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

## Part 1: Core Concepts in Data Modeling and Snowflake Architecture

### Chapter 5: Speaking Modeling Through Snowflake Objects

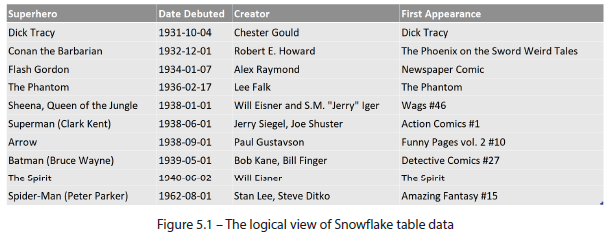
* In its purest form, **relational modeling** **(normalized tables w/ strictly enforced physical constraints) is most often found in *OLTP* databases**
* **Transactional databases** **store the latest (as-is) version of business information**
* Unlike **DW’s, which store historical snapshots + track changes in the information over time, allowing for additional (as-at) analysis across a temporal dimension**
* However, this does NOT mean that relational modeling concepts do not apply in an OLAP databases
* A **DW not only replicates *existing* entities + relations from transactional systems but *also* needs to manage the added task of conforming dimensions from other sources + joining them together in downstream transformations + analyses**
* Another reason to master the common language of modeling is the **Hybrid Unistore table**, which is poised to blur the line between a transactional system + DW and unlock use cases that have never before been possible in other architectures (**such as real-time, no-ELT-required analytical access to transactional data from the same platform that functions as both the operational system + warehouse**)
* **Main Topics**
* Ensuring efficient designs by understanding how **entities are represented as tables** + how Snowflake manages the underlying storage
* Exploring the benefits of **proper data typing for attributes**
* Understanding **database constraints** and managing them in Snowflake
* Learning how to reap the benefits of **identifiers** and **primary keys**
* Using **alternate keys** when multiple primary key candidates exist
* Tying the model together through **foreign key relationships**
* Specifying and checking for **mandatory columns**

#### Entities As Tables

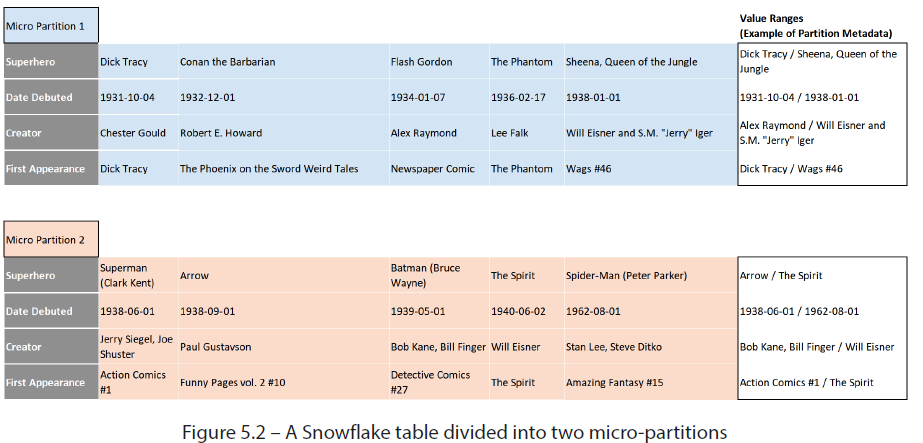
* Before delving into database details, recall the concept of an **entity** at the business level as **a person, object, place, event, or concept relevant to the business for which an organization wants to maintain information**
* In other words, an **entity** **= a business-relevant concept w/ common properties**
* **Identifying + naming entities** **rule of thumb** = conform to singular **nouns** (ex: customer, item, product, etc.)
* The **obvious candidate for storing + maintaining information in Snowflake is a *table***
* **Through SQL, tables give users a standard + familiar way to access + manipulate entity details**
* As seen, **Snowflake tables come in several flavors**, offering different backup + recovery options
* Besides selecting a table type that provides **adequate Time Travel + Fail-safe**, **Snowflake tables live up to the company’s claim of near-zero maintenance**, as there are **no indexes, tablespaces, or partitions to maintain *by hand***
* **The only 2 facets of storage over which users have control = the micro-partitions that Snowflake uses to store data in its internal columnar format + how they are clustered**

##### How Snowflake Stores Data

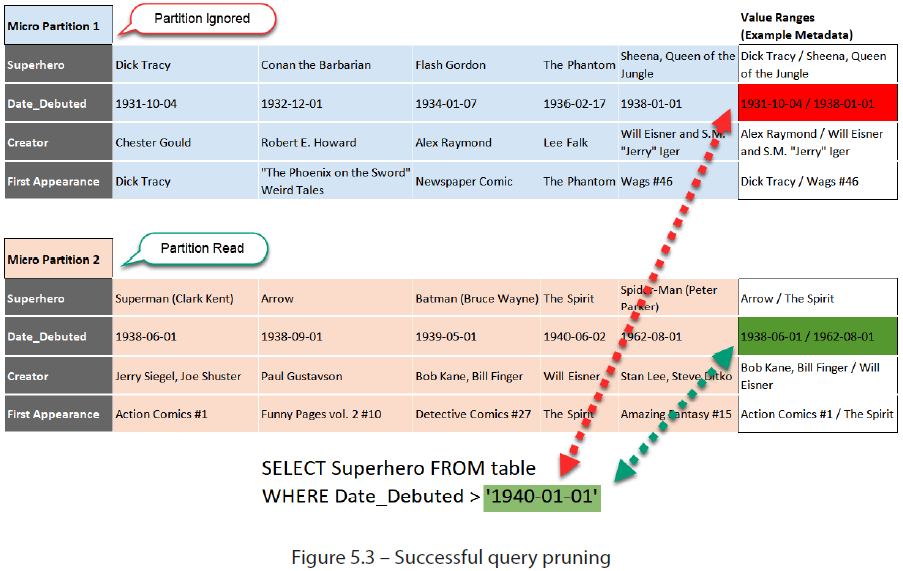
* **Data in Snowflake tables is stored in a columnar format using continuous storage units called micro-partitions**
* **Typically, databases divide large tables into physical partitions to improve performance + scalability**
* However, **physical partitions require maintenance + result in data skew (some partitions growing larger than others)**
* **Snowflake** escapes these limitations by **keeping the micro-partitions small (~50-500 MB, uncompressed) + managing them transparently through the services layer**
* **Micro-partitions are *immutable***, so **DML operations are more efficient + always create *new* partitions**
* Due to their small size, **granular statistics can be kept for individual columns w/in a micro-partition, + each one compresses optimally based on its data type**
* Unlike *traditional* partitioning, **micro-partitions *can* overlap in their value ranges (*but a single record can never be split across multiple partitions*)**
* **Micro-partitions are created by the natural ordering of data *as it is loaded into a table***
* Ex: If new transactions are loaded daily into a SALES table, transaction\_date would form the natural partition of the table
* If, on the other hand, the table was sorted by *client* instead, then partitions by *client* would be formed, + transaction\_date values would overlap
* **Main benefit of micro-partitions = their metadata (ranges of values, distinct counts, + others) allows the Snowflake optimizer to prune user queries**
* **Pruning** **avoids searching through micro-partitions where filter values are *known* not to exist**
* Ex: **Even if a table were to grow to millions of records + span thousands of micro-partitions, query results could still be returned in milliseconds by pruning unnecessary partitions + only scanning those whose value ranges match the query filter conditions**
* Using a simplified conceptual example, let us see micro-partitions in action
* The following table contains data about superheroes + their creators, + this is how it would appear to the user:



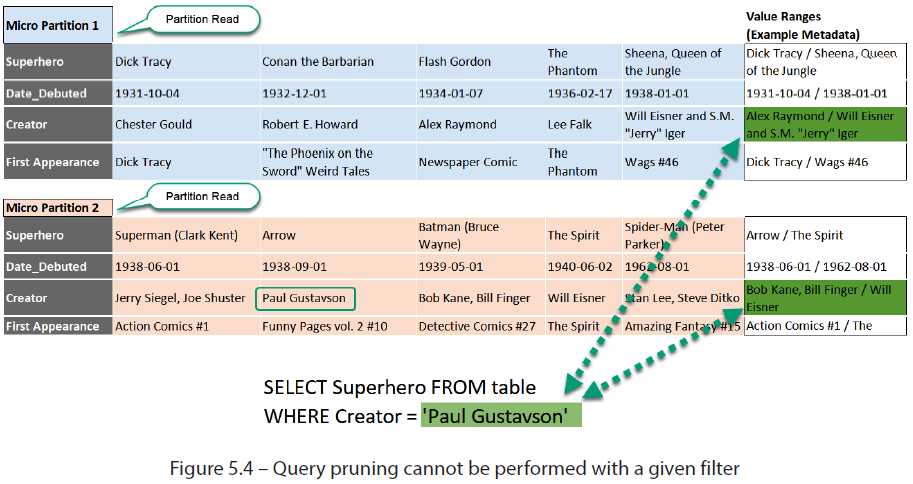
* ***Internally*, 2 micro-partitions are created, + the data is compressed using a columnar format:**



* Since the table is ordered by *[Date Debuted]***, when the services layer breaks the table up into micro-partitions, this column will have a natural clustering of values**
* While this example uses only 2 micro-partitions, **Snowflake tables can be composed of millions of micro-partitions, depending on their size**
* **Using partition metadata**, only some of which is shown above(**column value ranges**), **Snowflake’s query optimizer can prune or ignore partitions depending on the query filters**
* Below, the optimizer can safely skip reading partition 1because its value range (known from the metadata) does not correspond to the filter condition



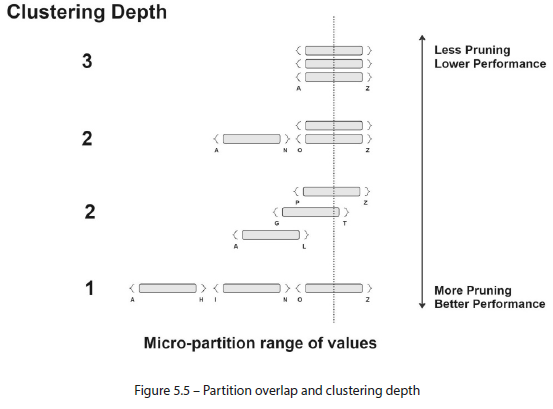
* **If the optimizer *cannot* eliminate any micro-partitions based on their value ranges, then *ALL* partitions + their contents need to be read to find matching records**
* Below, query pruning *cannot* be performed b/c both partitions match the search condition based on their value range, even though creator *Paul Gustavson* only exists in *partition 2*



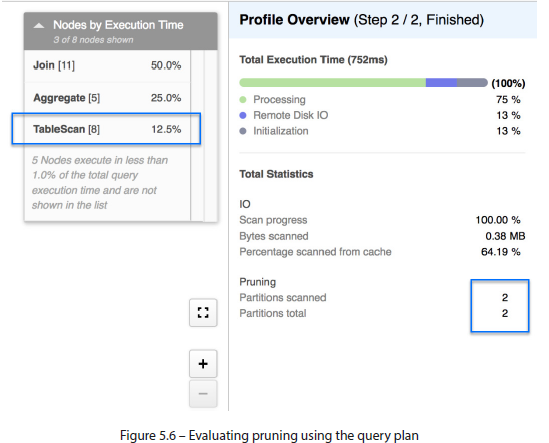
* ***How Snowflake manages micro-partitions depends on order in which data is loaded into a table***
* **Data can be ordered explicitly upon insertion using the ORDER BY** **clause or implicitly by the natural load pattern (ex: by CREATED\_DATE or LOAD\_DATE)**
* ***When partitioning matches search filters, query performance is improved through pruning***
* Ex: In a table w/ millions of rows, where data is loaded by LOAD\_DATE + queried for LOAD\_DATE = CURRENT\_DATE, performance will be *identical* to a table that only held data for CURRENT\_DATE
* ***However*, discounting this filter or searching for *any other attribute* would likely not benefit from query pruning**
* As shown in the preceding examples, **micro-partitioning creates multiple data segments w/ varying degrees of overlapping column values**
* This **data segmentation**, known as **clustering**, **can be used strategically to organize the table structure + improve query performance**

##### Clustering

* **As table data is broken up into micro-partitions, column values naturally group themselves into data segments**
* **Clustering** **is the *degree of variance of column values + their overlap across micro-partitions***
* **The less overlap is encountered between partitions, the higher the degree of query and column pruning that the optimizer can provide**
* **In addition to column metadata for micro-partitions, the Snowflake services layer also stores clustering statistics such as the following:**
* **Total number of micro-partitions** in each table
* **Number of micro-partitions containing values that overlap w/ each other** (in a specified subset of table columns)
* The **depth** **of the overlapping micro-partitions**
* The (simplified) example below shows **how 3 micro-partitions can overlap in clustering ranges**
* An *actual* table could have many more partitions + columns, + **achieving a perfect separation is neither likely nor required for improving query performance**



