# 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 10: Database Naming and Structure

* In previous chapters, we **took a database design through the necessary steps to transform it into a logical model**
* While a **logical model is database-independent,** it is **also close enough to a physical design that it can easily be adapted + deployed to any database**
* Before tackling the Snowflake-specific properties of the data model, we get acquainted w/ naming conventions + database organization best practices that govern all database designs
* **Naming conventions are the guiding principles w/ which database objects are constructed**
* Consistent naming standards **reduce uncertainty for developers + help business users orient themselves w/in a database** + find the required data assets
* Beyond object naming, this chapter will also cover the **structure of the database itself, organizing it into logical groupings by schema, which improves usability**
* Finally, we will look at **database environments** and **replication** to ensure reporting data is isolated from volatile development changes.
* A theme that quickly emerges in every section of this chapter is the **focus on consistency**
* Rather than advocating for the *right* way to name or structure a database, it’s important to **be flexible + adjust standards to organizational and project needs, + then stick to those decisions**
* Main topics:
* Choosing a case standard that works best w/ Snowflake + maximizes cache reusability
* Object naming for tables + relationships for enhanced usability
* Styling + usage considerations to consider at design time
* Organizing a database through standard + managed schemas
* Structuring a database for OLAP *and* OLTP use cases
* Separating database environments for reporting, testing, + development

#### Naming Conventions

* Before creating the *physical* model, **naming conventions that govern its design need to be established**
* Following consistent naming conventions **improves understanding, reduces errors, facilitates collaboration, + generally makes it easier to work w/ your database**
* While there are many (often conflicting) theories + standards on the right convention to follow, the **most important thing is to choose one that is easy to understand + to use it *consistently* throughout your database**
* However, there’re **some general best practices to keep in mind when naming objects in *Snowflake***
* After all, **object names are like the API to a data model + should be regarded as a contract between the modeler + the data consumers**
* Once an object is created, downstream systems, users, + processes will reference it by name, forming dependencies + increasing the cost of future changes
* This section will cover some of the most crucial considerations in database naming, + instead of presenting naming conventions by object type, this section will be organized by importance, starting w/ those rules that are strongly advised to those merely suggested
* Whatever you decide, **choose a standard that best aligns w/ your organization and *stick to it***

##### Case

* Internally, **Snowflake stores all objects in uppercase *unless enclosed in double quotes***
* **For SQL queries, Snowflake’s parser treats anything *NOT* enclosed in double quotes as uppercase in order to match storage**
* In practice, this means *all of the following object names are compiled identically:*
* my\_table
* MY\_TABLE
* mY\_TabLE
* "MY\_TABLE"
* **Object names in *double quotes* are case-sensitive + may include spaces**
* Therefore, **any query that references an object in double quotes must *precisely match* the name** (i.e.,"mY TabLE" != MY\_ TABLE) **to return the expected result**
* **To avoid having to enclose *every* object + column in double quotes, it is advised to avoid them and use snake case instead**, where spaces are replaced w/ **underscores** (\_), + every word, *including the first letter*, is written in **lowercase** (e.g., my\_table)
* **While *object* names may be case-insensitive, the *results cache* is *not***
* Observe the following example where the same query is executed consecutively using various casing formats:
* --upper

SELECT O\_ORDERKEY, O\_TOTALPRICE FROM ORDERS;

* --snake

select o\_orderkey, o\_totalprice from orders;

* --mixed

SELECT o\_orderkey, o\_totalprice FROM orders;

* While **all 3 queries will return the same result, *NONE* will leverage the results cache (+ thus will all consume credits) b/c of the difference in case**
* **Whatever convention you settle on, ensure that the entire team is aligned on the chosen standard and stays consistent in object naming + keyword capitalization**

##### Object Naming

* This section will focus on naming conventions for the **objects** used most frequently in modeling + data analysis
* B/c users of a database + downstream tools will reference these objects for reporting + analytics, a **consistent naming convention will save time for everyone by ensuring that required data assets are discoverable + self-explanatory**

