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

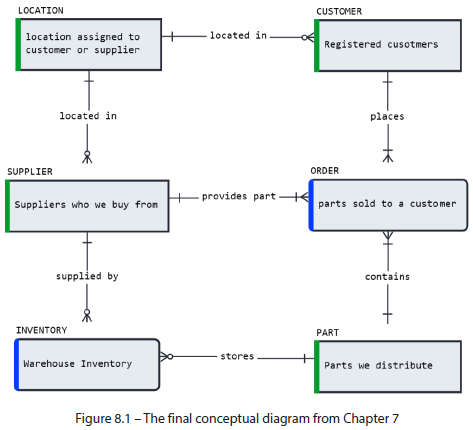
## Part 2: Applied Modeling from Idea to Deployment

### Chapter 8: Putting Logical Modeling into Practice

* In the previous chapter, we observed **data teams working w/ business teams to create a high-level conceptual model representing an organization’s main entities + relationships**
* While **conceptual models help to understand the overall structure + requirements of the data w/out going into excessive detail**, the **next stage in the modeling process requires us to go further and develop a *detailed* model to be used as a blueprint for moving to a *physical* database design**
* To complete the **logical model**, the **data team will have to collaborate w/ domain experts from the business once again to *expand* the list of entities, attributes, + relationships that will be used in the database, as well as the data types + constraints for each element**
* Just as the conceptual model held bidirectional benefits in both developing a fresh design and simplifying an existing one, **a logical model is not merely a stepping stone in the modeling process**
* **Logical models** are **much more detailed + granular than conceptual ones**
* Although they **lack the database-specific information of a physical model**, they make up for it by **providing contextual information that physical databases cannot capture**
* Once developed, **a logical model provides a basis for taking a design forward + adapting it for deployment on a specific database**
* Main topics:
* Identifying and adding **attributes** **+** **data types**
* Adding **structural + technical details** to **entity relationships**
* Resolving **many-to-many relationships**
* Expanding **weak entities**
* Understanding **inheritance** in database models

#### Expanding From Conceptual to Logical Modeling

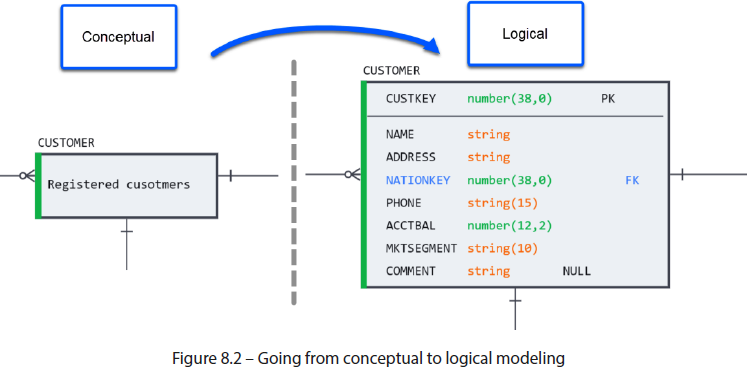
* We’ve used **Kimball’s 4-step methodology to develop a bus matrix + create a conceptual model based on the recorded information**
* **Details that informed the bus matrix were gathered through workshops + discussions between the data team + experts on the business side, who could elucidate the business’s operational model** andcreate a functional artifact: **the conceptual diagram**
* The following diagram shows the **conceptual model** as it looked at the end of the exercise:



* Despite the obvious deficiencies, such as missing attributes + Snowflake-specific object properties, **the conceptual diagram did *not* attempt to add any contextual detail on the functional relationships between the entities, such as subtypes + many-to-many associations**
* To uncover the relevant fields that should be added to our model’s entities, we should *again* engage business experts specializing in the related business domains