* **When considering how data is loaded + queried from a table, the clustering depth or degree of overlap (less is better for pruning + performance) should be considered as tables reach a considerable size**
* **Although ordering a table across 1 or multiple dimensions might improve query performance through pruning, the cost of the sorting operation may offset those gains**
* Generally, even tables w/ several million rows would provide adequate performance w/out any sorting or ordering applied to the rows
* However, **query performance may degrade after a certain table size if load pattern does not match search pattern** (Ex: data is loaded by TRANSACTION\_DATE but queried by CLIENT)
* **Users can assess query pruning information using Snowflake’s visual query profiler**
* The **total number of micro-partitions is displayed on the screen** alongside the **number of partitions accessed during a query**
* The screenshot below shows that the number of scanned partitions equals the total, meaning the optimizer could not perform *any* pruning



* As stated earlier, **a lack of pruning alone is NOT an indicator of poor query performance**
* **Query duration is subjective due to factors such as caching, query complexity, + data volume**
* **However, when query performance fails to meet expectations**, factors such as a **high table scan % relative to other query operations** (as shown above) **or a lack of pruning may indicate a need to re-cluster the table**
* **Re-clustering can be performed manually by recreating the table while applying the required sorting, OR can be handled automatically by Snowflake**

###### Automatic Clustering

* **Snowflake automatically handles clustering through the natural loading pattern of incoming data**
* ***However*, on *large* tables (multi-TB+), DML operations may degrade the clustering quality on desired columns + impact query performance**
* **Due to the large table size, *manual* re-sorting of the table would be costly**, so Snowflake provides the **option of automated re-clustering** of a table **through a specified** **clustering key**
* **Clustering key = 1+ columns (or an expression) that Snowflake uses to automatically sort the data in a given table + transparently *re-sort* it when necessary**
* **Benefits of defining a clustering key** include the following:
* **Improved query performance through pruning** by skipping data that does not match query filters
* **Better column compression + reduced storage costs**
* **Maintenance-free automated re-clustering performed by Snowflake**
* ***However*, remember that automated re-clustering operations incur the same processing costs as those performed manually**
* Therefore, **only consider setting a clustering key if performance is prioritized over** **cost** ***OR*** the **clustering performance offsets the credits required to maintain it**
* The following considerations help determine when the latter is likely to occur:
* The **table contains *multiple* TB of data + *many* micro-partitions**
* The **queries can take advantage of clustering**
* Typically, this means that 1 or both of the following are true:
* The queries need to read only a small % of rows in the table
* The queries sort the data (for example, using an ORDER BY clause)
* **A high % of queries can benefit from the same clustering key** by selecting or sorting on the same few columns
* Further information on clustering depth can be found in Snowflake documentation:
* <https://docs.snowflake.com/en/user-guide/tables-clustering-micropartitions.html>
* **Entities are the main elements of a relational model**
* We just saw **how to translate an entity into a physical table in Snowflake** + the **design considerations to contemplate for optimal performance + storage**
* Now we expand on the entity in by adding **properties**
* **Notes on clustering vs. partitioning:**
* <https://cloud.google.com/bigquery/docs/partitioned-tables>
* <https://cloud.google.com/bigquery/docs/clustered-tables>
* <https://stackoverflow.com/questions/17021203/what-is-different-between-database-clustering-and-database-partitioning>

#### Attributes as Columns

* Recall: **entity = a business-relevant concept for which an organization wishes to maintain information**
* Recall that **attributes** (defined *with* the business team during conceptual modeling *OR* loaded from existing source data during the ETL process) **are *properties* that describe the entity and are stored as columns**
* Can be **descriptive** (NAME, ADDRESS, QUANTITY, etc.) or **metadata** (ETL\_SOURCE, LOAD\_DATE, etc.)
* The **nature of the attribute** (whether numeric, string, date, or other) is **an essential detail for understanding the business requirement at the conceptual level + selecting the right data type at the physical level**
* Snowflake offers basic data types found in other databases (VARCHAR, DATE, INTEGER, etc.) and less-common ones (such as **VARIANT** and **GEOGRAPHY**), which offer exciting possibilities for modeling + working with table contents

##### Snowflake Data Types

* Snowflake is an **ANSI-compliant database** + supports the common SQL data types for storing strings, dates, + numbers
* *In addition*, **Snowflake provides more *advanced* types for semi-structured + geospatial data**
* When we discussed micro-partitions, **column compression (based on data type) was mentioned as a critical factor in performance + cost management**
* However, **using data types *correctly* can also yield usability benefits when querying table contents**
* Review the list of Snowflake data types in the documentation:
* <https://docs.snowflake.com/en/sql-reference/intro-summary-data-types.html>
* Numeric Data Types
* NUMBER (default precision and scale are (38,0))
* DECIMAL, NUMERIC (synonymous w/ NUMBER)
* INT, INTEGER, BIGINT, SMALLINT, TINYINT, BYTEINT (synonymous with NUMBER except precision + scale cannot be specified)
* FLOAT, FLOAT4, FLOAT8
* DOUBLE, DOUBLE PRECISION, REAL (synonymous with FLOAT)
* String & Binary Data Types
* VARCHAR (Default (and maximum) is 16,777,216 bytes)
* CHAR, CHARACTER (synonymous with VARCHAR except default length is VARCHAR(1))
* STRING (synonymous with VARCHAR)
* TEXT (synonymous with VARCHAR)
* BINARY
* VARBINARY (Synonymous with BINARY)
* Logical Data Types
* BOOLEAN (Currently only supported for accounts provisioned after January 25, 2016)
* Date & Time Data Types
* DATE
* DATETIME (Alias for TIMESTAMP\_NTZ)
* TIME
* TIMESTAMP (Alias for one of the TIMESTAMP variations (TIMESTAMP\_NTZ by default))
* TIMESTAMP\_LTZ (TIMESTAMP w/ local time zone; time zone, if provided, is not stored)
* TIMESTAMP\_NTZ (TIMESTAMP w/ no time zone; time zone, if provided, is not stored)
* TIMESTAMP\_TZ (TIMESTAMP with time zone)
* Semi-structured Data Types (*can contain other data types*)
* VARIANT, OBJECT, ARRAY
* Geospatial Data Types
* GEOGRAPHY, GEOMETRY
* Some data types offer properties to apply during table creation, like **collation** + **auto-increment**
* **Collation** lets users **set options such as case, accent sensitivity, + space-trimming for string comparisons**
* For numeric data types, **identity** and **auto-increment** options **generate unique sequences that come in handy for surrogate keys**
* **Besides storage + compression, adequate typing of attribute columns offers advantages from a *usability* perspective by ensuring consistent formatting during data loading**
* Ex: No stray characters in numeric columns or improper date formats in date
* Beyond that, **proper typing unlocks all relevant data type functions that Snowflake supports**, such as DATEDIFF (calculates the difference between 2 dates) for dates and STRTOK (**tokenizes** a given string + returns the requested part) for strings
* Complete SQL function documentation reference: <https://docs.snowflake.com/en/sql-reference-functions.html>
* Note: Storing strings in Snowflake
* Snowflake uses the VARCHAR data type to store string values, whose maximum length is 16 MB, or 16,777,216 characters/bytes
* How much data fits in 16 MB?
* How about the contents of Tolstoy’s *War and Peace* (587,287 words) 4X over?
* Interestingly, the 16 MB length is *also* the default
* But do not panic 🡪 **Snowflake efficiently compresses column contents to their *exact* lengths + does NOT bill for the 16 MB maximum**
* However, **sometimes it makes sense to *explicitly* limit a column to a fixed set of characters to inform users that its contents are not meant to exceed a specified setting** (Ex: setting a column containing ISO currency codes to VARCHAR(3))

