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

## Part 3: Solving Real-World Problems with Transformational Modeling

* This part focuses on **advanced transformational modeling techniques**, covering the theory and practice of putting them into action in Snowflake
* First, we explore **slowly changing dimensions (SCDs)**, which are **critical for maintaining historical data accuracy + ensuring consistency across different versions of data**
* Then, we learn how to **model facts for rapid analysis**, including best practices for optimizing query performance
* We then explore **semi-structured data formats + hierarchies**, which are becoming increasingly prevalent + which Snowflake is so adept at handling
* This part also covers the basics of **Data Vault methodology for building agile, scalable, + auditable architectures**, as well as **Data Mesh, an increasingly popular approach to managing data domains at scale across large organizations**
* Finally, you will experiment hands-on w/ fun + effective SQL recipes that couldn’t be included in the rest of the chapters

### Chapter 12: Putting Transformational Models into Practice

* In the preceding chapters, we went from **gathering requirements w/ business teams to creating + deploying a physical data model to Snowflake**, which aligns w/ our organization’s operations
* Now it is time to **leverage Snowflake’s powerful query processing engine + its full-featured library of functions + data manipulation features to creatively *transform* data to answer business questions**
* While **physical modeling *creates* objects by defining the structure**, **transformational modeling uses logic** **(*selecting* existing data + creating a *new* object from the query result)**
* However, **query processing in Snowflake comes at the cost of compute credits**
* This chapter will cover best practices for writing efficient queries in Snowflake to help control costs + increase performance
* As we build transformational models, we will also learn how to **monitor their performance + detect issues using the Snowflake query profile**
* Most importantly, we will see how the **structural foundation of relational modeling helps us create transformations faster + drives down warehouse costs through join elimination + other features**
* Main topics:
* **Separating logic from objects** to improve the database design
* Using the **physical model to drive transformational design**
* Performance gains through **join elimination**
* **Joins and set operators** available in Snowflake
* **Monitoring Snowflake queries** to identify common issues
* **Building a transformational model** from a business requirement

#### Technical Requirements for Local Snowflake Work

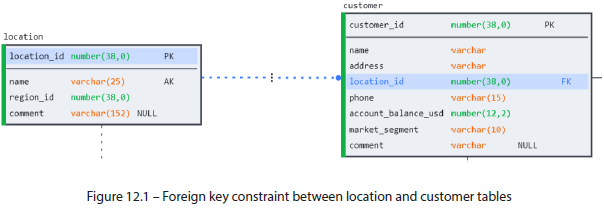
* Examples + exercises in this chapter require data to execute
* The script to load the model from the previous chapter w/ sample data:
* *create\_physical\_model\_w\_data.sql*
* Included at <https://github.com/PacktPublishing/Data-Modeling-with-Snowflake/tree/main/ch12>
* The complete end-of-chapter transformational model exercise is *create\_loyalty\_customer.sql*, although readers are encouraged to **recreate it from scratch**

#### Separating the Model from the Object

* The **ability to instantly scale up (increase size of) warehouses gives Snowflake users easy control over query performance + duration**
* However, **increased warehouse size comes at the price of compute credits**
* **Even keeping the warehouse size constant, changes in data volume + query patterns can cause performant + cost-effective data sources to degrade**
* **To mitigate performance degradation, a view may need to be materialized as a table, or a table may need to become a materialized view**
* However, **even when converting from a view to a table, the transformational logic stays constant**
* While traditional modeling advice advocates differentiating views and other database objects through suffixes (like *CUSTOMER\_V*), **Snowflake users are encouraged to avoid such conventions**
* **Orienting object names to their *contents*** (e.g., CUSTOMER, DIM\_DATE) **rather than their *object type* allows modelers to easily pivot between them w/out breaking downstream dependencies**

#### Shaping Transformations Through Relationships

* The exercises in previous chapters devoted much attention to **relational constraints** + how they help **ensure data integrity, consistency, + quality while making data management easier**
* *Exactly how much easier* will be the recurring theme of this chapter
* Even simple business questions, such as “What is the customer and location name for the customer with identifier 775699?” require us to refer to the relational model to find the tables in which these attributes are stored + the FK columns that can be used to join them



* Using the information from the physical model, we can construct a query to obtain customer and location names for customer 775699 as follows:
* SELECT c.name AS customer\_name, l.name AS location\_name

FROM customer c

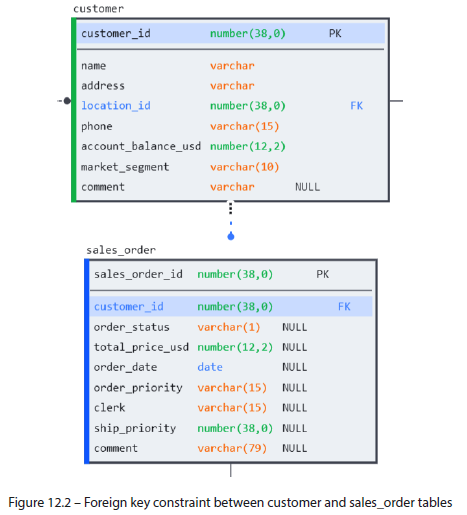
JOIN location l ON c.location\_id = l.location\_id

