# Data Modeling with Snowflake: A Practical Guide to Accelerating Snowflake Development Using Universal Data Modeling Techniques

## Part 2: Applied Modeling from Idea to Deployment

### Chapter 11: Putting Physical Models into Practice

* **The modern DW is a fast-paced environment**
* **Multi-source + near-real-time data in Snowflake streams + transforms at the speed that scalable virtual hardware will allow**
* **W/ potentially limitless computing resources available at trivially low prices, there emerges a tendency to undervalue *planning* in favor of post hoc adjustment**
* When this happens, platform + maintenance costs spiral, + suspicion is cast on the platform instead of the data model (or lack thereof)
* So tempting is **Snowflake’s promise of near-zero maintenance + effortless scalability** that many take it as an excuse to perform just *adequate* data modeling before diving in
* The Snowflake data platform does indeed live up to expectations (and beyond) *when the underlying data landscape is built on a* ***pre-planned data model***
* Compared to other data platforms, **Snowflake handles much of the DBA on the user’s behalf**
* However, **some considerations still fall to the database architects to ensure desired levels of performance + scalability while keeping costs in check**
* **The physical model will not only serve to deploy pre-existing conceptual designs but also helps in auditing existing databases**
* Main topics:
* Factors to consider for establishing physical properties
* Translating conceptual modeling elements to physical equivalents
* Creating the relational foundation for transformations and analytics
* Ensuring error-free deployment of a physical model to Snowflake
* Generating a diagram for an existing database

#### Technical Requirements for Local Snowflake Work

* The DDL for the completed physical model created through the exercises in this chapter is available to download + use from the following Git repository: <https://github.com/PacktPublishing/Data-Modeling-with-Snowflake/tree/main/ch11>
* Will arrive at the same result by following the steps in this chapter or by using the code provided
* The physical model will be used as the foundation for transformational examples in later chapters

#### Considerations Before Starting the Implementation

* When transitioning from a conceptual or logical design, where entities, attributes, relationships, + additional context have already been defined, there appears to be little to do at first glance when moving to a physical model
* However, **the specifics of Snowflake’s unique cloud architecture** (discussed in *Chapters 3* and *4*), from its variable-spend pricing to time-travel data retention, **leave several factors to consider before embarking on physical design**

##### 1) Performance

* **Query performance in Snowflake is heavily dependent on the clustering depth of the micro-partitions, which, in turn, are *influenced by the natural sort order of the data inserted***
* Apart from Hybrid Unistore tables (which allows users to enable **indexes**), **there are few performance tuning options left to the user *besides* sorting data before inserting + clustering**
* If the **data volume in a given table is expected to reach the TB range, Snowflake recommends defining a clustering key, which can be declared on 1 or more table columns or expressions**
* Where data is meant for downstream consumption, the **expected query patterns should be evaluated for performance + memory consumption against the intended warehouse size**
* If **aggregations + complex calculations are required**, Snowflake **materialized views (single-table) or manual materialization via CTAS (multi-table) should be considered**
* These **materializations will** **improve query performance compared to standard views by pre-aggregating + pre-joining data**
* Of course, **in a cloud platform, performance always comes at a price**

##### 2) Cost

* The **bulk of the cloud database costs will be incurred by query processing + data transformation**
* However, **data *storage* costs are *directly* associated w/ a physical design + should be considered at this stage before they accumulate unwittingly**
* As discussed in Chapter 4, **Snowflake tables offer a configurable data retention period to allow time travel**
* **Not *all* tables benefit from extended time travel periods** (some may forgo it altogether), so **be sure to set transient + data retention properties as and where needed** (table, schema, or database level)
* **Some data may not require a database table *at all***
* Recall **Snowflake can reference data in external storage, such as Amazon Simple Storage Service (S3), both in the internal + public clouds**
* **If *no* DML is required, external tables can provide a fully workable option w/out incurring storage costs**
* When DML *is* required for slow-changing datasets such as those used in data lakes, Snowflake is now offering support for external tables in Apache Iceberg format (in private preview as of this writing (2022) but expected in 2023)

