# 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

### Chapter 14: Modeling Facts for Rapid Analysis

* **Fact tables are used to store the *quantitative* measurements of business operations or events** such as sales, employee headcounts, or website traffic
* Because they contain the **official record of business transactions**, fact tables are a **prime target for operational analytics**
* Fact tables **aggregate metrics** such as sales totals + active users, **as well as historical trends** (deltas), such as the margin impact of daily returns or same-day bookings before cancelations
* B/c business needs vary by industry + sector, **various fact table models exist** to fit these different demands
* Facts such as product sales + returns are erratic, while others, such as manufacturing or fulfillment, follow a predictable pattern
* The **fact tables** supporting such processes **must anticipate not only the nature of the data they aim to capture but also the organization’s analytical needs to allow for efficient reporting**
* This chapter will cover the various fact table models + the industry use cases to which they are best suited
* Most importantly, we will look at several Snowflake-driven patterns that can tackle some of the most common design challenges associated w/ fact tables in a cost-effective + low-maintenance manner
* Main topics:
* Getting to know various fact table types
* Understanding the various types of measures or facts
* Reverse balance, the world’s most versatile transactional fact table
* Recovering lost (physically deleted) facts
* Working with facts over points and ranges of time
* Snowflake recipes for building, maintaining, and querying each of these fact tables efficiently

#### Technical Requirements for Local Snowflake Work

* The scripts used to instantiate + load the examples in this chapter are available in the following GitHub repo: <https://github.com/PacktPublishing/Data-Modeling-with-Snowflake/tree/main/ch14>
* While the key section of each script will be explained in the latter half of this chapter, please refer to the repo for the complete code used to maintain + schedule the loading of the objects discussed

#### Fact Table Types

* By capturing the **daily operational transactions** of an organization, **fact tables tend to contain large amounts of records that are constantly growing**
* By analyzing the data in fact tables, analysts + business users glean insights into business performance and identify trends + patterns
* **Considering these demands, fact tables must be designed in such a way that balances data loading efficiency w/ analytical needs + query patterns**
* After nearly 20 years + 3 editions, the **definitive guide to designing fact tables remains The Data Warehouse Toolkit** where authors Ralph Kimball + Margy Ross expertly cover the fundamentals of dimensional modeling, fact table design, + related industry case studies
* We focus on what The Data Warehouse Toolkitdoes *NOT* cover: ***database-specific* transformations for managing + maintaining fact tables *in Snowflake* in cost-efficient + analytically versatile ways**
* However, before jumping into Snowflake-specific recipes, an overview of fact table types + their various metrics is required
* Nearly every organization has some common analytical needs, such as obtaining operational totals through aggregations across business + temporal dimensions
* However, **differences in dimension types**, such as time, geography, or product category, **will drive design decisions that facilitate user-defined requirements** such as drilling down or across or comparing historical records
* To meet these needs, 5 basic fact table types have been identified:

##### 1) Transaction Fact Tables

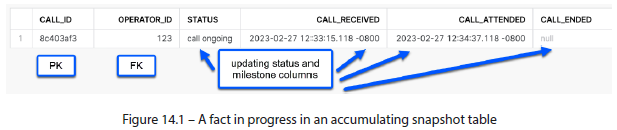
* **Transaction fact tables** store information about **individual transactions**, such as sales or bookings
* They typically have the **most detailed level of information** + **the largest number of rows**
* Most organizations will use this type of fact table as the **baseline for granular analysis, summarized aggregates, + generating the other types of fact tables in this list**

##### 2) Snapshot Fact Tables

* **Snapshot fact tables** store information about a **specific point in time**, such as inventory levels or customer metrics **at the end of a given period**
* They are **usually updated on a regular basis**, such as daily, weekly, or monthly
* They **do *not* contain granular information** such as individual order details + **instead** **provide predictably occurring slices of operational data**, which **can** **easily be compared due to their identical granularity**
* **Snowflake’s materialized views** should be considered a maintenance-free way of maintaining snapshot tables using a transaction fact table as a source

##### 3) Accumulating Snapshot Fact Tables

* **Accumulating snapshot fact tables** are similar to snapshot fact tables, but **track the progress of a process w/ predictable steps over time**
* They typically have a **fixed set of milestones** that **reflect the stages of a fixed business process**, say, the customer journey through a call center from receipt to resolution
* Unlike other fact table types, in which records are *only* inserted and *not* updated (except to correct errors), **accumulating fact tables contain milestone, status, + corresponding change-tracking columns, which receive updates as the underlying record changes**



##### 4) Fact-less Fact Tables

* **Fact-less fact tables do NOT contain *any* measures but *are* used to record events or transactions *between dimensions***, such as student enrollments in college courses

##### 5) Consolidated Fact Tables

* **Consolidated fact tables** occur when **separate but related facts share the same grain + need to be analyzed together**
* Sales + sales forecasts are classic examples of consolidated facts that are analyzed side by side
* The **choice of which type of fact table to use depends on the nature of the data being analyzed, the specific business requirements, and the performance + storage constraints of the DW platform**
* *Just like SCD types*, **a DW may combine the different fact table types to support various analytical needs**
* Having seen the various types of fact table structures, we should familiarize ourselves w/ the **different categories of measures they can contain**
* Understanding the kinds of measures will play an important role in creating the transformational logic required for maintaining them

#### Fact Table Measures

* **Facts** are **numerical measures associated w/ a business transaction** that fall into **3 basic categories**:

##### 1) Additive Facts

* **Additive facts** are measures that **can be summed across *any* dimension**
* These are the **most common type of fact** in a DW, allowing for a wide variety of analytical calculations + insights
* They **can be aggregated across any combo of dimensions**, such as time, geography, or product
* Examples include sales revenue, profit, + quantity sold