###### Tables

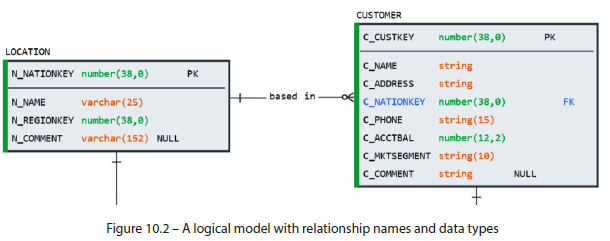
* Plural or singular for table naming? The database community can’t agree
* Proponents of plural (i.e., CUSTOMERS instead of CUSTOMER) argue that a table is a collection of many singular customer records + should therefore be named in the plural
* The singular camp counters that pluralization often introduces unnecessary confusion
* How to pluralize the person entity, for example, PERSONSor PEOPLE?
* **E.F. Codd’s relational model (the basis for RDBs) is consistent w/ first-order predicate logic**: **Tuples become rows**, + **relations become tables**, **all referred to in the *singular***
* Another **advantage of singular names is that they transfer cleanly from conceptual modeling**, where business entities tend to be thought of in the singular form (e.g., CUSTOMER, PRODUCT, COUNTRY)
* *At least, this is the academic opinion*
* **If still undecided, go w/ singular for table names**
* **But more importantly, be consistent**
* With table names covered, how about their identifiers?

###### Primary Key Columns

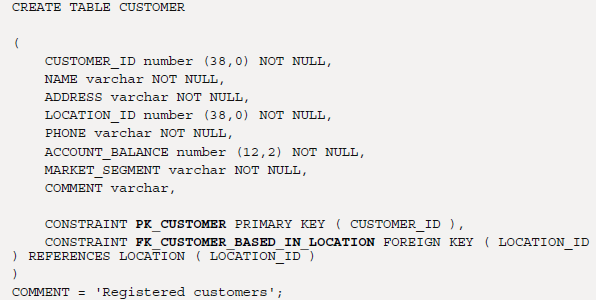
* Unlike tables, there is not much debate on naming their unique identifiers
* **PK columns should be named in *singular* format, following the table’s name**, as in <singularTableName>\_id
* Whether your table name is singular or plural (i.e., PERSON or PEOPLE), the unique identifier should be named person\_id
* **A generically named “ID” column for all tables is discouraged b/c it leads to confusion + ambiguity in downstream transformations**
* *But what about additional unique identifiers such as surrogate and business keys?*
* **As long as primary\_id is present, include additional unique columns following a convention consistent throughout the database**, which may include alternate ID suffixes such as “\_skey” or “\_BKEY”

###### Foreign Key Constraints

* **B/c relationship names are not supported in physical models, a descriptive + consistent naming for *FKs* becomes the best alternative**
* Take an example that features relationship names between entities:



* The example describes a M:M relationship where a CUSTOMER is *based in* a LOCATION
* **Although a physical model does NOT allow descriptions for FK constraints, *naming conventions* can be used to retain this information**
* Using a naming pattern of *FK\_<Child\_Table>\_<Relationship\_ Name>\_<Parent\_Table>* (and PK\_<Table\_Name> for PKs) results in a physical model that preserves the relationship details:



* While these are the most important considerations from a modeling perspective, there are a few more that should be considered