##### 3) Data Quality and Integrity

* While there is no guarantee of data quality for data coming into the warehouse, the **physical data model should enforce critical business rules + preserve relational metadata**
* The **importance of relational constraints** is the central theme of this book but **NOT NULL + UNIQUE constraints, default column values, + proper typing of columns** **are** **equally important when designing the physical model**
* It is worth repeating that **NOT NULL is the *only* constraint enforced by Snowflake**
* **To ensure data integrity, user-enforced logic pre- or post-load should be considered**
* W/ the recent (November 2022) announcement of **join elimination**, the **RELY property for relational constraints plays an important role in query performance + cost reduction**
* The use cases + implementation of the RELY property will be covered during the discussion of “transformational modeling”, but the **foundation must be set in the physical model**

##### 4) Data Security

* The **physical database model should include security measures to protect the data from unauthorized access + to meet compliance regulations**
* **Snowflake’s access controls, roles, + object ownership** are outside the scope of this book, but model-related features are mentioned here for reference, along w/ links to further reading
* Bear in mind: **the following features are only offered in Snowflake’s Enterprise Edition or higher**:
* **Column-level security**: Allows the application of a **masking policy** to a column w/in a table or view + is available through **internal functions** or **external tokenization**
* <https://docs.snowflake.com/en/user-guide/security-column.html>
* **Row-level security**: Allows the application of a **row access policy** to a table or view to determine which rows are visible in the query result for a given role
* <https://docs.snowflake.com/en/user-guide/security-row.html>
* **Object Tagging**: Allows **data tracking** for compliance, discovery, protection, + resource usage
* **Tags** can be assigned to higher-level objects such as schemas + tables and inherited by their children (tables + columns, respectively)
* <https://docs.snowflake.com/en/user-guide/object-tagging.html>
* **Tag-based masking policies**: Allows protecting column data by assigning a masking policy to a tag + then setting the tag on a database object or the Snowflake account
* <https://docs.snowflake.com/en/user-guide/tag-based-masking-policies.html>
* **Data classification**: Auto-detects + categorizes (using tags) potentially personal or sensitive data to support compliance + privacy regulations
* <https://docs.snowflake.com/en/user-guide/governance-classify.html>
* **For all these data governance features, a visual data model becomes vital to their maintenance, management, + tracking**
* While Snowflake provides metadata on their application in the Account + Information schemas and through internal functions, **a navigable diagram of the physical database beats tabular text results for usability + understanding**
* For a complete overview of Snowflake access controls, refer to the documentation:
* <https://docs.snowflake.com/en/user-guide/security-access-control-overview.html>
* Having covered 4 primary considerations before embarking on a physical design (performance, cost, data quality + integrity, data security), it’s worth mentioning 2 more that, while an integral part of traditional database design, are now handled by Snowflake architecture

##### a) Non-Considerations

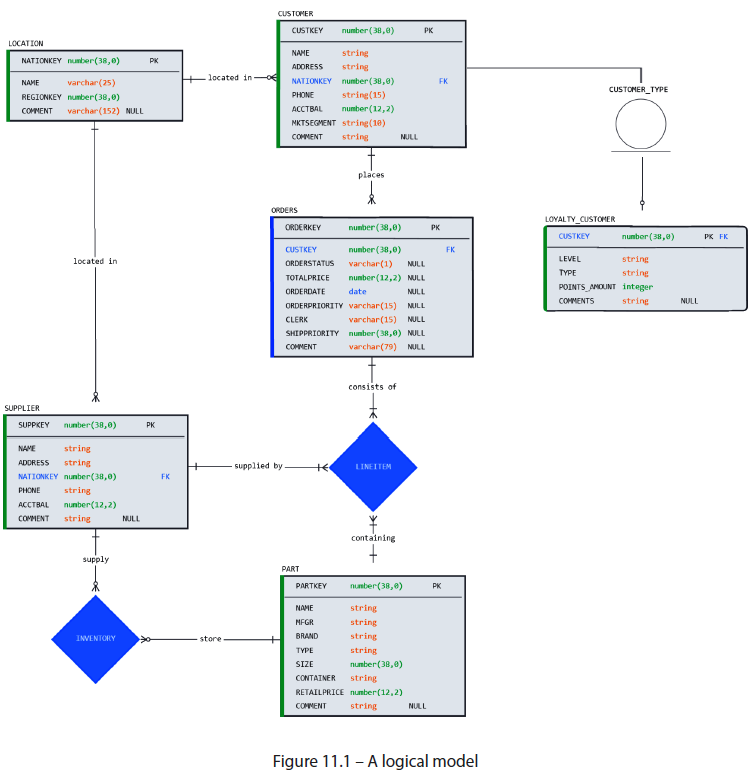
* **Snowflake’s cloud-native architecture delivers on the promise of near-zero maintenance by removing 2 critical factors from database design**:
* **1) Backup + recovery**: Using the **CLONE command**, Snowflake users can instantly **create zero-copy** **snapshots** of tables, schemas, or databases
* This command **can be combined w/ time-travel features to give users even more flexibility to take a snapshot *retroactively*** (see Chapter 3 for more details)
* **2) Scalability**
* For ***storage***, there is **no (practical) limit on the size of individual tables or entire databases**
* For ***compute***, **virtual warehouses are independent of the database model + can be resized instantaneously**