##### Storing Semi-Structured Data

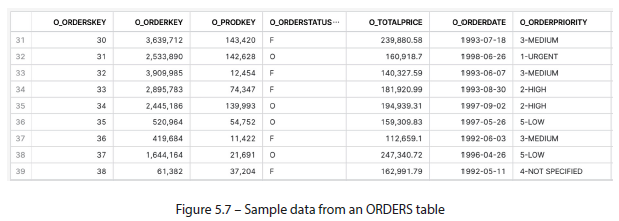
* Snowflake is widely recognized for its ease + agility in handling semi-structured data
* There are **3 dedicated data types available for storing semi-structured information** (VARIANT, ARRAY, OBJECT), + aside from **unique functions available to each** (discussed shortly), **they allow users to query their semi-structured contents using native SQL, w/out having to convert or flatten the data beforehand**
* Querying semi-structured data is discussed in later chapters + explained in documentation:
* <https://docs.snowflake.com/en/user-guide/querying-semistructured.html>
* **Snowflake tables can consist of structured *and* semi-structured data types, which can be queried natively in the *same* SELECT statement**
* ***However*, there are some differences between the 3 semi-structured data types + how they are used**, so let us understand what they are:
* **1) VARIANT**: the **recommended data type for hierarchical semi-structured data that allows users to load + query JSON, Avro, ORC, + Parquet data w/out converting or explicitly describing their structure**
* Has max length of 16 MB + **can store a value of *any other data type* (*except* VARIANT)**
* **Common uses include email logs, web activity, + event data**
* **2) OBJECT**: **analogous to a JSON object or dictionary (or hash, or map,** as known in other programming languages)
* **An object = a key-value pair, where the key is VARCHAR text that identifies the value and the VARIANT-type value itself**
* **Used when keys (names) convey meaningful information** (ex: a list of countries by ISO currency code used)
* Has a max size of 16 MB + **can store a value of any other data type, *including* OBJECT**
* **3) ARRAY**: similar to arrays in other programming languages, + **contains 0 or more (length is *dynamic* + NOT set explicitly) elements + references to their positions**
* Unlike an OBJECT, whose elements are accessed by (key) name, **ARRAY elements are referenced by their positions**
* Therefore, it’s **suitable for looping or processing data in natural/chronological order**
* An OBJECT has a max length of 16 MB + can store a value of any other data type, including ARRAY
* *However*, **ARRAY positions *contribute* to the 16 MB limit, so their *effective* size is smaller**
* We only briefly touched on the available options + functions surrounding Snowflake’s various structured + semi-structured data types
* For further reading on Snowflake data types, the following documentation provides a sectioned reference for each of them: <https://docs.snowflake.com/en/sql-reference/data-types.html>
* Having understood data types for defining attribute columns, we can move on to learning about the various **constraints that can be defined on them**
* However, before we can explore constraints in detail, we need to highlight one crucial factor regarding Snowflake’s management of them

#### Constraints and Enforcement

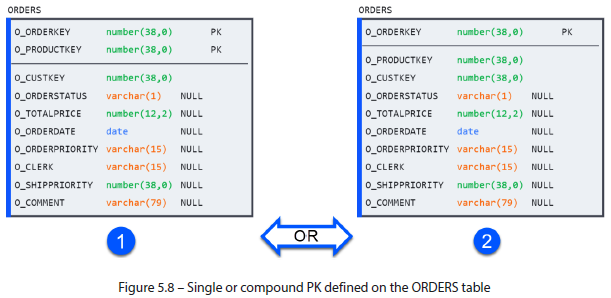
* **In the ANSI-SQL standard, constraints define integrity and consistency rules for data stored in tables**
* **Snowflake supports 4 constraint types:** PRIMARY KEY, UNIQUE, FOREIGN KEY, NOT NULL
* Since the function of each of these constraints is covered later in, for now we talk about their **enforcement**
* **Enforcement, on the part of the database, means actively monitoring the integrity rules of a given constraint when DML operations are performed on a table**
* **By enforcing a constraint, a database ensures that an error is raised when the constraint is violated, + the offending DML operation is *not allowed to complete***
* Ex: A NOT NULL constraint on a column indicates that a column cannot contain NULL values
* By *enforcing* this constraint, Snowflake raises an error if an operation tries to insert or update a NULL value in that column.
* The **NOT NULL** example was chosen strategically because out of the four existing constraints, this **is the only one that *IS* enforced**
* The **other 3 constraints are orientative, meaning while they’re not enforced (i.e., they can be violated w/out resulting in an error), they inform users of some valuable metadata details (discussed later)**
* ***At least, this was the case before Hybrid Unistore tables were announced***
* Hybrid Unistore tables have a completely different **HTAP architecture, allowing Snowflake to enforce all 4 constraints**

