s.discount,
s.status_id,
s.date_allocated,
s.purchase_order_id,
s.inventory_id,
s.order_date,
s.shipped_date,
s.paid_date,
current_timestamp() AS insertion_timestamp,
FROM {{ ref('fact_sales') }} s
LEFT JOIN {{ ref('dim_customer') }} c
ON c.customer_id = s.customer_id
LEFT JOIN {{ ref('dim_employees') }} e
ON e.employee_id = s.employee_id
LEFT JOIN {{ ref('dim_products') }} p
ON p.product_id = s.product_id
)
SELECT *
FROM source
```

Then we will dbt run -m analytics_dbt and now we have successfully created our OBT sales overview. Now let’s go back to BigQuery and check if the creation has been successful.

Oops—we ran into an error and our table was created as a view under the data warehouse layer? That’s because we need to update our dbt_project.yml file with this…

```
analytics_obt:
+enabled: true
+materialized: table
+schema: obt_northwind
```

Now let’s continue with the process.

We need to create two more OBT tables, one for product_inventory and one for customer_reporting.

But when we were selecting our business processes there were four business processes, no? That’s correct, and the reason we only have 3 is because our sales_overview reports can now also serve the sales_agent report because we have all the customer and employee information in the same table so we don’t need to repeat the same process again for a different report.

product_inventory

Once again we go back to our dbt project and create a new SQL file. We’ll repeat the same process as before, we’ll come to our source to target mapping document, get all the same fields as before and put it into our SQL file and start making our way towards creating our OBT product inventory table.

```
WITH SOURCE AS (
SELECT
p.product_id,
p.product_code,
p.product_name,
p.description,
p.supplier_company,
p.standard_cost,
p.list_price,
p.reorder_level,
p.target_level,
p.quantity_per_unit,
p.discontinued,
p.minimum_reorder_quantity,
p.category,
i.inventory_id,
i.transaction_type,
i.transaction_created_date,
i.transaction_modified_date,
i.product_id AS ipd,
i.quantity,
i.purchase_order_id,
i.customer_order_id,
i.comments,
current_timestamp() AS insertion_timestamp,
FROM {{ ref('fact_inventory') }} i
LEFT JOIN {{ ref('dim_products') }} p
ON p.product_id = i.product_id
)
SELECT *
FROM source
```

Now when we view this table in BigQuery we can see there’s a loooot of NULL values, and we have an inventory_id associated with them. That’s why earlier we added that i.product_id as ipd alias earlier, to point out this issue so it makes things a little more obvious and helpful for debugging purposes. If we go back to BigQuery we can see that those ids associated with null values, do not exist in our dim_product table and that’s obvious if we scroll down. We can’t make an executive decision at this moment to see if we should remove this or not, but what we can do is validate this issue by going to our dim_products table and running a simple query to double check if that issue actually exists. So if we choose one of those random null ids, say 74, and then inside the dim_products table just look for that id with a simple SELECT query, and if nothing comes up, then we were right and the reason there are no values there is because those ids don’t exist within the dim_product table.

What we do at this point is we go back to the business, explain the situation, and double-check with them if they want us to retain these values or should we handle this within our OBT layer. For now we can leave it as it is.

customer_reporting

We follow the same process as before, and this is our SQL query.

```
WITH source AS (
SELECT
c.customer_id,
c.company AS customer_company,
c.last_name AS customer_last_name,
c.first_name AS customer_first_name,
c.email_address AS customer_email_address,
c.job_title AS customer_job_title,
c.business_phone AS customer_business_phone,
c.home_phone AS customer_home_phone,
c.mobile_phone AS customer_mobile_phone,
c.fax_number AS customer_fax_number,
c.address AS customer_address,
c.city AS customer_city,
c.state_province AS customer_state_province,
c.zip_postal_code AS customer_zip_postal_code,
c.country_region AS customer_country_region,
c.web_page AS customer_web_page,
c.notes AS customer_notes,
c.attachments AS customer_attachments,
e.employee_id,
e.company AS employee_company,
e.last_name AS employee_last_name,
e.first_name AS employee_first_name,
e.email_address AS employee_email_address,
e.job_title AS employee_job_title,
e.business_phone AS employee_business_phone,
e.home_phone AS employee_home_phone,
e.mobile_phone AS employee_mobile_phone,
e.fax_number AS employee_fax_number,
e.address AS employee_address,
e.city AS employee_city,
e.state_province AS employee_state_province,
e.zip_postal_code AS employee_zip_postal_code,
e.country_region AS employee_country_region,
e.web_page AS employee_web_page,
e.notes AS employee_notes,
e.attachments AS employee_attachments,
```

```
p.product_id,
p.product_code,
p.product_name,
p.description,
p.supplier_company,
p.standard_cost,
p.list_price,
p.reorder_level,
p.target_level,
p.quantity_per_unit,
p.discontinued,
p.minimum_reorder_quantity,
p.category,
po.purchase_order_id,
po.quantity,
po.unit_cost,
po.date_received,
po.posted_to_inventory,
po.inventory_id,
po.supplier_id,
po.created_by,
po.submitted_date,
po.creation_date,
po.status_id,
po.expected_date,
po.shipping_fee,
po.taxes,
po.payment_date,
po.payment_amount,
po.payment_method,
po.notes,
po.approved_by,
po.approved_date,
po.submitted_by,
current_timestamp() AS insertion_timestamp,
FROM {{ ref('fact_purchase_order') }} po
LEFT JOIN {{ ref('dim_customer') }} c
ON c.customer_id = po.customer_id
LEFT JOIN {{ ref('dim_employees') }} e
ON e.employee_id = po.employee_id
LEFT JOIN {{ ref('dim_products') }} p
ON p.product_id = po.product_id
)
SELECT *
FROM source
```

Finally we have created our OBT tables for the reporting requirements. This completes our requirement to build a staging layer, our data warehouse layer, and our one big table reporting layer.

Debugging (dbt)

The model we have built so far can be improved in many ways but as a first step, this is how to build many layers within your warehouse system and how to use dbt as your transformation tool.

It is also very important to understand how to do debugging within dbt.

Say we make an error. How would we fix that error? Whenever we do any sort of dbt command, behind the scenes dbt compiles a SQL code that you have given it, and all of those can be found under the target folder. Within the target folder you will see the compiled folder, and inside there you will see replications of the files inside your model folder (analytics, staging, warehouse). Within each of those, you can see the final compiled SQL query. So whenever there is a problem or any bug or issue, you can copy that code, go back to your BigQuery, and then try to run it. That’s the easiest way to do debugging.

Adding Tests (dbt)

If we look at our warehouse table, we have our dim and fact models. We want to make sure that whenever there is a new record added within our dim_customer table, that the ID is always unique and not null. Whenever there is a value that is null or not unique, we want to fail the transformation. How can we do that?

It’s very easy in dbt. We create a new file called schema.yml and within it, define the configuration such as this…

```
version: 2

models:
- name: dim_customer
- columns:
- name: customer_id
tests:
- not_null
- unique
```

All we need to do now is write a command called dbt test and run it, and this should do the test and let us know if the data that we have in our dim_customer table is going to be not null and unique.