#### Adding Attributes

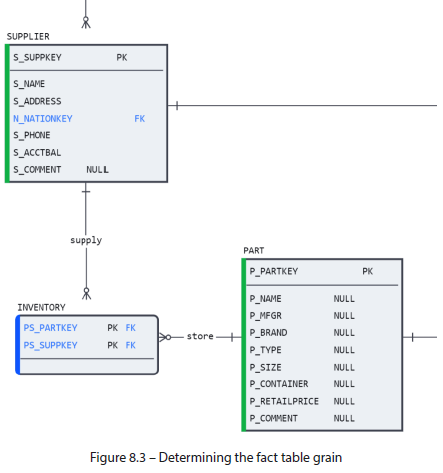
* The **most important detail in a dimension is its *unique identifier***, which **determines what constitutes a unique instance of each entity in our business**
* Examples of unique identifiers include things such as serial numbers for parts + employee IDs for a company’s human capital
* **Domain experts from each business area can confirm these + other necessary details**
* We use the CUSTOMER dimension as an example to identify the relevant details + incorporate them into the model:
* Suppose we sat down w/ the head of the sales team to learn about how our organization identifies customers + their relevant attributes
* The domain expert explains that besides typical customer attributes such as name + address, our organization is also interested in tracking account balance + identifying its market segment
* The sales team also explains how customers are grouped into regions based on their NATION attribute, maintained in a separate LOCATION dimension
* Besides identifying the attributes and their data types, we also learn that all of them are *required* (cannot be NULL), besides the optional comments field
* **With the feedback from the sales team, we now have a logical model of the customer dimension, complete w/ details about its PK, FK, data types, + their mandatory or optional nature**
* The transition from conceptual to a logical level of detail is shown in the following image:



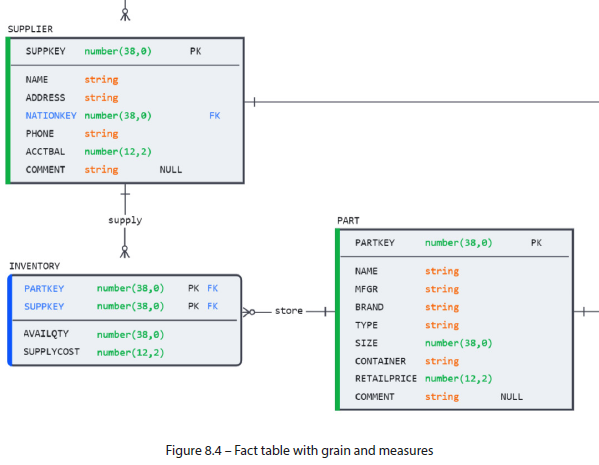
* After completing the exercise for all existing dimensions, we **focus on how they come together to record the facts**
* **NOTE:** ANSI compliance in Snowflake data types
* Snowflake supports most basic SQL data types in table columns as well as semi-structured and geospatial data types
* For compliance w/ American National Standards Institute (ANSI) standard SQL, **Snowflake also supports *synonyms* for many common data types found in other databases + logical modeling**.
* Ex: Although Snowflake stores string data as VARCHAR, a column could be modeled as STRING or CHAR, and upon deployment, such a column would be created as VARCHAR
* For a summary of supported data types + aliases, refer to the following documentation:
* <https://docs.snowflake.com/en/sqlreference/intro-summary-data-types.html>

#### Cementing the Relationships

* W/ attributes and PK + FK relationships established in the dimensions, we **again turn to the business experts to help flesh out the fact tables + determine their grain**
* *Building from the conceptual model*, we **ensure that ALL relationships have a defined granularity and optionality, and that the data teams understand their business context**
* ***Mistakes made in the fact table definition are the costliest to reconcile + could result in costly readjustment, so extra care should be taken***
* **Work w/ domain experts to ensure that the fact tables capture the *true* atomic grain of the information recorded by business operations + that the optionality is correctly understood**
* In the following example, the logistics team confirms that our warehouse separates parts by supplier to facilitate returns + inventory tracking
* In short, the warehouse can store many parts from many suppliers
* Role names are also verified + documented at this point for non-obvious relationships
* The resulting logical model now looks like this:



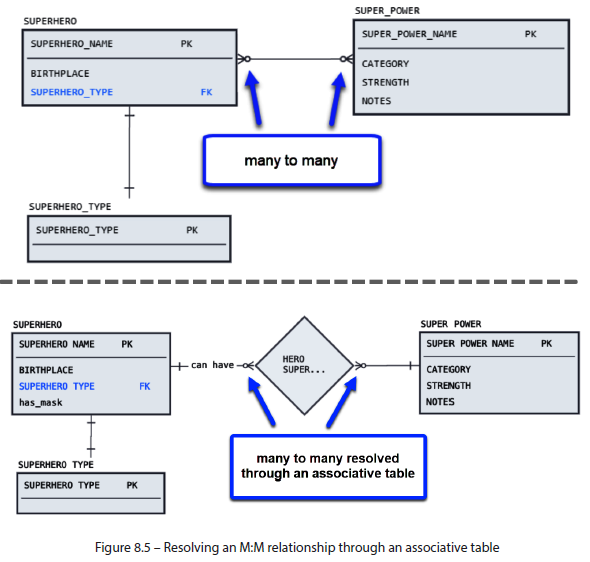
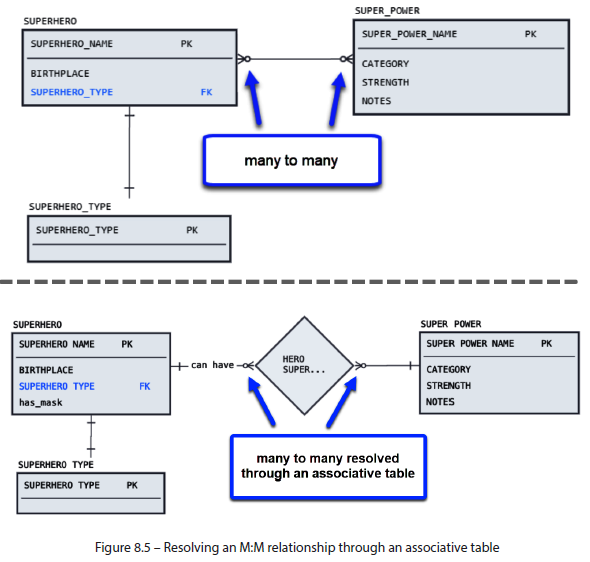
* ***However*, we have *yet* to define the *measures* describing the facts stored in the table**
* For warehouse inventory, this can be as simple as the quantity of the parts on hand + the cost for which each part was obtained (since this can fluctuate by supplier).
* The following diagram shows the *final* state of the INVENTORY table, complete w/ cardinality, granularity, role names, + required measures:



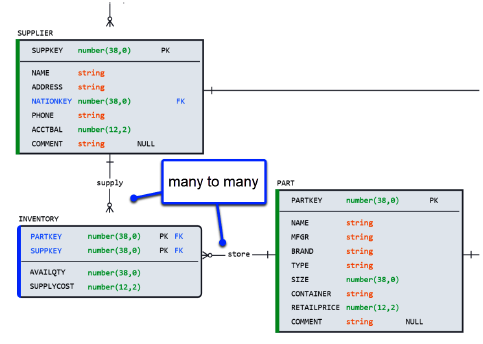
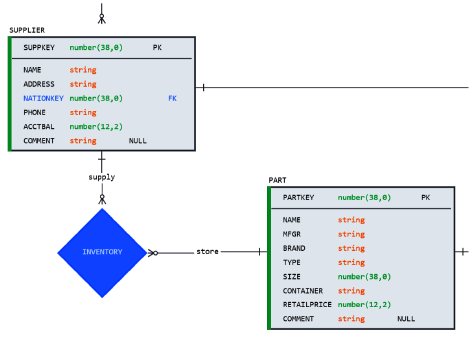
* **Repeating the exercise for all the facts in our model gives us a complete vision for proceeding to subsequent design phases**
* The next step is to identify the many-to-many relationships

##### Many-to-Many Relationships

* A **many-to-many (M:M) relationship** exists **between 2 entities when each can have many instances of the other**
* Ex: In our organization, suppliers can supply many parts, + parts can be supplied by many different suppliers
* Therefore, a 3rd entity (inventory), called an **associative entity** or **junction entity**, is required to store + track which part came from which supplier
* Recall the example from Chapter 2, where a list of all available superpowers required an **associative** **table** to represent the possible superpowers of a superhero:

🡺

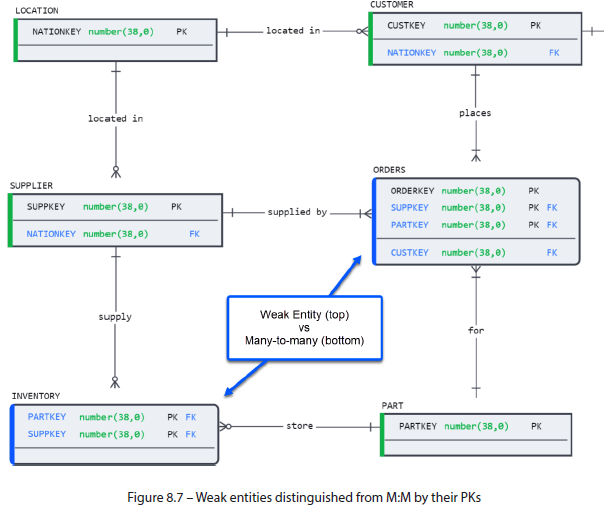
* In the *current* example, there are *no* unresolved M:M relationships b/c the real-world operation of our business necessitates a *physical* entity (the warehouse) to store an inventory of many parts provided by many suppliers
* **However, by using the diamond notation to capture the *business context* of this relationship, we communicate *beyond* the database level the idea that inventory *cannot exist on its own***
* Rather, **it only exists due to an interaction** (procurement) of a part from a supplier:

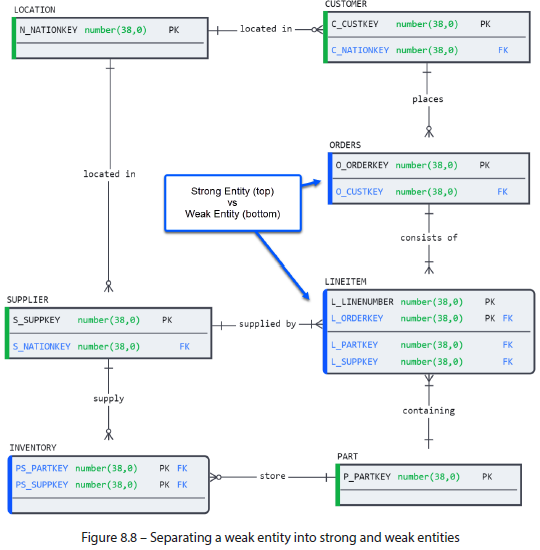
* **A M:M relationship is one example of an entity that *cannot exist in isolation***, + **weak entities are another**

##### Weak Entities

* A **weak entity** is an entity that **cannot be uniquely identified by its own attributes** **+ it must use a FK in conjunction w/ its own attributes to create a PK**
* i.e., a weak entity **has a FK *as part of its PK* (vs. a strong entity that depends entirely on its unique identifier)**
* On a diagram, weak entities are identified by rounded corners instead of the perpendicular edges of strong entities
* However, **their implications extend beyond semantics + they impact how the model handles changes in the data**
* Let’s look at an example using the ORDERS table.
* At first glance, ORDERS has multiple crow’s feet connectors, which make it seem like an associative M:M table
* Notice the similarities between ORDERS and INVENTORY in the following diagram:



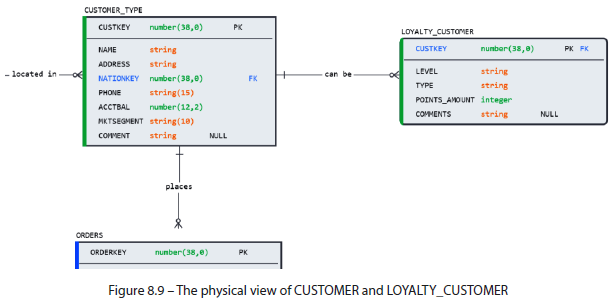
* Orders are business entities w/ unique codes that must be tracked + audited
* On the other hand, INVENTORY has *NO* unique identifier, nor does it make business sense to give it one (recording the PKs of the part, supplier, + quantity on hand is sufficient)
* After speaking to the sales team, we learn that ***an order is viewed differently depending on the business context***
* An order is a *single* transaction between a customer + the company *viewed from a sales perspective*
* However, from a *logistics* perspective, an order consists of *many* records to reflect individual items purchased, returned, or discounted
* Our **business teams need to reference + analyze individual orders, but keeping everything in one table makes that difficult**
* **The Solution is to separate orders into a strong entity (ORDERS) and a weak entity (LINEITEM), thereby facilitating storage, analysis, + maintenance**
* This means using the order identifier (ORDERKEY) as the PK of the ORDERS entity + creating a weak entity called LINEITEM, w/ its own unique identifier + previously existing FK columns as part of the PK and ORDERKEY as a non-PK, FK attribute
* The transition from a weak entity to separate strong + weak entities can be seen below:



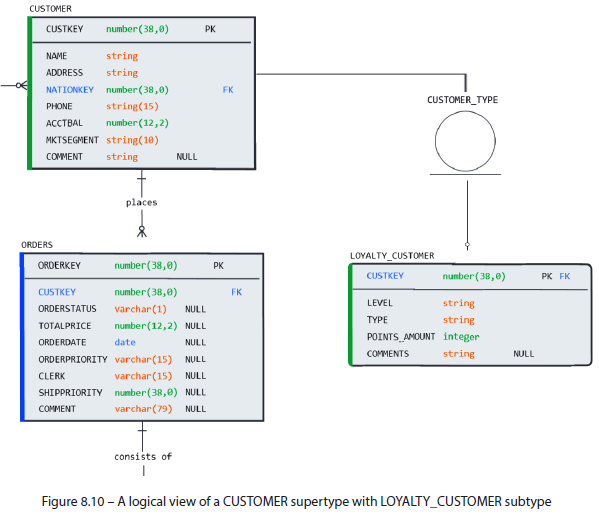
* In this scenario, **orders can be quickly reviewed + counted because each is stored in 1 individual record, + line items can be changed + updated in a *separate* table**.
* **While weak entities rely on strong entities to identify them, they’re inherently different elements**
* **What happens when we want to model entities similar in nature + share common characteristics?**

##### Inheritance

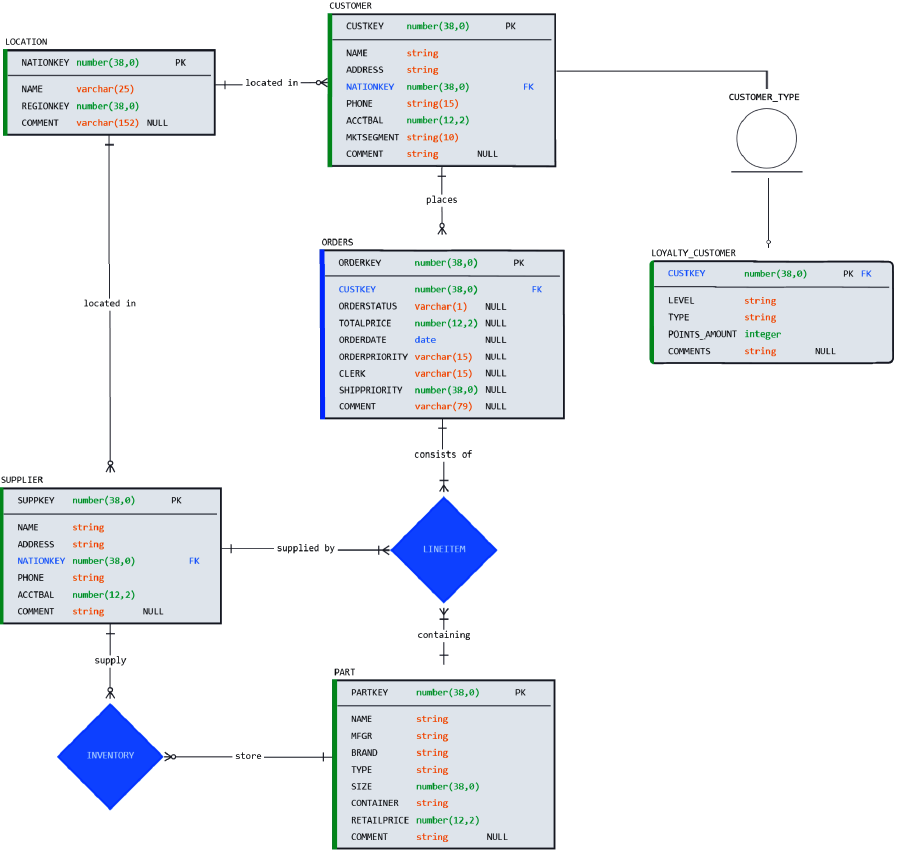
* Those with a background in OOP will be familiar w/ **inheritance**, which is the **passing of properties from a parent to a child class**
* The **same is true in data modeling, except that parent and child classes are called supertype and subtype, respectively**
* **Subtypes share common characteristics w/ a supertype entity but have additional attributes that make them distinct**
* Suppose our sales team maintains a particular category of loyal customers based on the volume of their repeat business
* A “loyal customer” has the same attributes as a “regular customer”, even sharing the same identifier, + contains attributes that describe their loyalty level + reward points
* At the physical level, the resulting table, LOYALTY\_CUSTOMER, may appear as an ordinary child table or weak entity when viewed on a diagram