#### Identifiers as Primary Keys

* **Tables store information for business entities using attributes of relevant data types**
* A row in a CUSTOMER table holds information for a given customer, and a row in an ORDERS table represents an order. *Or does it?*
* Perhaps in this example, orders can contain *multiple* products + span *just as many rows*
* **To determine a unique instance of an entity, an identifier, or primary key (PK), if referring to a physical database, is used**
* **A PK is a column/set of columns whose values uniquely determine an instance of an entity, and *only one PK can be defined per table***
* **From a business perspective, a PK represents a single entity instance**
* So, is an order a single row containing 1 product, or does our organization allow *multiple* products per order?
* **PKs inform database users of what that *reality* looks like *at the table level***
* The following shows some sample data from a fictitious ORDERS table:



* There appear to be no duplicate values for the O\_ORDERKEY column, but this is a small sample from a table w/ millions of records
* **It’s impossible to tell by looking at the data what an order is uniquely identified by**
* O\_ORDERKEY *only*, O\_ORDERKEY *in conjunction* with O\_PRODKEY, or some other combo of columns?
* **When a PK consists of *multiple* columns, it is called a composite or compound key**



* **Even reviewing *every* record in a table + finding no duplicates does NOT guarantee one won’t arrive later if the business model + the corresponding data model are configured as per scenario 2 above**
* ***Only* by establishing a *conceptual identifier* + defining it *physically* in the corresponding Snowflake table can users definitively know how to isolate a unique entity instance within it**

##### Benefits of a Primary Key

###### 1) Determining Granularity

* Once data has been populated in a table, the need to query individual entities or aggregate entity statistics immediately follows
* *However, how can the users tell how many rows constitute a single record?*
* Is an order stored in a single line, or does it generate 1 line per product that it contains?
* In other words, **what is the lowest level of detail?**
* A **PK answers this question by defining the granularity of the table + telling users the lowest level of data it contains + how to uniquely identify an entity instance**
* **Knowing the PK of a table allows database users to query the data *precisely* on the required column(s) + values and eliminate ambiguity** (of having multiple “Dave” records, for example)
* The **precision of isolating a unique entity also applies to aggregating table data for counting and summarizing its metrics (numerical attributes)**
* Snowflake offers various aggregating functions to summarize table data (COUNT, SUM, MAX, etc.) but they require the user to specify the entity’s unique identifier (or other grouping dimensions) in the GROUP BY statement
* **Incorrectly specifying (or assuming) a valid identifier of a table will result in queries that return erroneous counts + aggregates**

###### 2) Ensuring Correct JOIN Results

* Queries often require data from one table to be joined + enriched by information from another
* While there’re many types of joins (LEFT, INNER, FULL, etc.), **users tend to avoid Cartesian or CROSS JOIN**, which multiplies the rows from one table by number of matching rows in another
* **Cartesian joins most often occur due to the inability to correctly identify a unique record in one or both tables involved in a join**
* In a database, where queries may operate on millions of records, unexpected record explosions could result in billions of extra rows + a meaningless final result
* Such queries can run exponentially longer than expected under correct JOIN conditions
* In any database, this is a **total waste of time**, but in Snowflake, this also comes with a **hefty credit spend**

###### 3) Avoiding Duplicated Values (Hybrid Unistore)

* **A PK uniquely identifies an entity**, so therefore, **duplicate values in PK columns would violate this constraint + should be disallowed**
* ***However*, ONLY Hybrid Unistore tables can enforce this rule**
* They can do so **through an index that allows Snowflake to efficiently scan for existing values when performing a DML operation**
* **If using *regular* Snowflake tables, ad-hoc validation can be performed before/after a DML operation to test for duplicates**

##### Specifying a Primary Key

* A **PK** can be **specified when creating a Snowflake table in 1 of 2 ways**: **inline, next to the corresponding column(s) in a CREATE TABLE statement**, or **out of line, at its end**:
* **New table inline**: declared directly next to a corresponding column when creating a table
* ***Only valid for single-column PKs and cannot be used to define composite keys:***

CREATE TABLE ORDERS (

O\_ORDERKEY number(38,0) CONSTRAINT my\_pk\_name PRIMARY KEY,

O\_PRODUCTKEY number(38,0) NOT NULL,

O\_CUSTKEY number(38,0) NOT NULL,

-- < rest of columns>

O\_COMMENT varchar(200) COMMENT 'Order details'

);

* **New table out of line**: allows Snowflake users to **specify a PK at the end of a CREATE statement + include *multiple* columns as part of the definition**:

CREATE TABLE ORDERS (

O\_ORDERKEY number(38,0),

O\_PRODUCTKEY number(38,0),

O\_CUSTKEY number(38,0) NOT NULL,

-- < rest of columns>

O\_COMMENT varchar(200) COMMENT 'Order details',

CONSTRAINT my\_pk\_name PRIMARY KEY ( O\_ORDERKEY, O\_PRODUCTKEY )

);

* **Existing table out of line**: PK can still be defined for *existing* tables by performing an **ALTER TABLE statement + specifying it like an out-of-line constraint**:

ALTER TABLE ORDERS

ADD CONSTRAINT my\_pk\_name PRIMARY KEY (O\_ORDERKEY, O\_PRODUCTKEY);

##### Keys Taxonomy

* Having seen the vital role that PKs play in defining + querying database tables, it is time to drill down into the **various kinds of keys that exist** to understand their taxonomy **+ the roles they play when designing tables**
* Although the *function* of a PK remains unchanged (to uniquely identify a record in a table) the **language of modeling allows for distinctions based on the nature of the key columns**

###### a) Business Key

* **Business key = a PK whose values hold meaning or significance within the organization**
* ***Any random group of characters can serve to identify an entity instance as long as they are unique***
* What **sets the business key apart is that its *values are significant in themselves***
* All the examples from this chapter have been business keys
* Ex: The values in O\_ORDERKEY and O\_PRODKEY are understood w/in the organization in operational teams + their systems, + the same codes are referenced, for example, in the CRM system + in other tables where they exist
* If used as PKs, a SSN or an ISO country code would also qualify as business keys b/c of their broader significance in the outside world
* **When modeling columns that contain business keys, a prefix or suffix of BKEY is often used to designate the distinction**
* However, **a key need not have business significance to be useful in table design**