##### 2) Semi-Additive Facts

* **Semi-additive facts** are measures that **can be summed across *some* dimensions but NOT all**
* They are **usually numeric values that can only be aggregated across certain dimensions**, such as customers or products
* Examples include account balance + inventory levels, respectively
* These **require special handling in data analysis to ensure that the aggregation is done correctly and does not spill over to non-compatible entity instances**

##### 3) Non-Additive Facts

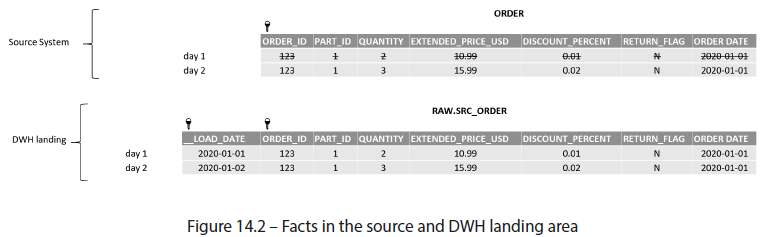
* **Non-additive facts** aremeasures that **cannot be summed across *any* dimension**
* They are usually ratios, percentages, or other **derived values that cannot be aggregated meaningfully**
* Examples include the average price per unit or customer satisfaction score
* They **require special handling in data analysis + are typically used in more complex analytical calculations**, such as forecasting or predictive modeling
* **In most cases, breaking non-additive facts into their fully additive *components*** (e.g., the price and number of units instead of the average price per unit) **is encouraged**, as it gives users the **flexibility to aggregate at any granularity + re-calculate the derived value**
* Having understood the different types of fact tables + the measures they contain, we can look at the challenges involved in designing + maintaining fact tables, keeping in mind that where there’s a challenge, there’s a Snowflake feature to help overcome it

#### Getting the Facts Straight

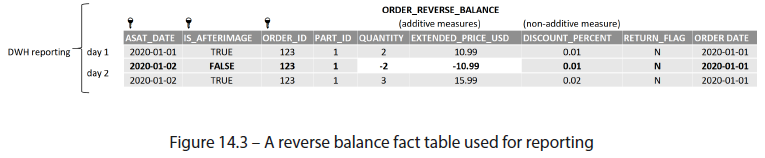
* The **facts *in a source system* are recorded in real time + updated in case of adjustments**
* **By definition, they are always current**
* A **DW has a much harder task b/c it needs to capture + report *current* facts *and* track *historical* changes**
* Suppose an order was adjusted from containing 2 items to 1:
* The DW must find a way to report that a change was made while avoiding the issue of double-counting (as the total quantity is now 1, not 3)
* The **task of historical tracking is made even more complicated when facts are not point-in-time transactions but instead are intervals** such as advertising campaigns or employee hires and leavers
* In such cases, tabulating the cost of a department can no longer be accomplished by simple aggregation b/c employees can come + go at various intervals
* **Operating a business is messy + complex, and the data that it generates is no exception**
* Employees come + go, orders are returned, and in some cases, records in the source system may be physically deleted, leaving no artifact for the DWH to detect + reconcile
* We will address these age-old reporting challenges using versatile fact table designs incorporating unique Snowflake features to minimize the costs and required maintenance

##### Reverse Balance: The World’s Most Versatile Transactional Fact Table

* When it comes to recording business transactions, **capturing changes is often as important as their final states**
* Ex: In online retail, being able to analyze products that were removed from a shopping cart can yield valuable insight for understanding customer behavior + minimizing such actions in the future
* However, **tracking changes in a fact table presents a dual challenge:**
* **1) Tying a changed record to its previous state**
* **2) Storing both states w/out distorting the overall totals**
* The following example shows a sales order placed on day 1 that suffers a change in the source system on day 2 + the resulting records that land in the DW



* While the record in the **source system is overwritten by the UPDATE**, a **DW captures *both* versions**
* However, **it is difficult to answer even simple business questions by looking at the RAW table**:
* What is the total number of parts sold? (3, not 5)
* How many parts were sold on day 2? (1, not 3)
* To help the business answer these (+ many other) questions, we **can use a fact table design known as Reverse Balance or Mirror Image**
* **The Reverse Balance method yields a fact table that can answer virtually any business question using aggregation (*the operation for which OLAP systems are optimized*)**
* As the name suggests, in a reverse balance structure**, a change is performed by inserting not one but *two* records; one that contains the *new* values, another that *negates* the original (the titular reverse balance)**
* The result in our example would appear as follows, containing (from top to bottom) the original value, its reverse balance, + the updated value:



* The **reverse balance table resembles the raw table w/ a few key differences**:
* 1) The **landing table *load date* becomes an as-at date (ASAT\_DATE), indicating when a given version of a fact is valid**
* 2) A **before/after-image indicator** **(IS\_AFTERIMAGE)** **is added**
* After-image flag **distinguishes the reverse image** (middle row) **from the after-image** (3rd row) + is therefore **included in the PK**
* This approach makes it easy to answer the prior business questions:
* -- total number of parts sold

SELECT SUM(quantity) FROM lineitem\_reverse\_balance

* -- parts sold on day two (delta)

SELECT SUM(quantity) FROM order\_reverse\_balance

WHERE asat\_date = '2020-01-02'

* In addition, **as-at date makes it possible to observe historical facts *as they appeared at that point in time***
* For example, what is the number of parts sold as is on day 1 vs. day 2?
* -- parts sold on first of Jan as at day one

SELECT SUM(quantity) FROM order\_reverse\_balance

WHERE order\_date = '2020-01-01'

AND asat\_date= '2020-01-01'

AND is\_afterimage = TRUE

* -- parts sold on first of Jan as at day two

SELECT SUM(quantity) FROM order\_reverse\_balance

WHERE order\_date = '2020-01-01'