#### Expanding From the Logical to the Physical Model

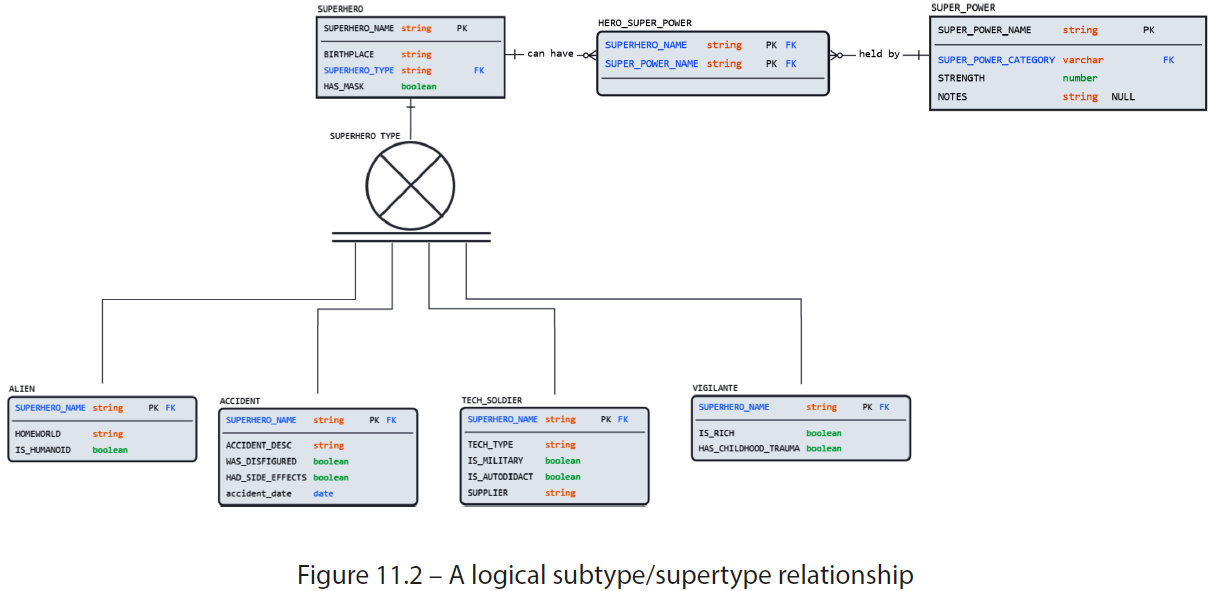
* At this stage in the modeling journey (prepping to transform a logical model into a physical one), the use of a **modeling tool** will make a marked difference in efforts required to generate the final DDL
* While this exercise can be done using anything from a sheet of paper to Excel, using a data modeling tool to accelerate the process is encouraged
* See the *Technical requirements* section of Chapter 1for a link to a free trial of **SqlDBM**, the only cloud-based tool that supports Snowflake + offers a free tier
* Picking up from the finished logical model from Chapter 8, let’s begin the physical transformation

##### Physicalizing the Logical Objects

* **Logical models contain all the information needed to transform them into a *physical* design, but they are NOT one-to-one equivalent regarding the number of elements**
* **Besides direct translations, such as entities to tables + relationships to constraints, some elements, such as subtypes, may not have a physical counterpart**
* Working from the completed logical design from Chapter 8, *where many-to-many relationships have already been resolved*, we are left to consider the subtype/supertype relationship between LOYALTY\_CUSTOMER and CUSTOMER:



* Recall that **subtypes represent inheritance between entity classes**
* **Before defining the physical structure, inheritance rules should be decided**
* Chihuahuas and mastiffs, despite obvious size differences, share many common properties of the “dog” class
* Likewise, **logical subtype properties are mutually compatible w/ the supertype** (both CUSTOMER and LOYALTY\_CUSTOMER have a CUSTOMER\_ID)
* **When deciding on a physical design, business requirements + query patterns will dictate whether the two should remain separate tables or fuse into one, + by what means if so**
* Perhaps our organization has an entire marketing department dedicated to working w/ loyalty customers + frequently maintains attributes that are exclusive to them
* In that case, a standalone LOYALTY\_CUSTOMER table makes sense from a maintenance standpoint
* If the organization were only concerned w/ keeping tabs on the % of loyalty customers relative to the total, a single CUSTOMER table containing loyalty attributes would be preferable
* **There are several options when deciding the physical representation of subtypes based on the business requirements**
* Recall the scenario of superhero subtypes from Chapter 2:



* Converting this arrangement to a physical model can happen in one of the following ways:
* **1) Maintain logical**: Keep all 5 tables + existing attributes as defined in the logical model
* **2) Inherit from supertype**: Keep all 5 tables + ***duplicate supertype attributes*** (such as BIRTHPLACE and HAS\_MASK) ***to the subtypes***
* Results in **DE-normalized supertypes** that are **easier to query but harder to maintain**
* **3) Rollup**: **Keep** *only* the ***supertype*** (SUPERHERO) by **inheriting the attributes** **from** the 4 **subtypes**
* **4) Rolldown**: **Keep** only the 4 **subtypes** by **inheriting the supertype’s properties** (such as BIRTHPLACE and HAS\_MASK)
* For this exercise, we will choose the 1st option (keep all tables) + maintain the logical definition, which results in 2 physical tables (CUSTOMER and LOYALTY\_CUSTOMER).
* Now that all the tables are known, it’s **time to adjust their physical properties**

##### Defining the Tables

* It is time to unify the lessons of previous chapters + put them into practice
* **This is the point in the design process where decisions about naming, such as case, plurality, prefixes, + suffixes, must be *finalized***

###### Naming

* This example will follow standard modeling best practices, reiterating the **caveat that every organization must choose what works best + aligns w/ any existing standards**
* The conversion yields the following changes from the logical model:
* **Case + table naming**: Convert to **singular + snake case** (ORDERS to sales\_order)
* **Unique identifiers**: Name unique identifiers **consistently** <table\_name>\_id (P\_PARTKEY to part\_id)
* **Remove prefixes + suffixes**: Applies to table + column names (P\_NAME to name)
* **Remove abbreviations**: Make columns unambiguous (P\_MFGR to manufacturer)
* **Add units to amounts**: Make amounts unambiguous by adding units (C\_ACCTBAL to account\_balance\_usd and L\_DISCOUNT to discount\_percent)
* W/ naming in place, it’s time to **determine the table properties for the entities** in our model

###### Table Properties

* **Considering storage costs, backup requirements, data sensitivity, + performance, table properties should be set according to business needs**
* Suppose our example will be used to manage daily operations at the historic AdventureWorkscompany
* In this case, we should opt for **standard (non-transient) tables that offer a fail-safe + an adjustable retention period**
* If 2 weeks of fail-safe is required for our data, we can set that default at the schema level to be used by all tables unless otherwise specified
* Because this is an ***operational* schema, access should be tightly controlled by the owner, so we opt for managed access**
* The database and schema declaration would look something like this:
* create database adventureworks;
* create schema operations