###### b) Surrogate Key

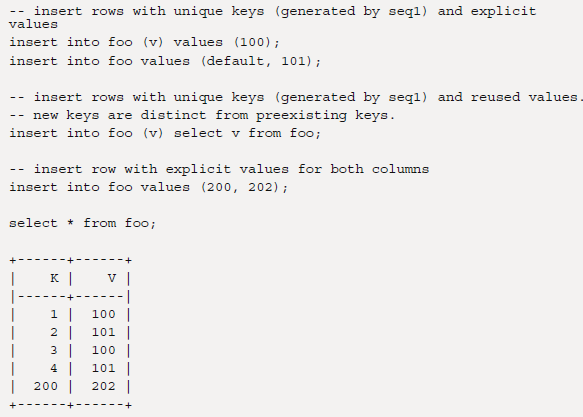
* Unlike a business key (holds business significance) a **surrogate key = a PK whose value holds NO special meaning**
* They are **typically created using random characters, column hashes, or sequential integers, and they satisfy the condition of uniqueness while their values carry no intrinsic meaning**
* Tables often have compound PKs w/ numerous member columns, which makes them tedious to type when, for example, needing to JOIN one table to another
* **Surrogate keys get around this inconvenience because, by nature, they are single-column and have a standard value format** (Ex: column hash or sequence)
* Some data models (such as Data Vault 2.0, discussed in a later chapter) rely on surrogate keys to ensure strict naming + value patterns to make table design repeatable + consistent
* **When modeling columns that contain surrogate keys, a prefix or suffix of SKEY** **is often used to designate the distinction**

##### Sequences

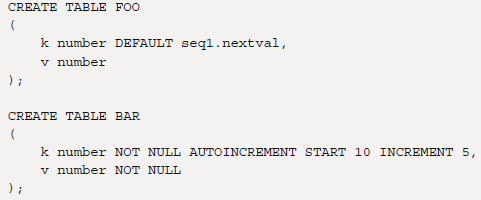
* When creating a *sequential* surrogate key, Snowflake provides a mechanism for generating it called a **sequence**, **an independent database object that generates sequential (but *not necessarily gap-less*) integers, ensuring unique column values**
* A **start number + increment can be specified when creating a sequence**
* **As *independent* objects, a single sequence can be shared by multiple tables or manipulated through function calls**
* **Sequences can be used as surrogate keys by assigning them as the default column value when creating a table**
* To do so, use a sequence’s **nextval function** as the column default like so:
* CREATE OR REPLACE sequence seq1;

CREATE OR REPLACE TABLE foo (k number default seq1.nextval, v number);

* The following example demonstrates how new records would behave under various conditions of the above nextval function:



* Alternatively, Snowflake can create + manage the sequence object transparently for a given table by using the **AUTOINCREMENT** or **IDENTITY** **keywords** when creating a column:



* For more information on using sequences, consult Snowflake’s documentation:
* <https://docs.snowflake.com/en/user-guide/querying-sequences>
* We now understand the PK taxonomy + **can distinguish a business key, which has business significance, from a surrogate key, which is not meaningful in itself + whose sole purpose is to provide a unique value**
* However, recall that **a table can only have *ONE* PK defined**
* *So, what happens when there are multiple columns that can uniquely identify an entity instance?*

#### Alternate Keys as Unique Constraints

* Suppose we were modeling an EMPLOYEE table that contains an EMPLOYEE\_ID column (a unique business identifier) + and SSNs (government-issued personal identifiers)
* *Either* column would satisfy the PK requirement of uniquely identifying a record in EMPLOYEE, but **recall that a table may only be assigned *one* PK**
* **To let database users know that *another* column (or columns) satisfies the conditions for a PK *when a PK already exists*, alternate keys (AKs) or UNIQUE constraints can also be defined**
* So, EMPLOYEE has 2 valid PK candidates: EMPLOYEE\_ID and SOCIAL\_SECURITY\_ID
* **In *OLTP* databases, the column/columns that act as the organizational business key should be made the PK**
* **In a *DW (OLAP)*, where business keys from *multiple* source systems may be loaded, a surrogate key would be used instead**
* By this convention, EMPLOYEE table should be modeled with EMPLOYEE\_ID as the PK and SOCIAL\_SECURITY\_ID as the AK (UNIQUE)
* The following example shows what that would look like:

CREATE TABLE EMPLOYEE (

EMPLOYEE\_ID varchar(10),

SOCIAL\_SECURITY\_ID number(9),

-- < rest of columns>

NAME varchar,

CONSTRAINT pk\_employee PRIMARY KEY ( EMPLOYEE\_ID ),

CONSTRAINT ak\_employee UNIQUE (SOCIAL\_SECURITY\_ID)

);

* **Unlike a PK, a table can have *as many AKs as necessary***
* **AKs give database users insight into the granularity of a table + its various columns + carry the same functional benefits as PKs**
* Like PKs, **UNIQUE constraints are only enforced in Hybrid Unistore tables + are merely orientative (not enforced but provide information) for standard ones**
* **Modeling is concerned w/ defining *entire database landscapes*, which represent the interrelated nature of the *many* business entities**
* Forming **relationships** **between entities** is a crucial detail in a database model

#### Relationships as Foreign Keys

* When business entities are **related**, their **corresponding tables must have a way to capture the details of the interaction**
* i.e., When a customer orders an item, the order details must capture who the customer is and what items they ordered
* Remember, **PKs identify a unique record in a table**
* Therefore, when 2 tables share a **relationship, the PKs of *one* table must be included *in the other* to store the details of the interaction**
* In a database, this relationship is established through a **foreign key** (**FK**) **table constraint**
* When PK columns from one table (known as the **parent**) are included in another (known as the **child**), an FK constraint can be **declared** in the child table to formalize the relationship
* The **FK constraint tells the database + its users that it is no coincidence that the 2 tables have some columns in common** but that they **share a relationship through some business context**
* An FK is represented on a diagram by a line connecting the parent table to the child
* Recall the example of PERSON and ACCOUNT from Chapter 1, where it was established that an account must be assigned to a person, + a person can open 0 or many accounts
* If it is established that PERSON\_ID uniquely identifies a PERSON (its PK), this column *must* be included in the ACCOUNT table and declared an FK constraint
* When creating the ACCOUNT table (a child table), the PK + FK constraints must be defined
* The following code snippet shows the creation of the PERSON and ACCOUNT tables:
* CREATE TABLE PERSON (

PERSON\_ID number(8,0) NOT NULL,

SOCIAL\_SERCURITY\_NUM number(9,0) NOT NULL,

DRIVERS\_LICENSE varchar(10) NOT NULL,

NAME varchar NOT NULL,

BIRTH\_DATE date NOT NULL,

CONSTRAINT PK\_1 PRIMARY KEY ( PERSON\_ID )

);