AND asat\_date= '2020-01-02'

AND is\_afterimage = TRUE

* Using this method, **analysts can quickly aggregate + compare totals, deltas, point-in-time snapshots, + year-end totals as they appear today or *at any point in the past***
* Later, we will review an efficient technique for constructing + maintaining a reverse image fact table in Snowflake
* The only thing we require is the updated record to be loaded in the landing area
* Unfortunately, the **data quality in some source systems is less than ideal, where records are physically *deleted* instead of being canceled or corrected**
* Luckily, the **DW can *look back in time* + recover deleted records to align them w/ the source**

##### Forward-Leading-Insertion: The Leading Method for Recovering Deleted Records

* **In a perfect world, records are never physically deleted in the source system**
* Instead, they **should be nullified *logically* by marking them w/ a *deleted* flag + setting the measures to 0**, but as we know, the world is not perfect, + records are occasionally removed
* **Deletions are not a grave problem in *dimension* tables** b/c a quick aggregation from the landing area table can produce + identify the latest active + deleted entries:
* SELECT

<dimension-PK>

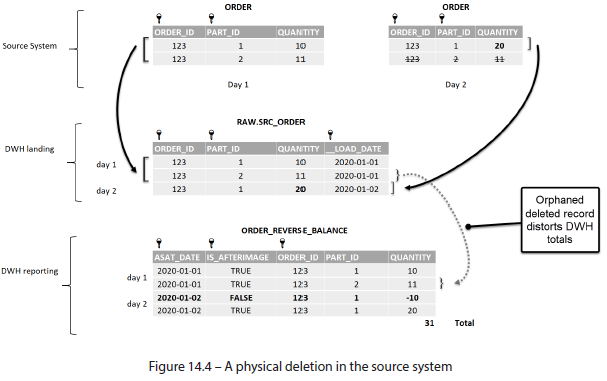
, MAX(load\_date) AS latest\_load\_date

, IFF(latest\_load\_date = current\_date, false, true) AS is\_deleted

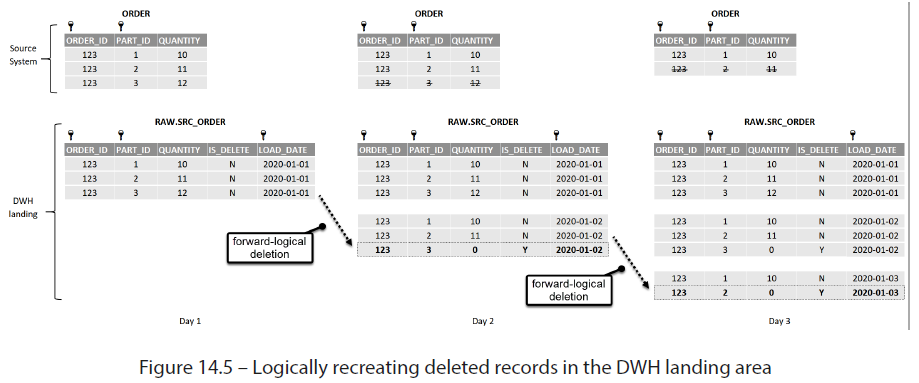
FROM src\_dimension

group by <dimension-PK>

* This is to say**: in a full daily load, any records that don’t exist but have been loaded previously must have been deleted**
* **Facts pose a greater challenge b/c they occur at a precise point in time + are typically too numerous to be loaded + compared using full loads**
* Instead, **facts are typically loaded + processed as** **delta snapshots**, **containing only changes and updates that occurred *since the previous load***
* Whether using a reverse balance fact table or any alternative, the **issue of orphaned records arises when the DW is not informed of a deleted fact**
* In the following example, an order was placed for 2 parts on day 1
* On day 2, the number of units for part 1 was increased to 20, but part 2 was (improperly) canceled by a physical deletion of the record.



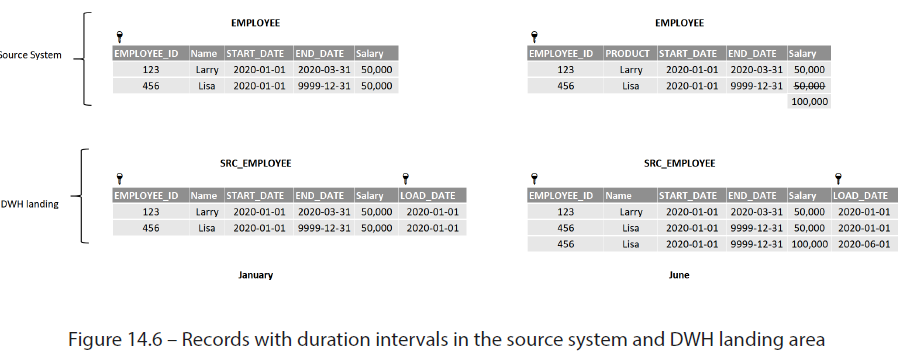
* Physical deletion is a quick (+ tempting) way to adjust data in a source system b/c in there, it poses no risk to data accuracy (as of day 2, the system correctly reflects the order for 20 units of part 1)
* However, the **DW is *not* informed of the change + is now out of sync w/ the official source of truth**
* **Unlike dimension entities, looking backward to detect + rectify deleted records (facts) in a DW is a resource-intensive task that usually involves aggregates, temporary tables, window functions over large datasets, + comparisons**
* However, **by turning the process on its head + looking *forward*, we can drastically reduce the processing resources necessary to identify + revert deleted changes**
* The **basic concept of the Forward-Leading-Insertion method is to insert (+ logically delete) the physically-deleted record into the *next load date* in the raw/landing table**
* Let’s say an order is created on day 1 containing 3 parts
* A part is subsequently deleted for days 2 and 3
* Below illustrates the result of running the process on the DW landing area table.