WHERE customer\_id = 775699;

* And we obtain the result we are interested in:



* **NOTE: Using descriptive aliases**
* The examples in this book use a **single-letter alias style** intended to maximize readability on a printed page but is **NOT recommended as a general practice**
* **Paying attention to constraints also allows users to avoid join explosions, which are queries that return more rows than expected**
* **Join explosions** **distort query results by returning duplicate values + multiplying the amount values returned by the query**
* However, join explosions are **even more insidious at scale b/c they can multiply the number of records returned by a query by several orders of magnitude, severely impacting performance and needlessly burning through warehouse credits**
* Blindly trying to obtain a customer name from the SALES\_ORDER table can have unexpected consequences
* As the relational diagram indicates, a customer may place *multiple* orders
* Therefore, CUSTOMER\_ID is *NOT* unique in the SALES\_ORDER table



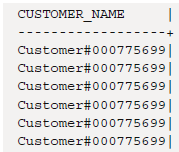
* Attempting to join CUSTOMER to SALES\_ORDER to obtain customer name would not be as straightforward as in the previous example, although the resulting query would look similar:
* SELECT c.name AS customer\_name

FROM sales\_order AS so

JOIN customer AS c ON c.customer\_id = so.customer\_id

WHERE so.customer\_id = 775699;

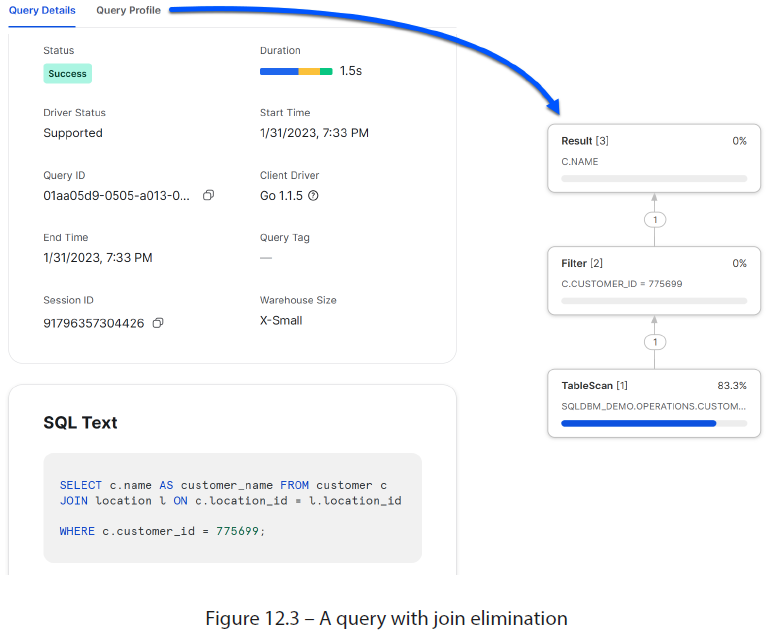
* However, the query returns *multiple* records in place of a single name (1 per order made by customer 775699):



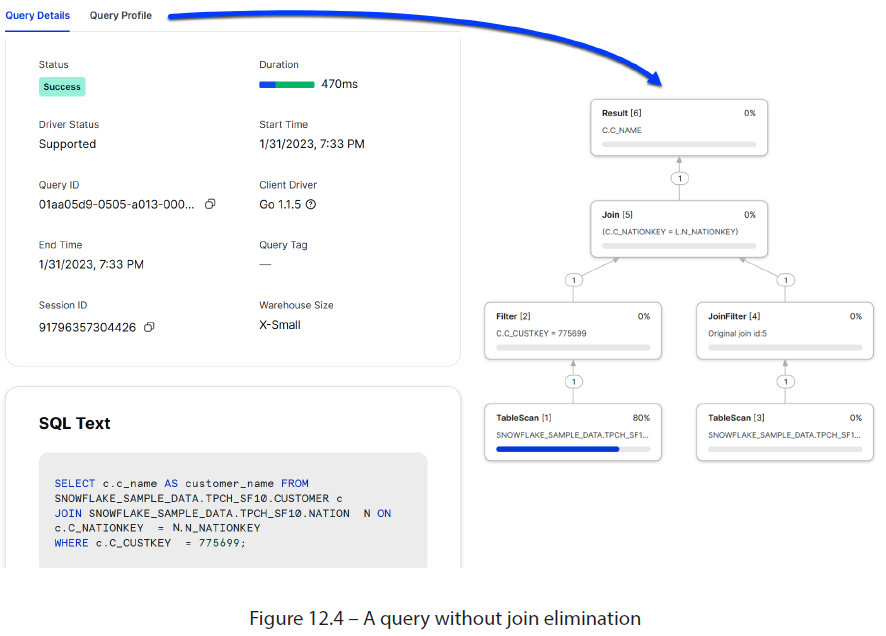
* The **constraints in a relational model accelerate analytics by informing users how to link data from multiple data sources to answer business questions + avoid unpleasant surprises**
* But **constraints can play an even more important role in query performance by *eliminating* certain joins altogether**