* CREATE TABLE ACCOUNT (

PERSON\_ID number(8,0) NOT NULL,

ACCOUNT\_ID varchar(12) NOT NULL,

ACCOUNT\_TYPE varchar(3) NOT NULL,

IS\_ACTIVE boolean NOT NULL,

OPEN\_DATE date NOT NULL,

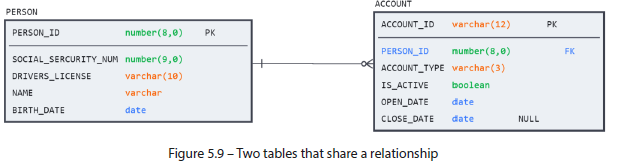
CLOSE\_DATE date,

CONSTRAINT PK\_2 PRIMARY KEY ( PERSON\_ID, ACCOUNT\_ID ),

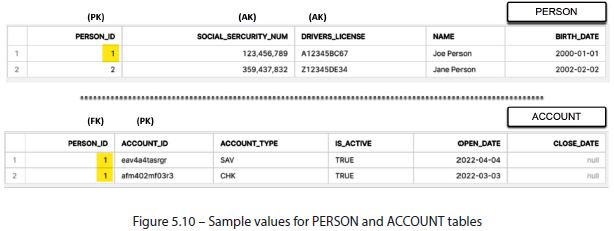
CONSTRAINT FK\_1 FOREIGN KEY ( PERSON\_ID ) REFERENCES PERSON ( PERSON\_ID )

);

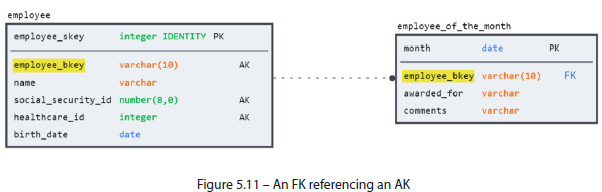
* The resulting diagram, which makes this relationship much easier for a human to perceive, is the one initially shown in Chapter 1:



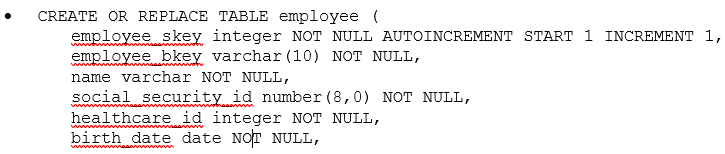
* The **PKs + FKs defined** in the preceding DDL + expressed in the figure below**ensure that the business rules** regarding account opening **are baked into the database design**
* Sample values for the resulting tables are presented in the following diagram to illustrate the data such a model might contain:

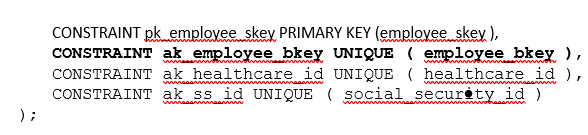


* **In databases, FK-to-PK references are the most common form of relationship, but they are not the *only* form**
* An **FK may also reference an AK**
* Often, a table might have a **surrogate key** as the PK + **various business keys declared as AKs**
* Since the surrogate key is a technical artifact *w/ no business value*, related tables will instead refer to the alternate business key, as shown in the following example:



* As you can see in the accompanying code, the **syntax for referencing an AK for an FK is identical to that used to reference a PK:**



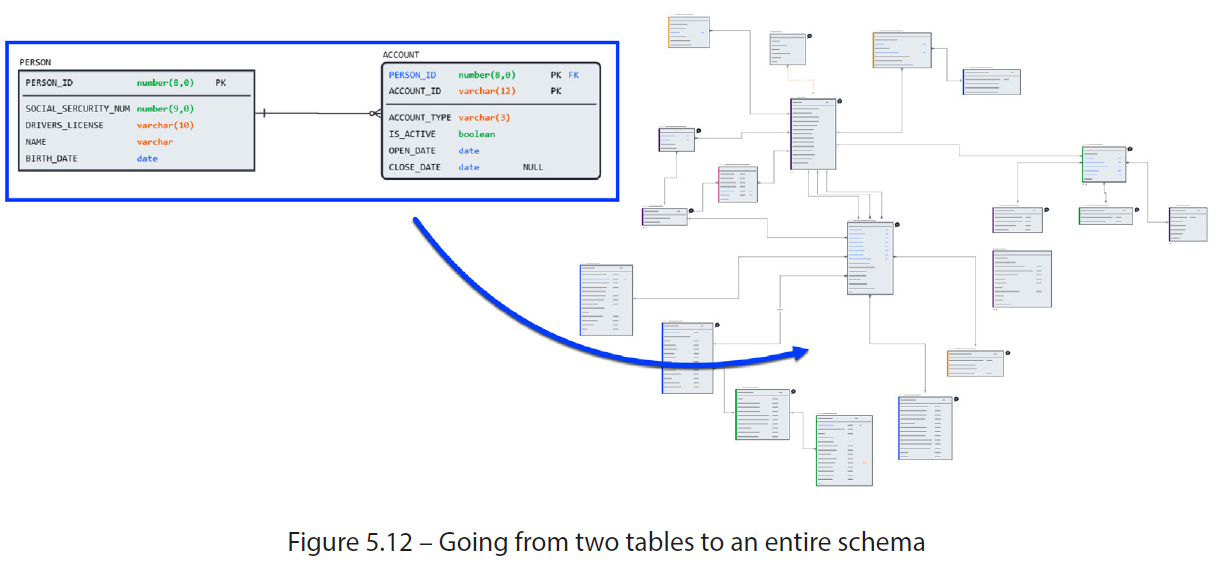


##### Benefits of a Foreign Key

* **Declaring an FK *formalizes* the functional relationship between 2 tables**
* This **metadata now becomes part of the data model** + is available for reference to anyone using the database
* Here are some of the advantages this brings:

###### Visualizing the Data Model

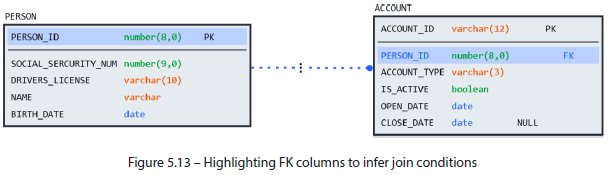
* Humans use visualizations to enhance cognition
* The previous section demonstrated how a simple diagram could communicate the same information stored in code faster + in more detail
* Now imagine going from 2 tables to an entire database schema
* W/out visual guidelines to help map out the entities + relationships, it would be challenging to make sense of all the tables and their contents without prior knowledge.
* The following figure shows what going from 2 tables to 20 would look like on a diagram:



* **Considering that data models in enterprise systems can span hundreds if not thousands of tables, the aid of a diagram becomes invaluable in navigating the business/data landscape**
* *However*, **relationships between tables are not just visual aids, as they serve a practical purpose for writing queries**

###### Informing JOINs

* **Defining an FK requires the user to specify the reference between the parent PK column(s) + those in the corresponding child table**
* This means that **the link between the 2 tables is now baked into the data model** **+ can be referenced by users wishing to JOIN the table data for analysis or transformation**
* The following shows how the FK relationship highlights the common column (PERSON\_ID) between the 2 tables:



* Now, users can leverage this metadata to write queries that retrieve PERSON and ACCOUNT data by joining the 2 tables using PERSON\_ID
* However, FKs are not just helpful to people, **+ BI tools also leverage them**
* More importantly**, Snowflake can leverage these details to avoid performing the JOIN + improve performance when it is not required** (more on this + the **RELY property** in *Ch. 12*)

###### Automating Functionality in BI Tools

* **Many BI tools use FKs to enhance usability + save time for users**
* Ex: Dashboarding + analytics tools such as Tableau and Power BI detect FKs to use them to *automatically* create JOINs between tables w/out explicit help from the users.
* Modeling tools + some SQL IDEs can generate an ERD by reading database DDL
* While every database object carries a corresponding CREATE statement, the **ERD will only be meaningful if the objects represented by the CREATE statement are *also* related through FKs**
* The next benefit we’ll discuss is only possible through enforcing the FK constraint + is therefore reserved for Hybrid Unistore tables
* However, it unlocks powerful new possibilities in the Snowflake platform that users should be aware of

###### Enforcing Referential Integrity (Hybrid Unistore ONLY)

* **As discussed, an FK relationship is NOT purely orientative**, but it **formalizes business rules through database conventions**
* Recall that it was stipulated that an ACCOUNT cannot be opened w/out a PERSON\_ID
* So, when a new account is created, *how can a database ensure a PERSON\_ID is provided +, more importantly, that it corresponds to a valid record in the PERSON table?* 🡪 **By enforcing the FK constraint**
* **When a database *enforces* an FK constraint, it checks to ensure the unique identifiers used in the child table *exist* in the parent** (known as **referential integrity**)
* **Referential integrity** **ensures that the FK values in the child table correspond to valid records in the parent table**
* Just as it would not make sense to open an account for a person who does not exist in the database, **referential integrity checks ensure that DML operations do not result in erroneous + anomalous data**
* Just like enforcing a PK constraint, **FK enforcement relies on existence of an index to perform the referential integrity check quickly, + is therefore only possible in Hybrid Unistore tables (unless manually performed)**
* ***Whether enforced or not*, there are many compelling reasons to use FKs**
* The last constraint to cover is the only one enforced on *ALL* Snowflake tables, not only Hybrid Unistore

#### Mandatory Columns as NOT NULL Constraints

* **When defining attributes for an entity, the question of which ones are mandatory + which are optional inevitably arises**
* **As w/ most modeling decisions, the answer depends on the business context more than any technical database property**
* The same attribute (Ex: email address for CUSTOMER, may be mandatory for an online store but optional for a brick-and-mortar retailer)
* In the latter case, not having an email address means missing sales announcements, while in the former, it may mean being unable to access the website.
* When moving from a conceptual model to a physical Snowflake design, **mandatory columns can be defined through the NOT NULL constraint**
* The NOT NULL constraint is **declared inline next to the corresponding column + does not need to be given a name**
* Due to this, it is NOT possible to declare NOT NULL constraints out of line
* Format for adding a NOT NULL constraint: <col1\_name> <col1\_type> [ NOT NULL ]
* Previous examples, such as the ACCOUNT table, have demonstrated instances of NOT NULL constraints
* Below, we can see that CLOSE\_DATE is the only column in the ACCOUNT table that is not mandatory and is allowed to have a NULL value:
* CREATE TABLE ACCOUNT (

PERSON\_ID number(8,0) NOT NULL,

ACCOUNT\_ID varchar(12) NOT NULL,

ACCOUNT\_TYPE varchar(3) NOT NULL,

IS\_ACTIVE boolean NOT NULL,

OPEN\_DATE date NOT NULL,

CLOSE\_DATE date,

CONSTRAINT PK\_2 PRIMARY KEY ( PERSON\_ID, ACCOUNT\_ID ),

CONSTRAINT FK\_1 FOREIGN KEY ( PERSON\_ID ) REFERENCES PERSON ( PERSON\_ID )

);

* **Unlike the other constraints, enforcing NOT NULL does NOT require lookups or references to other tables, + it can be done *directly* on the individual rows by checking for values in the designated columns**
* For this reason, **NOT NULL is enforced for all Snowflake tables + does not incur any performance overhead during DML operations**

#### Summary

* This chapter discussed how to transition from *logical* modeling concepts to *physical* Snowflake objects
* During this process, we learned how **Snowflake handles tables of near-infinite size by breaking them down into manageable micro-partitions** + how **these partitions can be clustered to optimize query + DML performance**
* After that, we learned how to define **attributes** by understanding **Snowflake’s data types + their properties**
* Snowflake offers a variety of functions to make working w/ data types easier + more performant, to say nothing of the powerful options it offers for semi-structured data
* Before diving into individual **constraint** types, we understood what database constraints *are* + how Snowflake organizes + enforces them depending on table type where they’re applied
* We saw why **unique identifiers** are vital for defining tables + how Snowflake manages this through the **PK constraint**
* **PKs help make life easier for database users by helping identify the minimum level of detail in a table + ensuring accurate searches + joins**
* We also saw that sometimes, when multiple candidates exist for a PK, **AKs** **can be defined through UNIQUE constraints + offer the same benefits**
* Then, we saw how **relationships** are formalized in a physical database by defining **FK constraints**
* **FKs make visualizing + navigating complex data landscapes possible by capturing the functional relationships between tables** (information that human users of the database + BI tools frequently leverage)
* Finally, we learned how to declare **mandatory columns** + enforce the rule using the NOT NULL constraint to ensure data quality.
* This chapter + others have frequently relied on relational diagrams to clarify examples + explain abstract concepts
* However, there has been no formal explanation of the modeling notation or an overview of the various elements in the visual modeling toolkit
* Now that we understand the core elements of a data model at a logical + physical level, next chapter will focus on modeling notation for designing or visualizing data models at any scale