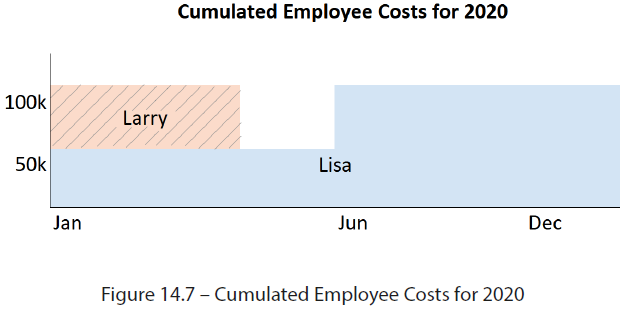
* This **logical deletion process aims to mark the physical deletion *as it happened in a point-in-time snapshot***
* Therefore, the **logically deleted record (IS\_DELETE = Y) is inserted *once* + is NOT passed down to future load dates, as can be seen with PART\_ID 3 on day *two***
* Later, we write the script that will allow us to perform this operation for an *entire* fact table or restrict it to a single load date to economize DW credits for a daily load
* However, there is **one more fact table scenario that we must learn to handle first: range-based facts**
* **For range-based facts** such as employee active periods or promo campaigns, **a *single* as-at date is insufficient, + a *mix of intervals* is required**
* Knowing how to construct a fact table to handle such records efficiently can save developers from writing some truly grisly analytical queries down the line

##### Type 2 Slowly Changing Facts

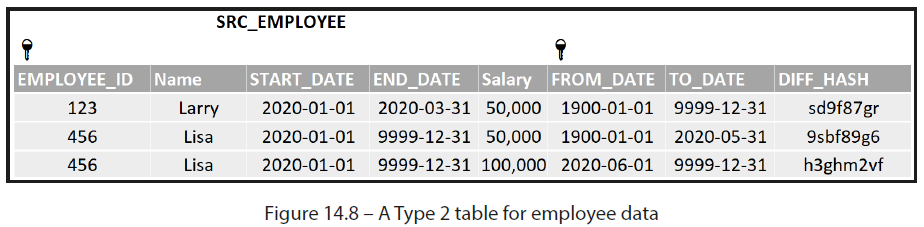
* **While some business operations occur at a given moment**, such as sales, **others**, such as contracts, **endure for a length of time**
* **Complex business data + fact-less fact tables often blur the line between facts + dimensions**
* Ex: An EMPLOYEE table may be used as a *dimension* when reporting store sales, but also as a *fact* when analyzing the **operational expenses (OPEX)** of a given department (e.g., a salary multiplied by the % of active service during a given period)
* In the latter scenario, analytics can be tricky
* Ex: A start-up hires 2 employees at the start of 2020: a contractor, Larry, due to leave in 4 months, + a full-time employee, Lisa
* In June, as the lone active employee in the company, Lisa is given a raise
* The source and DWH snapshots in January and June would look as follows:



* W/ an eye toward tracking costs, the company’s owners wish to know the *exact* OPEX costs of keeping the company running
* They need a fact table allowing them to generate the figures as they appear below:



* The **solution is to use the Type 2 SCD structure** **(new row)** covered in the previous chapter
* **Storing employee data in a Type 2 table allows you to query date intervals at a point in time and at ANY *effective* date or date range**
* A Type 2 fact table for this example would contain the following data:



* A **surrogate low-date such as the one in the FROM\_DATE column is often used**, as it **allows users to identify original/unmodified versions of a record**
* Otherwise, the creation date can be used instead
* Having the data in this format allows us to write a query to answer the OPEX question w/out having to use window functions or compare records:
* SELECT name

, IFF (start\_date > from\_date, start\_date, from\_date) AS query\_from\_date

, IFF (end\_date < to\_date, end\_date, to\_date) AS query\_to\_date

, salary

FROM employee

WHERE TRUE

AND start\_date <= '2020-01-01'

AND end\_date >= '2020-12-31'

* This simple example uses only *one* time band (employee start/end date), besides the meta from\_date/to\_date columns
* However, *in the real world*, **entities may contain *multiple* date ranges + *sets* of additive, semi-additive, + non-additive measures**
* Ex: A “contract” may have the following date attributes: service start/end, creation, modification, signing, authorization, release, renewal, review, + many others, which the business teams may wish to interrogate
* The **Type 2 structure**, as we will see, **allows us to easily obtain counts, aggregates, + snapshots for any *date* range or measure required by the business**
* Having seen the approaches that need to be taken to meet some of the most frequent fact table challenges, let us dive into + explore code that allows us to meet these w/ minimal effort + WH credit usage.

#### Maintaining Fact Tables Using Snowflake Features

* In this section, we practice creating + maintaining the fact table techniques discussed previously using available Snowflake features
* Like in previous chapters, load data from the *snowflake\_ sample\_data.tpch\_sf10* schema, which will serve as the sample set
* We will then **simulate source system updates by randomly loading records** from this sample
* The first 2 exercises will use data from the LINEITEM table
* To continue, create a schema to house the exercises from this chapter + instantiate the source system table, as well as the DW landing area.
* Open the 1st file in this chapter’s repository, ch\_14.1\_reverse\_balance\_fact.sql, + run the first 3 steps:
* CREATE OR REPLACE SCHEMA ch14\_facts;
* CREATE OR REPLACE TABLE source\_system\_line\_item ...;
* CREATE OR REPLACE TABLE src\_line\_item ...;
* These examples require line item orders to be loaded + processed *in their entirety* (containing all constituent line items for a given order)
* To accomplish this, order IDs are first selected at random from the sample set, + then all related line items are loaded:
* WITH complete\_orders AS (

SELECT DISTINCT sales\_order\_id

FROM source\_system\_line\_item SAMPLE (< N > rows)