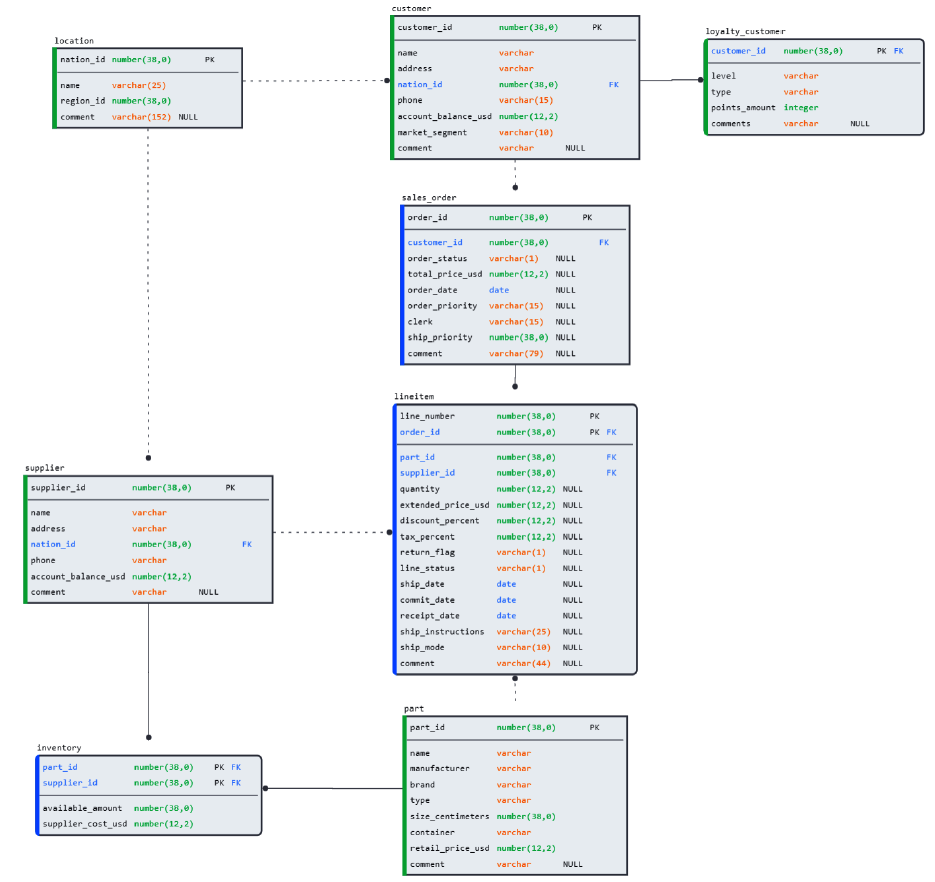
WITH MANAGED ACCESS

DATA\_RETENTION\_TIME\_IN\_DAYS = 14;

* **Although performance + data volume are a concern, Snowflake does not recommend clustering or re-partitioning *before baseline usage metrics are established***
* They **require 2 pieces of information before clustering: current query + DML performance and query + loading patterns**
* **After table properties, *column data types* are configured**
* Recall Snowflake accepts common data-type synonyms such as STRING + INTEGER and *automatically* translates them to their equivalents (VARCHAR and NUMBER, respectively)
* The final element that remains is **table constraints**

###### Declaring Constraints

* **Physical constraints translate *directly* from a logical model to help ensure data integrity + usability**
* **Although NOT NULL is the *only* constraint enforced by Snowflake**, PK, FK, + UNIQUE (AK/alternate key) **constraints provide valuable relational context + aid in query performance when used w/ the RELY property**
* <https://docs.snowflake.com/en/sql-reference/constraints-properties>
* **Snowflake query engine uses the RELY constraint property to eliminate unnecessary joins**
* *A deep dive into its use cases + implementation are in Chapter 12, so this discussion will focus only on its declaration*
* It should be noted that **there is NO monetary *or* performance cost to using the RELY property in tables that *don’t* violate uniqueness constraints, so for nearly *all* use cases, setting the RELY property is recommended**
* Following the previous chapter’s naming suggestions for relational constraints, we can now declare the PK, FK, + AK constraints for our tables
* To provide an example of an AK, we declare one on the NAME column of the LOCATION table
* Remember: To let database users know that another column (or columns) satisfies the conditions for a PK when a PK already exists, AKs can also be defined
* **AKs give database users insight into the granularity of a table + its various columns and carry the *same functional benefits as PKs***
* **To retain relationship names, remember to include them in the FK, as discussed previously**
* In summary, the following conventions are suggested:
* **PK naming**: Include table name (pk\_supplier for SUPPLIER table)
* **FK naming**: Include child table, logical relationship name, + parent
* Ex: fk\_sales\_order\_placed\_by\_customer
* **AK naming**: Include table name + qualifier as multiple alternate keys can be declared
* Ex: ak\_location\_name for the AK name in the LOCATION table
* **NOTE: AKs as a FK reference**
* **FK relationships can reference AKs (unique) as well as PKs**
* This fact is **often overlooked in database modeling** but **opens many more possibilities when establishing relations between tables**
* In cases where a surrogate key is declared primary, users can still reference the business key as a FK if declared as an AK
* The resulting **physical model** is now complete, + can be seen in the following diagram