##### Other Suggested Conventions

* **Effective + consistent naming for *tables + relationships* is of utmost importance when designing a physical model**
* The following suggestions should *also* be considered for **clarity** + **adaptability** in column + object names:
* **Ensure consistent column names through conformed dimensions**
* **Conformed dimensions are columns whose domain contents + naming do not change when used across multiple tables**
* As data moves across the DW layers, **ensure that column names are descriptive enough in the base layer that they do not need to be renamed for clarity or conflict in downstream tables** (i.e., that the ID column does not become CUSTOMER\_ID then MAIN\_CLIENT\_CODE as it moves from source to reporting layers)
* **Avoid abbreviations in table + column names**
* Even if you think you will remember what the abbreviation stands for (you will not), the **other users in the system will have a harder time interpreting it**
* Does CUST\_REF imply *customer reference*, *custodial refund*, or *custom refrigeration*?
* ***Except for unique identifiers,* avoid prefixing columns w/ table name or initial** (e.g., C\_NAME in CUSTOMER table)
* This style guarantees having to rename columns upstream + makes lineage tracking difficult
* The benefit of this naming convention (which is unambiguous references when writing queries) can be achieved by using table aliases (C.NAME) or qualifying w/ the complete table name (CUSTOMER.NAME)
* **Avoid naming data sources using their object types** (e.g., views as <entity>\_V or materialized views as <entity>\_MV)
* Depending on cost + performance considerations, the object type of data sources may change over time + referring to a static <entity> will prevent errors in the objects that reference it
* **Instead, use suffixes to identify special-use tables** such as historical (\_hist), control (\_ctrl), and staging (\_stg)
* The naming suggestions discussed previously are:
* **Case**: Use unquoted snake case (lowercase w/ an underscore word separator)
* **Table names**: Singular
* **Primary key (columns)**: <singular\_table\_name>\_id
* **Primary key (constraint)**: pk\_<table\_name>
* **Foreign key (constraint)**: fk\_<child\_table>\_<relationship\_name>\_<parent\_ table>
* **Column names**: Be descriptive, stay consistent across tables, avoid abbreviations
* **Object suffixes**: Avoid adding suffixes such as \_v and \_mv to denote object type
* **Most importantly, remember these are general guidelines + you should adapt them to existing company standards + use cases, which may vary from one project or database to another**
* In short, you **can be flexible *between* databases if you are consistent *within* databases**

#### Organizing a Snowflake Database

* The **data objects stored by Snowflake (optimized + compressed in an internal columnar format) are *not* directly visible nor accessible by customers, + are only accessible through SQL query operations**
* The **customer only manages the logical grouping of database objects into schemas + databases**
* As described in Chapter 3, **in Snowflake cloud architecture, data is shared virtually w/out needing to be physically replicated**
* Therefore, ***unlike* traditional database platforms, the database structure in Snowflake is less concerned w/ the colocation of physical data + is more concerned w/ the logical grouping of objects, allowing for simple discovery + fine-tuning of access controls**
* *What does this look like in practice?*

##### Organization of Databases and Schemas

* **All Snowflake objects are assigned to a schema upon creation + form a logical hierarchy from object to schema to database**
* This **tiered grouping affords several benefits when organizing a database, including organization, access management, + replication**
* The **apparent advantage of database/schema tiering is organization**
* **2-layer classification allows Snowflake users to group objects meaningfully + granularly according to project needs**
* This **logical grouping makes it easy for users to find tables they are looking for or separate the working layers of a warehouse** (as described later)
* **Data sharing and replication** also benefit from having a well-organized database w/ schema grouping
* The **ability to clone objects individually or at schema + database level allows users to efficiently replicate datasets for testing + debugging**
* *Sharing* data *beyond* the account is possible, *but only at the database level*
* Another advantage of grouping objects into schemas + databases is its **fine-grain access control**
* **Assigning grants + ownership at the object, schema, or database levels affords Snowflake users great flexibility in managing + fine-tuning access controls to secure data access**
* Managing access control within a schema can be handled in 1 of 2 ways

###### Managed Schemas

* In **addition to** **traditional role-based access control**, Snowflake provides an **additional method for managing object access known as** **managed schemas, which delegate privilege management to the schema owner**
* In ***regular* schemas**, the **owner of an object** (i.e., the role w/ the OWNERSHIP privilege on the object) **can grant further privileges to other roles**
* **In *managed* schemas, the schema owner manages ALL privilege grants on objects in the schema**
* To create a managed schema, add the WITH MANAGED ACCESS parameter to the CREATE SCHEMA command like so: create schema my\_schema with managed access;
* **In managed schemas, *developers* can make changes + create new objects**
* However, ***only the schema owner can grant privileges to them***
* This approach is **ideal for security-conscious organizations** b/c it allows developers to work unhampered w/in a schema while delegating access management to a supervisory role