)

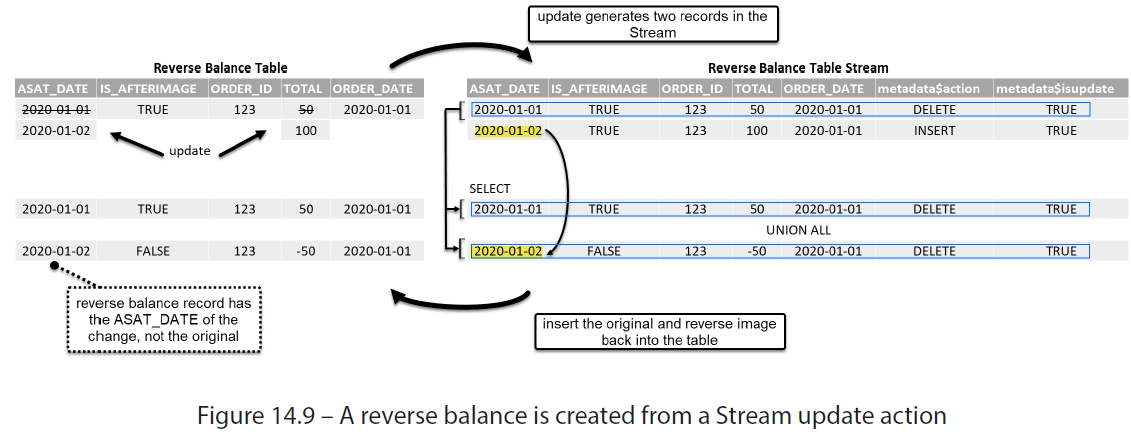
* SELECT < source columns > FROM source\_system\_line\_item src

INNER JOIN complete\_orders co ON src.sales\_order\_id = co.sales\_order\_id

* Let’s begin by constructing the model for a **reverse balance fact table** + building a script to load it **using a MERGE/INSERT strategy using Streams**

##### Building a Reverse Balance Fact Table with Streams

* While *querying* a **reverse balance fact table** is simple, traditional databases have often run into performance constraints when attempting to *maintain* one
* This is **due to the difficulty of isolating the *previous* version of a change (to create the reverse balance)**
* This operation **typically requires temp tables + window functions** to LAG + compare the same business key across various load dates
* However, using **Snowflake Streams**, we can **generate the reverse balance by simply negating the additive measures in the before-image of an updated record, a massive performance improvement over the traditional method**
* This is a similar strategy to the one used in the previous chapter for Type 2 SCDs
* Continue with the exercise from the file (*ch\_14.1\_reverse\_balance\_fact.sql*) we started with:
* After cloning *src\_line\_item* as a backup for re-running the exercise, **create the structure of the reverse balance table + populate it with an initial dataset (*before* any changes occur)**:
* CREATE OR REPLACE TABLE line\_item\_rb ...;
* **Note the characteristic columns that this type of table requires**:
* **Load type 🡪** An optional but useful metadata column **that identifies the type of change in a record**
* The values in this column are user-defined to fit the nature of the data + the types of changes that occur
* Our example uses values of *initial* and *update* to distinguish between newly created + updated records
* The exercise on recovering deleted records will use this field to mark *deletion* records.
* **As-at date 🡪** Although its value is equivalent to the load date, the term “as-at” paints a clear picture of the column’s function: **identifying the state of a fact at any point in time**
* Although facts are typically loaded daily, requiring a date type for this column, our example is meant to be run multiple times per day, so a timestamp is used instead
* **Before/after-image flag 🡪** used to **split out an update into the new version + its reverse balance**
* As such, it ***must* be included in the PK of the table** (**in addition to the as-at date and the business key**)
* With the reverse balance table created, create a **Stream to handle the upcoming changes:**
* CREATE OR replace STREAM strm\_line\_item\_rb ON TABLE line\_item\_rb;
* Now, **simulate a load from the source system**, which will **include new records**, as well as **updates to ones already** **loaded**:
* INSERT INTO src\_line\_item ...;
* Now, **MERGE the newly loaded records into the reverse balance table using its PK**
* Our example assumes only new + updated records and can’t contain existing line items w/ no changes
* If this *were* the case, use *diff\_hash* to determine whether an existing record does indeed constitute a change:
* MERGE INTO line\_item\_rb AS rb ...;
* The result of the **merge will insert new orders + update existing orders**, + it is the **update that concerns us**
* Because we overwrote the *original* order values, we now have to insert the *previous* version, + its reverse balance
* The process is illustrated in the following diagram:



* As we need to generate 2 records (the original + the reverse-balanced one) from a *single* record in the Stream, we can use a CTE to select the row once + insert it twice using UNION ALL
* As demonstrated in the figure above, the original record is inserted as it appears in the Stream, while the following modifications must be made to the reverse image:
* The as-at date is that of the latest record, NOT the original
* The before/after-image flag is marked as BEFORE
* Additive + semi-additive measures are multiplied by -1
* When you run the insert from the Stream (INSERT INTO lineitem\_rb ... ;), observe where these operations take place:
* INSERT INTO lineitem\_rb

WITH before\_records AS (

SELECT \*, asat\_after FROM strm\_lineitem\_rb

WHERE metadata$action = 'DELETE'

)