#### Join Elimination Using Constraints

* In the previous chapter, we set the **RELY property** on our **constraints** to pave the way for the performance gains we will now explore
* **Snowflake uses the RELY property to perform join elimination, which means avoiding unnecessary or redundant joins in queries**
* **Depending on table size, joins can be memory-intensive, so avoiding them when possible can significantly improve performance + save compute credits**
* **Even if a join is specified in a query, but *no columns from the joined table are selected as part of the result*, the RELY property will tell the Snowflake query engine to *avoid* performing the join**
* If we modify the previous query (joining CUSTOMER + LOCATION tables) but *only* request information from CUSTOMER, then RELY will help us avoid the unnecessary join operation
* A look at the **query profile** (below) confirms that only the CUSTOMER table was referenced as part of the query operation, saving us an extra table scan + a join



* Here is the same query, pointing at the equivalent tables in the SNOWFLAKE\_SAMPLE\_DATA database which does *NOT* use the RELY property:



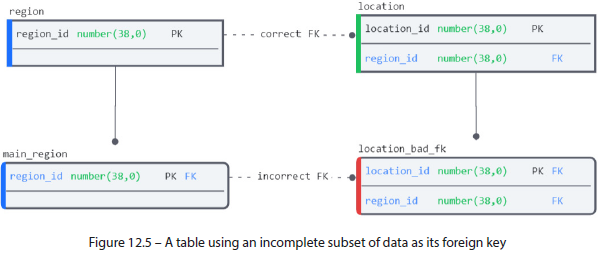
* Notice that *w/out* RELY, Snowflake scans *both* the CUSTOMER and NATION tables
* **While join elimination can significantly improve query performance in some cases, there are times when it may present unintended consequences**

##### When to Use RELY for Join Elimination

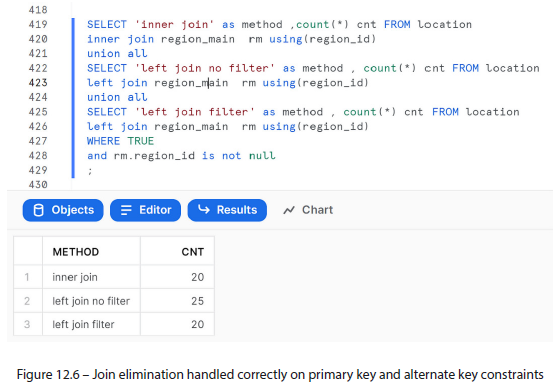
* **Star and snowflake schemas** (see Chapter 17) **are popular modeling choices + offer great opportunities for improving query performance through join elimination**
* In these scenarios, a **central fact table** is **joined to several dimension tables**, which **add descriptive attributes**
* **Enabling the RELY property on the constraints between the fact table + dimensions allows us to encapsulate all the join conditions + business logic in a multidimensional view + make it available to all users, regardless of the dimensions each is interested in**
* **When a query references the view, Snowflake will *only* perform the join on the dimensions *explicitly* requested**

##### When to Be Careful Using RELY

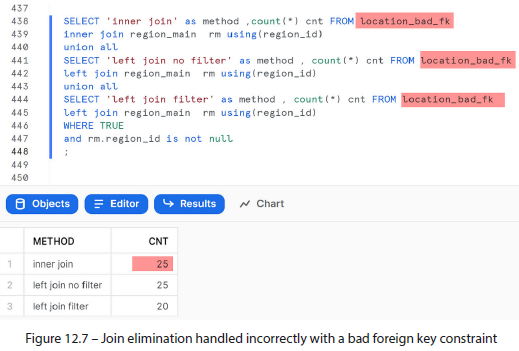
* **Not *all* joins that reference columns from only *one* table are redundant**
* **Inner joins return matching records between 2 tables + are often used as a filter instead of returning values from 2 tables**
* Often, master data exists as a subset of itself (e.g., VIP\_CUSTOMERS as a subset of CUSTOMERS)
* In such cases, the constraints, *if set*, are **semi-reliable** **(the PK contains valid + unique values, *but not all of them*)**
* Happily, the **Snowflake optimizer is smart enough to detect + handle such scenarios when the RELY property is set on the PK and AK constraints**
* In the following example, a MAIN\_REGION table is created as a subset of REGION
* A duly titled LOCATION\_BAD\_FK table references MAIN\_REGION as its FK as seen in the following image:



* ***W/out* an *explicit* FK constraint, join elimination is handled correctly for redundant joins and non-redundant INNER filter joins**



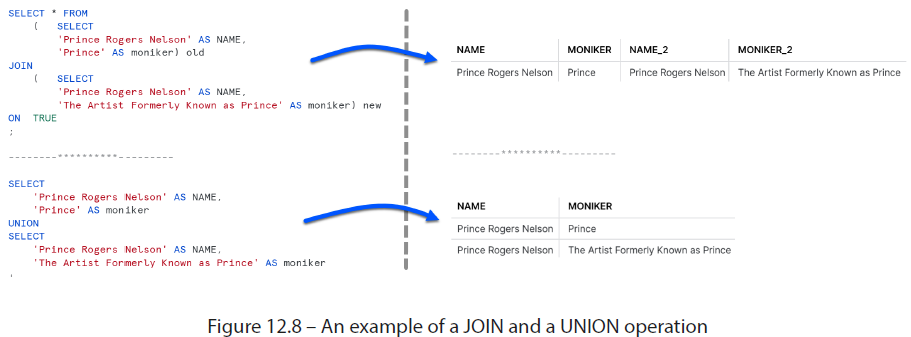
* It’s *only when we* ***create an incorrect FK relationship*** between the LOCATION\_BAD\_ FK and REGION\_MAIN that we can ***trick* the optimizer into giving us an incorrect result**
* To perform the experiment yourself, refer to the accompanying code in *join\_elimination.sql* in the repository for this chapter



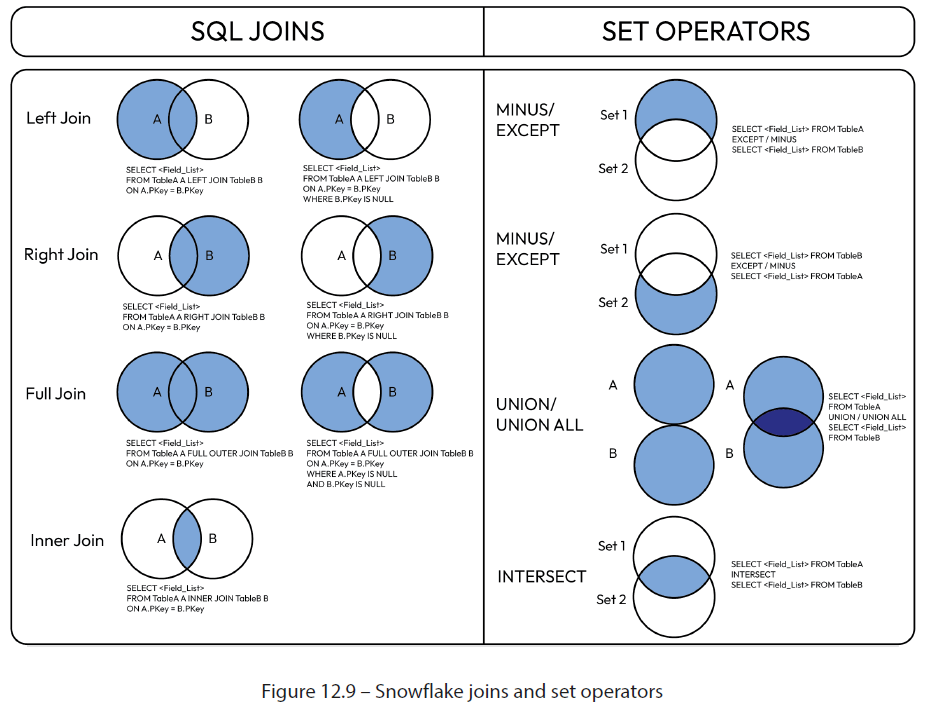
* **The takeaway is that the RELY property is an effortless + *nearly* risk-free way to optimize query performance on Snowflake views**
* For more on join elimination + using it in multidimensional queries, refer to the extras section of the repository at <https://github.com/PacktPublishing/Data-Modeling-with-Snowflake/tree/main/extras/04_join_elimination>
* Now that we have begun to experiment w/ together data from multiple tables, it’d be a good time to review the complete list of joins + set operators Snowflake offers to ensure everyone is familiar w/ their usage

#### Joins and Set Operators

* In a database, there are **2 ways to bring sources of data together**: **joins** + **set operators**
* A **join combines *rows* from 2+ tables based on related columns**
* A **set operator** (such as UNION) **combines the results of multiple *SELECT statements***
* But you can *also* use JOIN on multiple SELECT statements
* An easier way to think about it is **joins combine data *horizontally* (across related rows)** + **set operators work *vertically***
* The following example shows how identical records would look in a JOIN or UNION result