##### OLTP vs. OLAP Database Structures

* Suggestions for structuring a database will depend heavily on the application or use case that the database is designed for
* **OLTP (transactional) systems typically rely on *centralized* structures**, while **OLAP or DW’s segment their architecture into *layers***
* As Snowflake has just begun a concerted push into the OLTP space in 2022 w/ **Hybrid Unistore** tables, such use cases have had limited applications
* However, the **unifying theme in such transactional systems is that the database assets are *centralized* in a single main schema due to their *relational dependencies***
* Because **transactions affect multiple related tables, the logical grouping would NOT benefit usability + would only hamper usage**
* However, **in OLAP use cases (which make up the majority of Snowflake implementations), properly structuring the database WILL have a noticeable impact on usability**
* While there is **no one-size-fits-all suggestion**, the **following** **schema divisions should be considered**, starting with the landing area where data enters the warehouse

###### a) Source Layer

* **All DWs should contain a source/raw layer where data is loaded in its original, unaltered state from source systems**
* **Any renaming or transformation should be done *downstream* in subsequent DW layers to prevent having to reload in case of an adjustment in logic or business requirement**
* Depending on the amount of data sources the DW contains + the number of tables in each, it may even make sense to dedicate *an entire database* to source data + to separate source systems by schema
* **When source systems are separated by schema, tables can retain their original names**
* However, **when tables from various sources share the *same* schema, adding a source system prefix to the table name is recommended to identify + separate the sources like so**:
* salesforce\_account
* oracle\_account
* Although column names + their contents should NOT be modified at this stage, **standardized *metadata* fields may be added as long as this is done consistently for all tables in the schema**
* Such fields may include the following:
* **Load identifier**: can be used to remove data from a particular load if data corruption or other data quality issues are discovered
* A common technique is using a source system’s extract date/time to generate a unique value
* **Load date/time**: timestamp that indicates when the data was loaded into the DW
* If loading or data quality issues occur, this field may differ from the extract date/time of the source system
* **Source filename**: B/c data in Snowflake must be loaded from external file storage, recording the filename will help facilitate traceability + debugging
* **Loaded by**: When using various integration tools, recording the tool responsible for a given load will help point users to the location of logs, or the interface to retry failed loads is helpful
* **Load type**: If the loading pattern or method varies, it may help identify the load types, such as daily, scheduled, or ad hoc
* **A consistent naming pattern should be used to help distinguish DW metadata fields from source fields** (which may include similar metadata fields of their own)
* Common use cases include prefixing columns w/ a keyword or underscore as follows:
* etl\_load\_id
* \_\_load\_id
* As a rule, **access to the source schema should be limited to the central data or engineering team**
* B/c the **data has not yet been cleansed or conformed, business users will struggle to obtain meaningful results**
* However, if the data science team has sufficient business knowledge, they may benefit from direct access to the data in the source schema + glean meaningful insights w/out depending on the involvement of the DW team
* Once data is landed in the source layer, it’s time to clean it up for downstream consumption, and having a designated schema where this takes place is ideal

###### b) Staging Layer

* The **staging/working schema is intended as temporary storage for transformed source data as it makes its way downstream through the DW**
* The **tables in this schema are used for data loading + transformations, and should be off-limits to users + reporting tools**
* The advantages of grouping temporary tables in their own schema are as follows:
* **Cleanly segregates provisional data from trusted data**
* **Avoids having to prefix tables to make the distinction (i.e., stg\_table\_name)**
* **Allows schema-level default settings for like data retention + transient table types**
* Once the data has been cleaned up, conformed, + enhanced through the application of business rules + formulas, it can be shared w/ the rest of the organization through the approaches described in the following sections