--insert the original after image that we updated

--no changes required to column values

SELECT < stream fields > FROM before\_records

UNION ALL

--insert the before image as at after dts, but negate additive measures

SELECT < stream fields >

, asat\_after as ASAT\_DTS --USE the asat OF the AFTER image

, FALSE as IS\_AFTERIMAGE --because this IS the BEFORE image

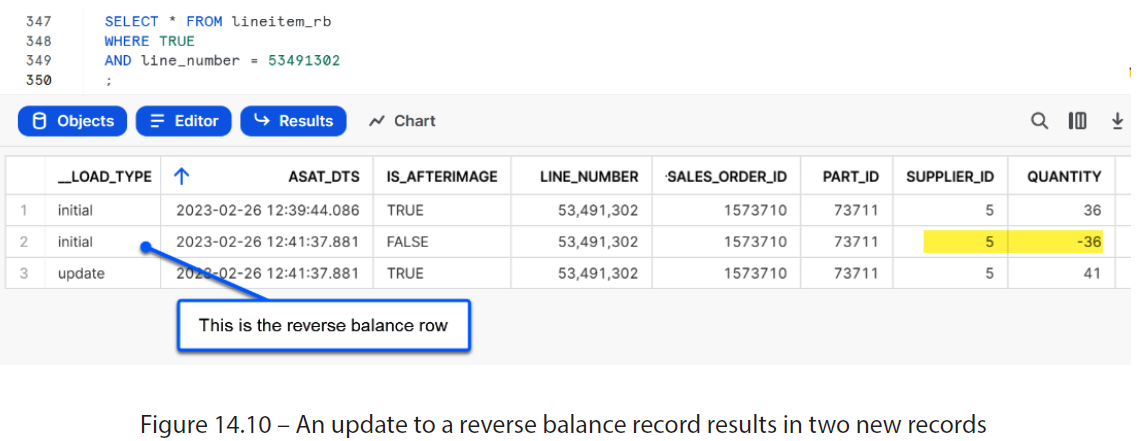
, -1 \* quantity

, -1 \* extended\_price\_usd

, discount\_percent --do NOT negate, non-additive measure

FROM before\_records;

* Now that the reverse balance table has been fully updated, select an *update* record to verify and inspect the result
* A sample line item update should generate 3 records, as in the following screenshot:



* The **reverse balance fact table** is a powerful tool b/c it **allows us to query the state of truth as it appeared on any date**
* Ex: In sales or reservations, a registered transaction may still be canceled in the future, but the as-at date gives us a simple filter with which to travel back in time to observe historical records as they appear today, or at any point in the past
* In the following example, we can query last year’s bookings as they appear today (with cancelations) + as they appeared last year (before this year’s cancelations):
* --last year bookings with future cancelations

SELECT sum(sales)

FROM bookings

WHERE booking\_date = <LAST\_YEAR>

AND asat\_date <= <TODAY> --implicit

--last year bookings without future cancelations

SELECT sum(sales)

FROM reservations

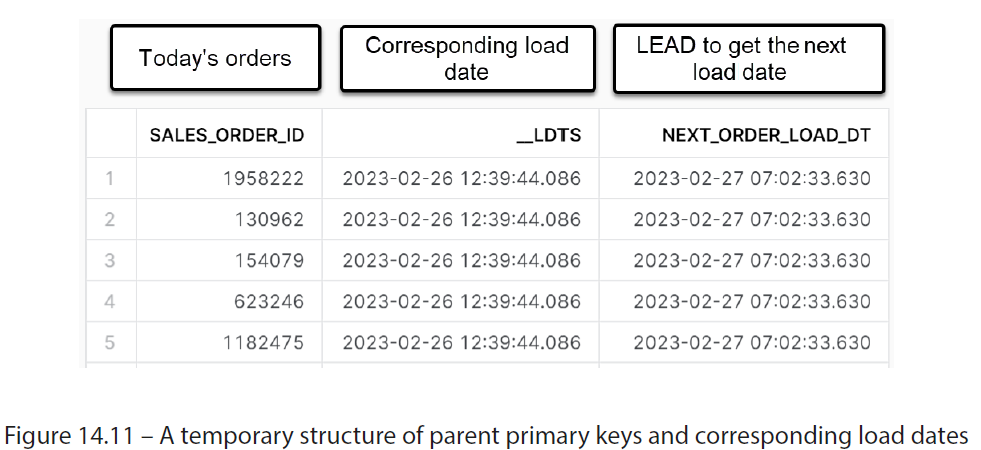
WHERE booking\_date = <LAST\_YEAR>

AND asat\_date <= <LAST\_YEAR>

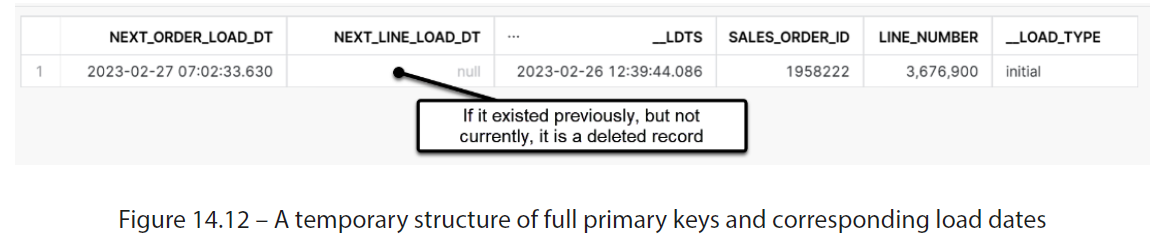
* Now we have a working fact table, + we’ll demonstrate how to detect + recover physically deleted records to set the correct balance in the landing area, so that the fact table perceives the event