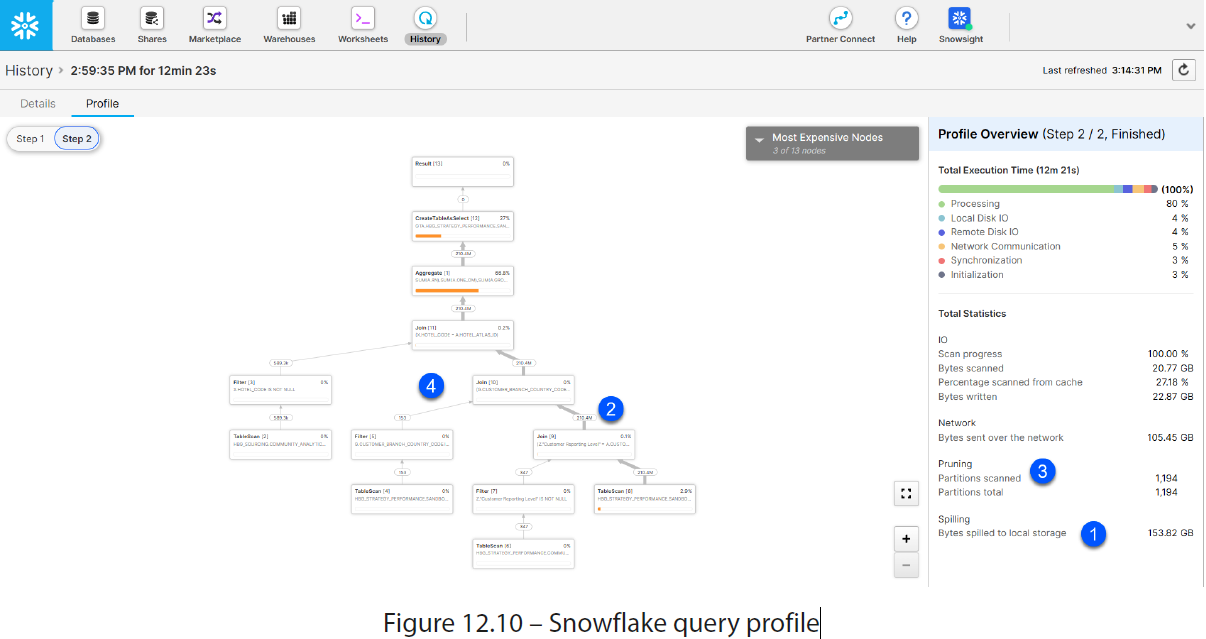
* JOIN and UNION are just *some* of the tools at the developer’s disposal
* Let’s review all the join + set operators Snowflake provides + briefly cover some of their use cases.
* **JOINs**
* **INNER JOIN – Selects records that have matching values in *both* tables**
* By doing so, this join **also acts as a filter**
* **Unless data integrity is *guaranteed*, avoid using the inner join to add related attributes in a query**
* Ex: **Selecting from a fact table + joining a related dimension to obtain descriptive attributes causes certain fact records to disappear if no matching dimension records exist**
* The INNER keyword is optional but encouraged for readability.
* **LEFT JOIN – Returns all records from the left table + the *matched* records from the right table**, thereby avoiding the issue described with INNER JOIN
* **RIGHT JOIN – Returns all records from the right table + *matched* records from the left table**, thereby avoiding the issue described with INNER JOIN
* **FULL JOIN – Returns all records when there is a match in *either* the left or right table**
* Helpful for data reconciliation + missing data analysis
* **Set operators combine the result set of 2+ SELECT statements *as long as the following applies***:
* Every SELECT statement has the **same number of columns**
* The columns all have **similar data types**
* The **columns are in the same order**
* **Overview of set operators**:
* **UNION ALL** – **Combines the result set of 2+ SELECT statements *w/out performing duplicate elimination***
* If a record exists in multiple sets, it will be included multiple times.
* **UNION – Same as UNION ALL but *does* performs duplicate elimination (using DISTINCT)**
* **Unless duplicate elimination is *required*, it is recommended to use UNION ALL to avoid the performance penalty of sorting DISTINCT records**
* **MINUS/EXCEPT** – **Removes rows from one query’s result set that appear in another query’s result set, w/ duplicate elimination**
* Often used in data validation + auditing (such as tracking changes in data over time)
* **INTERSECT – Returns rows from one query’s result set that *also* appear in another query’s result set, w/ duplicate elimination**
* Used in data reconciliation to ensure consistent data across all sources



* Before putting these into practice, let’s review some of the performance + query-writing best practices that will help make queries easier to read + maintain

#### Performance Considerations and Monitoring

* B/c the **Snowflake services layer takes care of query tuning + optimization, writing efficient queries + following best practices should be the developer’s only concern**
* To do this, **Snowflake provides a visual query profile** to **break down query execution into constituent sub-steps + help identify potential issues**
* A sample query profile can be seen in the following screenshot:



* A good way to familiarize yourself with the query profile screen is by reviewing the kinds of query issues that it is designed to identify
* The numbers in the query profile correspond to the topics listed in the following section

##### a) Common Query Problems

* Most day-to-day query issues can be identified by paying attention to the information displayed in the query profile + following the query best practices described here:
* **Queuing** – **When Snowflake determines that further queries in a DW would lead to performance degradation, the query is queued + it waits for the necessary system resources to become available**
* While a query queues, no profile information is visible
* To **mitigate queuing issues**, users can **scale out** **(add warehouses)** or **adjust the MAX\_CONCURRENCY\_LEVEL warehouse parameter to accommodate more processes** (w/ less resource allocation for each)
* <https://docs.snowflake.com/en/sql-reference/parameters.html#max-concurrency-level>
* **Spilling** – **When a query cannot fit into warehouse memory, it begins spilling (*writing*) data to the local disk + then to network remote storage**
* **Although it causes noticeable performance degradation, spilling is doubly pernicious because users incur credit costs for warehouse memory usage *while getting the performance of writing to disk***
* If spilling occurs, **review the query for potential issues + increase the warehouse size to give the operation more memory to work w/**
* Another strategy is to **split the query into multiple steps or process the data in smaller segments** (explained in the later point on “Inefficient join order”)
* **Join explosion** – **When a join results in a Cartesian product (also known as a cross join), reviewing the row counts entering + exiting a join node will reveal the error**
* **The number of records resulting from a join between 2 tables should be equal to or less than (in the case of INNER) the greater of the 2**
* While record multiplication is desired in *some* cases, it *usually* results from an incorrect join condition
* If the join condition matches the relational model, the tables should be checked for duplicate records
* **Insufficient pruning** – As described in Chapter 5*,* **Snowflake clusters table data in micro-partitions and maintains metadata + statistics for data stored in each partition**
* **Snowflake must read *every* partition if a query selects *all* records from a table**
* However, **if a query filter is applied, it should ideally prune (avoid scanning) some partitions + improve performance**
* **Review the number of partitions scanned vs. the total partitions in every table scan node to determine whether the pruning result is adequate**
* To improve pruning, the **general recommendation is to consider whether clustering can be improved**
* Review examples in Chapter 5for more details on how to achieve better query pruning
* **Inefficient join order – Snowflake does NOT support query hints, + the SQL query optimizer generally chooses the optimal join order for queries**
* However, if, on the rare occasion, you notice that joins are not being performed in the order of most to least restrictive, you can **break the query up + enforce the desired order yourself**
* Ex: Consider the following query:
* SELECT a, b

FROM x --large fact table

INNER JOIN y on x.c=y.c --more restrictive join

INNER JOIN z on x.d=z.d; --less restrictive join

* Reviewing the query profile, you see that tables Y and Z are being joined first, + the result is used to join table X
* Knowing the data, you determine there is a better way + rewrite the query, like so:
* CREATE TEMPORARY TABLE temp\_xy AS
* SELECT a, b, d

FROM x --large fact table

INNER JOIN y on x.c=y.c; --more restrictive join

* SELECT a,b FROM temp\_xy

INNER JOIN z on temp\_xy.d=z.d; --less restrictive join

* While the **temp table will only persist for the duration of the session, it still requires additional overhead to create**
* **Snowflake provides a more efficient alternative that takes advantage of the results cache**
* This is **achieved w/ the Snowflake functions LAST\_QUERY\_ID() and RESULT\_SCAN, and then casting to the table**
* The previous example can be **rewritten to use the query cache** by making the following adjustments:
* SELECT a, b, d

FROM x --large fact table

INNER JOIN y on x.c=y.c; --more restrictive join

* SELECT a,b FROM TABLE(RESULT\_SCAN(LAST\_QUERY\_ID())) rs\_xy

INNER JOIN z on rs\_xy.d=z.d; --less restrictive join

* **When increasing warehouse size is impossible, the preceding technique *can* fita query into available warehouse resources w/out spilling**
* In addition to the discussed methods, there are other suggestions to ensure clean, readable, + efficient queries