###### c) Normalized Self-Service Analytics

* A **trusted + governed mid-level schema w/ clean normalized data can be a powerful tool for self-service + analytics**
* In organizations w/ high data literacy among analysts + business users (i.e., users capable of joining tables w/out help from the BI team or ChatGPT), **a schema that contains normalized (typically to 3NF) + conformed dimensions (attributes from multiple source systems unified in individual dimension tables) can accelerate self-service reporting + analysis by removing the bottleneck from the DW team**
* There is no standard for naming this schema, but common variants include CORE, SELF\_SERVICE, and ANALYTICS
* B/c the **data assets in this schema are meant to be analyzed + combined ad-hoc by everyone** in the organization, **due diligence must be applied to naming, documentation, auditing, stewardship, and governance**
* Therefore, **participation from domain experts + data governance teams is encouraged**
* The alternative (or add-on) to a lightly curated self-service schema is a *strictly governed* and *denormalized* reporting schema built for plug-and-play consumption

###### d) Denormalized Reporting Analytics

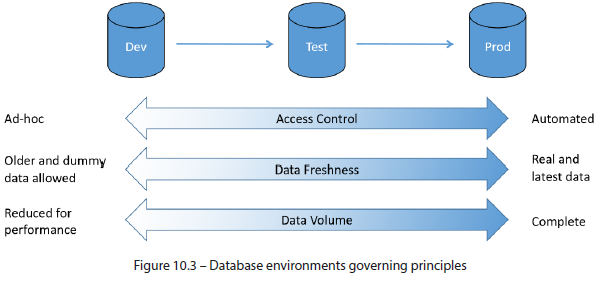
* Even in organizations w/ high degrees of data literacy among analysts + business users, **some analyses are too complex or too important to be done individually + must be centralized in a governed reporting schema**
* Data sources in this schema (typically named “reporting” or “analytics”) are **owned + maintained by the data team, who takes responsibility for their accuracy + reliability**
* Tables in the reporting schema tend to be *DE-normalized* (usually to 2NF) b/c the **emphasis is placed on having curated metrics + attributes in a single table to facilitate reporting**
* Although the DW team is responsible for building the reporting schema + keeping its contents up-to-date, the **governance team typically plays a vital role in establishing + enforcing business rules + ensuring that naming + concepts are used consistently throughout the organization** (i.e., if a “customer*”* is defined as someone having made a purchase in the last 12 months, then the definition is applied consistently to every table that includes a customer reference)

###### e) Departmental Data

* In even larger organizations, **business departments may have their own analytics teams + manage their own data sources**
* By **lowering the technical + operational barrier to entry to using a data platform, Snowflake has opened the door for business teams to take advantage of robust analytical capabilities w/out having to depend on the (typically) overburdened DW team**
* At a minimum, a **department should have its *own* database to load + manage the data sources that its team depends on**
* While some handholding by the data team may be required initially to get started, **a well-trained business department will quickly become self-sufficient**
* Once this happens, sharing + incorporating trusted data assets from a departmental database into the self-service or reporting schemas mentioned previously will become feasible
* B/c the data already exists w/in the same Snowflake account, it can be shared instantly w/ the rest of the company w/out duplication or the integration tools required to do so
* **Departmental databases can be thought of as miniature DWs unto themselves**
* Therefore, they will often include schemas separating source, stage, + reporting layers and separate schemas for sandbox, development, + testing
* **However, unlike a departmental database, a DW should be replicated + maintained across multiple environments to separate trusted data from that being developed**