##### Recovering Deleted Records with Leading Load Dates

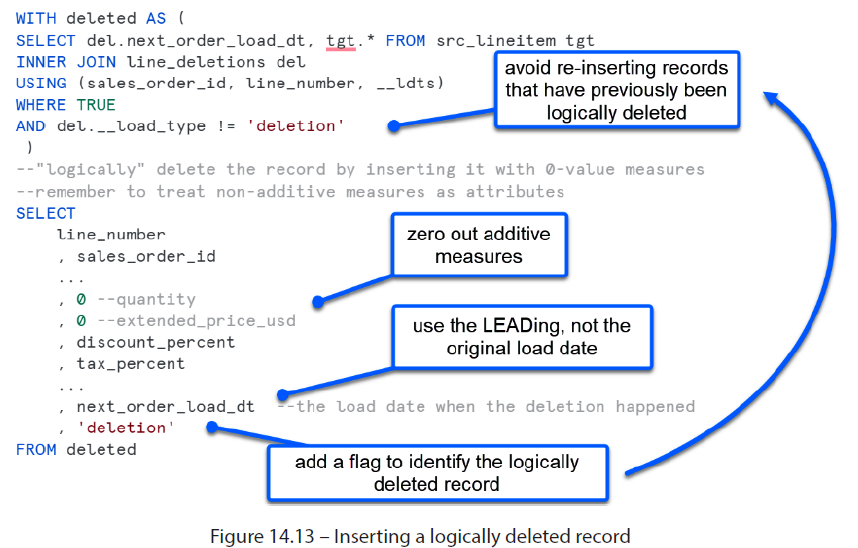
* This exercise will build on the objects created in the reverse balance example
* Open *ch\_14.2\_deleted\_records.sql* from the repository to get started
* First, use the backups from the previous example to reset the source objects to their initial state:
* CREATE OR REPLACE TABLE src\_line\_item CLONE src\_line\_item\_bak;
* CREATE OR REPLACE TABLE line\_item\_rb CLONE line\_item\_rb\_bak;
* CREATE OR replace STREAM strm\_line\_item\_rb ON TABLE line\_item\_rb;
* Now, we will simulate order updates that contain deleted records by filtering out line numbers ending in any number less than 3
* Run INSERT INTO for the source table: INSERT INTO src\_line\_item ...;
* Now comes the task of **detecting + recovering the deletions**
* Although this can be done as a *single* operation, this example has split the process out **using temp tables** to allow for testing + debugging.
* The 1st step is taking the orders in the current load + building a structure that contains the PK of the parent object (sales\_order\_id) + the corresponding load dates + using the **LEAD window function** to obtain the *next* parent object load date
* Build the temp table using the following command and observe the result:
* CREATE OR REPLACE TEMPORARY TABLE line\_load\_hist ...;
* This generates a structure of today’s order IDs w/ corresponding + leading load dates:



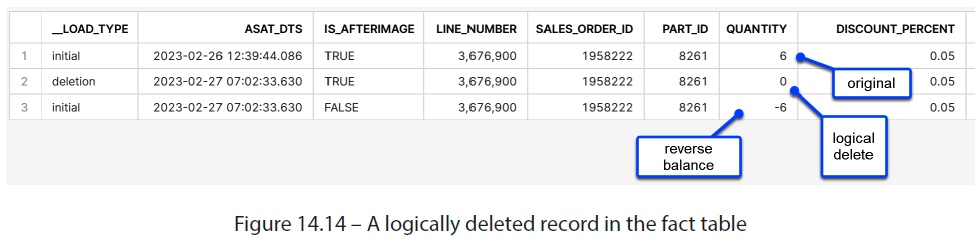
* Knowing the current + leading load dates for the order IDs, we can now do the same for the associated *line numbers*
* Run the temporary line\_deletions table by running the following command:
* CREATE OR REPLACE TEMPORARY TABLE line\_deletions ...;
* By inner-joining the line number w/ the previously created list of order IDs + load dates, we’re able to dramatically reduce the number of records that needs to be processed using the LEAD window function
* Having done so, deleted line items can be identified b/c they do NOT have a leading load date when we look forward
* A sample deleted record would look like this:



* Now that we have identified the deleted records + when they were last loaded, we can INSERT from the source table, zeroing out the additive measures
* Run the following statement to *logically* delete the missing records:
* INSERT INTO src\_lineitem...;
* Observe that during this process, we zero out the additive measures + use the LEAD load date that we obtained for the missing record
* Also, it is essential to create a **flag** that identifies a logically deleted record for the following reasons:
* We need to use it to avoid re-inserting logically deleted records in the future (as they won’t exist in any subsequent load either)
* We need to have a way to reconcile w/ the source system if asked why there are more records in the DWH
* This is all handled during INSERT, as seen here:



* Now that the missing records have been re-inserted as logical deletions, we can observe the impact on the fact table by loading it as in the previous example
* Note that although this example used a reverse balance fact table, **logical deletions *must* be performed in the landing area to avoid distorting the totals, *regardless of the type of fact table used***
* Load the fact table by running the two commands discussed previously:
* MERGE INTO line\_item\_rb rb...;
* INSERT INTO line\_item\_rb...;
* The fact table has now been updated to allow us to aggregate as-at totals correctly
* A sample deletion would generate 2 records to net out the original as seen here:



* Adjust the example parameters to repeat the exercise to see how new records + deletions on top of already deleted line items impact the final fact table.
* Having covered **point-in-time fact tables** + how to treat **possible deletions**, we will move on to an example that teaches us how to maintain + interrogate interval-based facts