##### b) Additional Query Considerations

* Besides the performance tips mentioned in the previous section, the following guidelines should also be considered to make queries more readable and easier to test + maintain:
* **Use table aliases**, which are a **good way to make queries more readable + avoid naming collisions when joining tables**
* Can also make referencing columns from specific tables in the query easier
* When choosing an alias, use the Goldilocks principle: not too long to type out but not too short that you forget what they mean, *just right*
* As w/ table naming advice in Ch. 10, the guiding principle is to **pick a style + be consistent**
* **Use common table expressions (CTEs)**, which can **simplify reading + debugging complex queries by creating named subqueries using a WITH clause**
* Creates **modular code blocks** that can be separately debugged
* As with all advice, **exercise constraint + avoid excessive modularization, which can be as challenging to read** as the spaghetti code it is meant to replace
* More info on CTEs: <https://docs.snowflake.com/en/user-guide/queries-cte.html>
* **Limit the rows returned**
* When ***testing*** queries, limits can reduce the number of records that need to be processed
* The traditional way to do this is using the **LIMIT keyword, but this has 2 disadvantages**:
* **1) LIMIT works on the *final* query result (*after* all joins + calculations are performed)**
* **2) It returns rows in the order they were saved to Snowflake**
* **If new data is written incrementally, LIMIT returns the *same* N (older) records *every* time**
* A **better way is to use the SAMPLE function, which picks records randomly from the entire table, giving a much better representation of the data**
* This can also be customized to return different kinds of samples
* More information on how to use + tune the SAMPLE command:
* <https://docs.snowflake.com/en/sql-reference/constructs/sample.html>
* An example of limiting query results is shown here:
* --Good

SELECT \* FROM customer LIMIT 10;

* --Better

SELECT \* FROM customer SAMPLE (10 ROWS);

* **UNION *without* ALL**
* **Unless the added DISTINCT step is *required* to de-duplicate the result set, use UNION ALL**
* **Use EXPLAIN PLAN**
* Generating a **query plan requires users to *run* the query in question**
* However, you **can check the execution plan of a query to see how Snowflake will execute it + check whether any issues need to be addressed**
* This option **will *not* include statistics on rows returned or spilling** but ***can* be useful in validating the join order**
* The command to do this is **SYSTEM$EXPLAIN\_PLAN\_JSON**, + more info, including how to convert the output to a table or text, can be found in the documentation:
* <https://docs.snowflake.com/en/sql-reference/functions/explain_json.html>
* **Continuous monitoring**
* **Even performant queries can degrade over time as data volumes grow or data quality deteriorates**
* **Consistent monitoring is essential to detect performance drift *before* it impacts warehouse costs**
* The **complete guide to using Snowflake’s query profile can be found in the documentation**:
* <https://docs.snowflake.com/en/user-guide/ui-query-profile.html>

#### Putting Transformational Logic into Practice

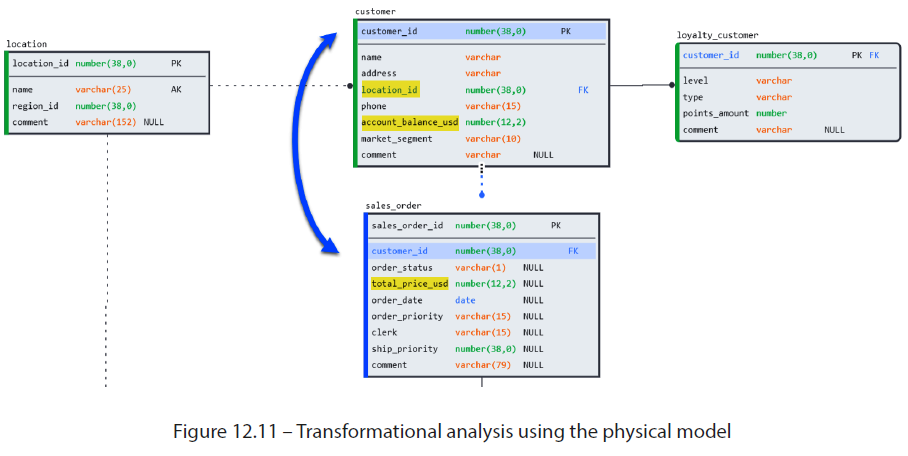
* Last chapter we **deployed** our sample physical model
* Here, we will **populate it** w/ data + **create a transformational model to satisfy a business requirement** using the techniques + best practices covered in the preceding sections
* **Transformational modeling requires data**
* The script to populate our operational schema w/ sample data can be found in the Git repository for this chapter
* Run the script titled *create\_physical\_model\_w\_data.sql* to recreate the physical model + load it w/ data from the SNOWFLAKE\_SAMPLE\_DATA database if you have not already done so
* After running the script, all transactional tables in the schema will have been populated w/ data
* However, the **LOYALTY\_CUSTOMER table is *not* transactional**, + needs to be *created* through a **transformational model**
* Just as the data model in previous chapters took shape after getting to know the workings of our company by communicating w/ the business teams, **transformational modeling works to satisfy business questions**