##### Database Environments

* Now that we’ve covered the layers w/in the database of a DW, we should address the **separation of its development environments**
* This is NOT meant to be a comprehensive guide on database environment segregation, merely a best-practice outline users should follow + possibly augment w/ specialized automation tools
* **Separate database environments ensures that developers can make changes + test new projects w/out any risk of impacting *existing* data sources that the organization relies on**
* There should be **at least 3 environments:** **development (dev), testing, and production (prod)**, w/ some organizations separate testing into various stages such as **quality assurance** (**QA**) + **user acceptance testing** (**UAT**)
* The purpose of each is summarized here:
* **PROD**: **Governed, secured, + (near) up-to-date data** used to drive company reports + analytics
* **TEST**: A close copy of PROD where **changes are validated before being applied to PROD**
* **DEV**: A **sandbox environment** where developers are free to make changes + experiment
* **All environments should aim to be synchronized as closely as possible but vary in the degree to which productive standards are applied to them** (from most restrictive in PROD to least restrictive in DEV)
* The following are some of the criteria that should be observed + applied according to company needs and data volumes:
* **Access control**: listed above all other considerations for good reason
* ***THE* critical concern in managing product data and ensuring trust + correctness is controlling *who* has access to consume (users + reporting tools) + make changes (ideally, no one)**
* **In a PROD setting, human users should NOT be allowed to make changes or perform DML operations by hand**
* **All such activities should be orchestrated by tools that handle deployment + execution**
* Ensures processes are **automated** + **auditable** + require **little or no human intervention**
* **Data freshness**: a **PROD** environment should have the **most up-to-date information** required for operational decision-making
* A **PROD *snapshot*** from a given date may be **sufficient in a TEST environment**
* **DEV may benefit from a reduced sample size** (for performance) + dummy data (for security)
* **Data volume + exactness**: For maximum precision in DEV + TEST environments, the underlying data *should* match that of PROD
* However, **this would require additional maintenance + may hamper performance for simple changes that do not require a complete dataset**
* **Ideal balance should be struck depending on data size + the type of testing that takes place in each environment**
* **Warehouse size**: Although the warehouse is independent of the database contents**, testing + development should be done under similar conditions as those used in the PROD environment**
* Ex: Even if dev + testing are successfully performed w/ a medium warehouse, performance may prove unacceptable to PROD business users who are limited to an extra-small DW
* The following diagram illustrates these principles on a spectrum across database environments:



#### Summary

* We saw how **naming standards + an organized structure make a database easier to use + facilitate maintenance**
* But **every organization is different + must choose the standard that best suits their needs + aligns with existing conventions**
* Internally, Snowflake stores object names as uppercase, + its query compiler converts all unquoted names accordingly
* **It is recommended to use snake case for naming + stick to an established pattern to maximize the results cache utilization + to avoid enclosing every column + table name in double quotes**
* For a clean transition between logical + physical models, **singular table names are encouraged**
* The **same applies to columns**, which should be **named consistently across the entire database**
* Using descriptive naming patterns for FKs allows users to preserve logical relationship names w/in the physical model
* After object naming, attention turns to **database organization through logical schema groupings**
* Snowflake allows for a **2-tier hierarchy (database + schema)** to maintain data access and replication across databases
* Following the DW data flow, the 1st step is the **source/raw schema**, where tables from source systems are replicated by the engineering team exactly as they appear
* Next, preliminary table cleanup + renaming are performed in the **staging schema**, kept off-limits to users + reporting sources
* Once data has been renamed + formatted, it makes its way to the **analytical schemas**, where it is consumed by users + reporting tools
* A **normalized self-service schema** is encouraged for organizations w/ knowledgeable + tech-savvy business users, as it lets business users derive analytical insights w/out depending on the data team
* A **denormalized reporting schema** is used for governed reporting sources + more complex analyses
* For larger enterprises, business teams may also have their own Snowflake databases, which function like miniature warehouses + can be integrated w/ the previously-mentioned reporting schemas
* Lastly, different **database environments** should be used to ensure data quality + separate works in progress from officially validated + governed data
* From the least restrictive DEV to the production-like TEST to the securely regulated PROD, **database environments should follow a spectrum of increasing access control + data quality**
* To summarize even further, **DBA comes down to 2 main factors: Keep it simple + keep it consistent**