* However**, this would not capture the true nature of its relationship to CUSTOMER as subtype notation would**
* Adding a **discriminator** **between the CUSTOMER supertype + the LOYALTY\_CUSTOMER** **subtype adds context that would otherwise be lost at the database level**:



* Compared to *Figure 8.9*, ***Figure 8.10* transmits the following business context**:
* CUSTOMER is the **parent/supertype entity whose attributes can be applied to its child/subtypes (defined by the discriminator symbol)**
* The LOYALTY\_CUSTOMER **subtype is incomplete (single, as opposed to double, the line below the discriminator), meaning more subtypes may be added in the future**
* The **LOYALTY\_CUSTOMER subtype is inclusive (the empty circle without an *X* mark), so a CUSTOMER can be a LOYALTY\_CUSTOMER** + **other possible types that are yet to be defined**
* This completes the transition from a conceptual to a logical model:



* **The conceptual model that we started w/ is a representation of the data requirements of our organization that is easy to understand + communicate to stakeholders**
* **It does NOT focus on the *specific implementation details* of the database but rather on the overall structure + relationships between different data entities**
* On the other hand, ***logical* database modeling creates a *detailed model* of the (pending) database that adds structural details and expresses the existing entities + relationships *in greater technical detail***
* **Goal of logical modeling = to lay the groundwork for designing the actual database while capturing additional nuances that cannot be represented using physical database objects**

#### Summary

* In this chapter, we continued the modeling journey by **taking the conceptual design** from the previous chapter + **expanding it to logical by adding structural + technical details that closely resemble the physical model we will create + deploy** in later chapters
* **Continuing to work w/ business experts from our organization, we identified the attributes that will be used to capture crucial business master data and transactions**
* We **defined each business entity’s identifiers, attributes + measures, and data types to do this**
* Once attributes have been set in place, we **reviewed the relationships between entities to understand the nuances of their associations to architect them in such a way as to fit the needs of the business + simplify their maintenance**.
* We **started by identifying the M:M relationships, which require an associative table to capture the interactions between the entities involved but are not considered entities themselves**
* **Associative tables**, therefore, **do NOT have a business key but rely on the FKs of their related entities**
* Next, we **checked for weak entities, which cannot be uniquely identified by their attributes alone**
* *Depending on the business needs*, weak entities may encumber the maintenance + analysis of the underlying entity, which can be split into separate, strong entities to overcome the problem
* Finally, we covered the concept of **inheritance** + how it **manifests in database design through subtype + supertype relationships**
* **Identifying subtypes + capturing their business context through a discriminator symbol helps set them apart from regular child tables**
* The next step in the modeling journey is **converting the logical model to a physical Snowflake database**
* However, before diving into the physical design, we **need to get comfortable w/ the concept of normalization + cover best practices for structuring + maintaining physical objects in Snowflake**