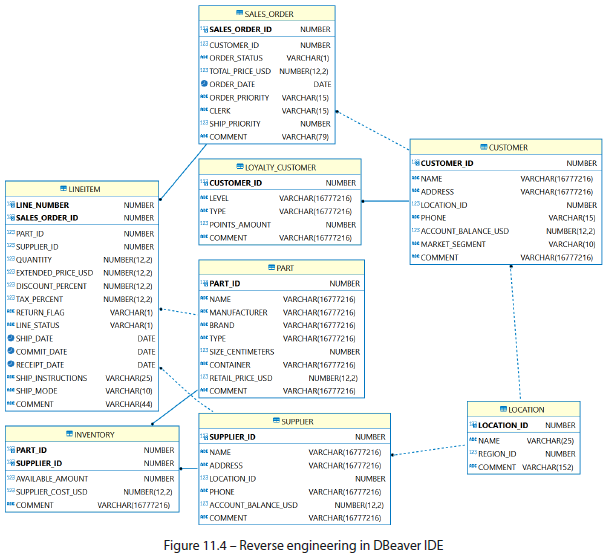
* With all the parameters configured, the physical model is ready to be deployed

#### Deploying a Physical Model

* At this point, all the tables, relationships, + properties have been defined + are ready to be deployed to Snowflake
* If you **use a modeling tool, all the DDL will be generated behind the scenes as adjustments are made to the diagram through a process called forward engineering**
* It is not strictly necessary to use a modeling tool to forward engineer, but doing so will make it easier to make adjustments + generate valid, neatly-formatted SQL for your data model
* *For those following the exercises, the forward-engineered DDL from this exercise is available in the shared Git repository mentioned at the start of this chapter*
* With the DDL in hand**, pay attention to the database and schema context in the Snowflake UI**
* **Creating a database or schema will automatically set the context for a given session**
* To switch to an *existing* database or schema, use the context menu in the UI or you can use the *USE <object> <object name>* SQL expression:
* use database adventureworks;
* use schema operations;
* Now, run the DDL you’ve generated (or use the example code provided)
* The physical model has been deployed to the database
* *But what happens if you want to visualize an existing database schema? Can modeling be used in reverse to help navigate unfamiliar databases?*

#### Creating an ERD From a Physical Model

* As just demonstrated through the forward engineering deployment process, **a physical database model is a one-to-one representation of its relational diagram (ERD)**
* This implies that the process of generating a diagram can be run in reverse, from Snowflake DDL to a modeling tool, in a process known as **reverse engineering**
* Again, it’s not strictly necessary to use a dedicated modeling tool, as many SQL IDEs such as VSCode + DBeaver can generate ERDs
* But doing so will offer greater flexibility in organizing, navigating, + making adjustments to a model
* A similar diagram to the one created in the previous exercise can be generated by connecting to our deployed model through a SQL IDE like below:



* What is evident in this exercise is often overlooked in database designs, which is the fact that **a neat, related, documented ERD is quickly rendered through reverse engineering *only when due diligence has been done to properly model its details***
* While some IDEs can render an ERD from a physical database, they do not have the editing capabilities of a modeling tool
* **W/out following proper modeling guidelines, any generated ERD will be lacking in practical value**

#### Summary

* The exercises in this chapter demonstrate what is required to transform a logical model into a deployable, physical design
* **However, before such transformation occurs, each project’s use case should be carefully considered**
* As there is **no one-size-fits-all guideline for Snowflake databases, decisions must be made considering performance, cost, data integrity, security, + usability**
* However, **unlike traditional databases, long-standing issues such as backup, recovery, and scalability are handled by Snowflake features and architecture**
* Once physical *properties* have been decided, users *create* physical equivalents of all logical objects, including many-to-many + subtype/supertype relationships, yielding a final set of physical tables
* Following this, naming standards, database objects, columns, + their relationships are declared *before* deploying the resulting model
* Deployable Snowflake DDL code is produced from an ERD through forward engineering
* As the physical model + its diagram are one-to-one equivalents, the deployment process is straightforward + can be run just as easily in reverse
* Generating ERDs from existing Snowflake databases can be done through reverse engineering, a process supported by modeling tools + many SQL IDEs
* **However, as the database structure and its physical diagram are equivalent, reverse engineering cannot miraculously create usable diagrams if proper modeling has not occurred**
* **i.e., Garbage in, garbage out**