##### Gathering the Business Requirements

* Knowing the structure of LOYALTY\_CUSTOMER from the physical model tells us *little* about what constitutes a loyal customer
* **By sitting down w/ domain experts** from the marketing team, we **can glean the business logic**
* The marketing team confirms there are 2 kinds of loyal customers: top 400 based on order volume and 5 early supporters
* They summarize the business logic in a formal request to define a table that recalculates the results every week
* The requirements are as follows:
* The 5 early supporters, identified by customer\_id, are 349642, 896215, 350965, 404707, 509986
* These are of type “early supporter”
* The top 400 customers are calculated based on the sum of total\_price\_usd from sales\_order
* These are of type “top 400”
* The loyalty level is calculated as follows:
* Gold: early supporters and top 20 customers by total\_price\_usd
* Silver: top 21-100 customers by total\_price\_usd
* Bronze: top 101-400 customers by total\_price\_usd
* Customers in location\_id 22 are *not* eligible for the program
* Customers with a negative account balance are *not* eligible
* The table should have a points\_amount column, which the marketing team will maintain
* It appears that **various tables need to be joined to complete this calculation**
* **Before attempting to *write* the query, validate the join conditions in the physical model**

##### Reviewing the Relational Model

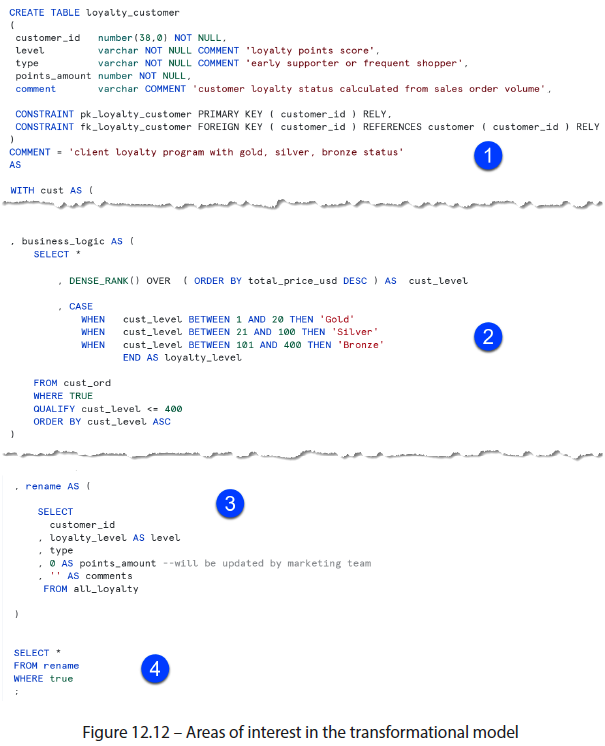
* **Using the physical model, we can analyze business requirements w/out relying on other team members**
* The information we need to get started can be seen in the following figure:



* Reviewing the diagram, it seems that the information needed to determine a loyal customer is in a single table: SALES\_ORDER
* However, we need to join to CUSTOMER to exclude those who are ineligible, + the way to do so is through CUSTOMER\_ID
* **Having confirmed the details needed to create the transformational logic, can now write the necessary SQL**

##### Building the Transformational Model

* The SQL required to write the transformation in this example is straightforward + is included in this chapter’s Git repository (*create\_loyalty\_customer.sql*) for reference
* Instead of focusing on its *exact* implementation, it’d be more beneficial to highlight the sections that relate to the principles mentioned in this chapter



* 1) Although we are creating this table through transformational modeling, **declaring relational constraints (PKs + FKs) + updating the model remains equally important for usability**
* This example contains the **constraints + the transformational logic in the same statement**
* Alternatively, constraints can be declared after the table has been created using ALTER
* 2) **Breaking the logic into modular chunks using CTEs allows for cleaner code that’s easier to read + debug**
* A developer could test the result of business\_logic CTE by simply calling it from the final SELECT statement
* 3) **Column renaming should be reserved until the end of the script**
* **Allows developers to refer to source columns throughout the code + find all renaming in one standard place**
* 4) **Modularizing sources, logic, + renaming simplify the final SELECT** **statement**, allowing easy filtering + debugging
* After delivering the finished model, the marketing team decides (as business teams often do) that a weekly refresh is insufficient, + that the **data needs to be updated in real time**
* Without changing a single line of logic, we **can change LOYALTY\_CUSTOMER to be a view** instead of a table by changing the first line of the CREATE statement:
* CREATE **VIEW** loyalty\_customer AS <existing logic>
* By **keeping the object name the same, downstream processes can continue to reference it w/out requiring updates**

#### Summary

* While a ***physical* model reflects the company’s *business* model, transformational models typically focus on *addressing analytical business questions***
* **Transformational models depend on *existing* data to perform functions and, since data grows and changes over time, the object type used to represent the underlying logic may also change**
* By **keeping object names constant, users can make adjustments + pivot between different database objects w/out breaking downstream processes**
* The **relationships established in the physical models can inform transformational designs as well as improving performance through join elimination by using the RELY property**
* **Can track performance to spot potential query issues using the query profile in the Snowflake UI**
* Valuable tool for identifying performance issues such as exploding joins + inefficient pruning
* **While transformational modeling is performed *on top* of the physical model, the two should always be kept in sync to provide database users w/ a complete overview of the system landscape**
* **Keeping the relational model updated ensures that everyone in the organization can locate the data assets they need + identify where they fit in the overall model**