##### Handling Time Intervals in a Type 2 Fact Table

* In this exercise, we will experiment w/ employee data that simulates the daily operations of an active business: hiring/terminating employees, adjusting salaries, + balancing headcounts between fixed-term contractors + full-time employees
* Open the *ch\_14.3\_type2\_facts.sql* file from the chapter repo to get started
* As the sample data contains records from within the range 1992-1998, we will use a variable to set *today’s* date to *1995-12-01* to allow us to load the historical data for *existing* employees, + also give us plenty of sample data for *future* changes:
* SET today = '1995-12-01';
* Now, instantiate the sample dataset for the simulated source system + the initial DW landing area:
* CREATE OR REPLACE TABLE source\_system\_employee...;
* CREATE OR REPLACE TABLE src\_employee...;
* In creating the initial tables, we can use the today **variable**, just as we’d use the current\_date() function in a real-world scenario
* Next, we create the Type 2 fact table using the method described in the previous chapter:
* CREATE OR REPLACE TABLE employee\_t2 ...;
* Note that we will use the surrogate low/high dates of 1900-01-01 and 9999-12-31 for the meta from\_date/to\_date fields, respectively
* W/ the Type 2 table created, increment the today variable by 1 + start to insert changes into the landing area: new hires, terminations, + promotions for some existing employees:
* SET today = $today::date + 1;
* INSERT INTO src\_employee...;
* Unlike the reverse balance fact table, **a Type 2 fact table does NOT discriminate between additive and non-additive measures, as it updates *all* changes accordingly + creates a *new* effectively dated record**
* Update the fact table w/ the latest records:
* MERGE INTO employee\_t2 tgt...;
* INSERT INTO employee\_t2...;
* Now, **increment the today variable + simulate several more days of changes to generate some meaningful data to query**
* Once several load dates have been added, we can begin to ask business questions about our organization’s headcounts
* The surrogate high date will *always* return the *latest* version of a fact, so we can use it as a filter when asking questions about the current state of the company, such as “*how many employees are currently active?“*
* --currently active employees

SELECT COUNT(\*) cnt FROM employee\_t2

WHERE TRUE

AND is\_active

AND to\_date = '9999-12-31'; --currently

* *What about at a point in the past?*
* Here, we must consider a ***range*** of dates: records w/ an effective date valid *before* the target date, and *after*
* For example, a contractor w/ a start-to-end range of 1994-12-01 to 1995-12-31 should match the target date that we’re interested in (1995-12-01):
* --active employees on day 1995-12-01

SELECT COUNT(DISTINCT employee\_id) cnt

FROM employee\_t2

WHERE TRUE

AND is\_active

AND from\_date <= '1995-12-01'

AND to\_date >= '1995-12-01' ;

* Instead of a point in time, **the query can use a range as well**, for example, to calculate (*w/out double-counting changes*) the number of active employees for the entire year of 1995:
* --active employees in all of 1995

SELECT COUNT(DISTINCT employee\_id) cnt FROM employee\_t2

WHERE TRUE

AND is\_active

AND YEAR(from\_date) <= 1995

AND YEAR(to\_date) >= 1995 ;

* Using various time criteria, we can mix current + historical values to ask targeted questions such as “*who was hired in Q1 1994 last year and was still active on the date of 1995-12-01?“*
* --active employees on day 1995-12-01

--who were hired in Q1 of 1994

SELECT COUNT(DISTINCT employee\_id) cntMaintaining fact tables using Snowflake features 235

FROM employee\_t2

WHERE TRUE

AND is\_active

AND hire\_date BETWEEN '1994-01-01' AND '1994-03-31'

AND from\_date <= '1995-12-01'

AND to\_date >= '1995-12-01';

* Yes, but now, only show me those who received a promotion. We can query this table, adding ever more complex criteria:
* --active employees on day 1995-12-01

--who were hired in Q1 of 1994

--and received a promotion

WITH promotions AS (

SELECT DISTINCT employee\_id FROM employee\_t2

WHERE TRUE

AND last\_change = 'Promoted' )

SELECT COUNT(DISTINCT employee\_id) cnt

FROM employee\_t2

INNER JOIN promotions USING (employee\_id)

WHERE TRUE

AND is\_active

AND hire\_date BETWEEN '1994-01-01' AND '1994-03-31'

AND from\_date <= '1995-12-01'

AND to\_date >= '1995-12-01';

* Whether it’s capturing distinct groupings by date (as seen in the following example) or aggregating totals over a range of dates, the Type 2 fact table can handle any range-based query that users throw at it:
* --what are the total changes per day by change type

--since the first load ( excluding 1995-12-01)

SELECT from\_date, last\_change, COUNT( employee\_id) cnt

FROM employee\_t2

WHERE TRUE

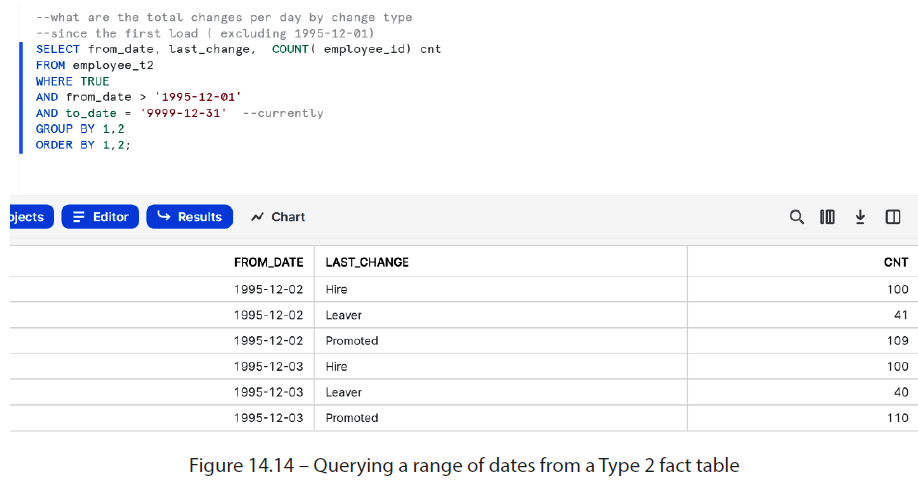
AND from\_date > '1995-12-01'

AND to\_date = '9999-12-31' --currently

GROUP BY 1,2

ORDER BY 1,2;

* A following screenshot shows the result of daily headcount movements since starting the exercise:



* **When facts *pivot* around several different dates + date ranges, a Type 2 configuration is the ideal table structure to take advantage of the performance of aggregate functions for a columnar database such as Snowflake**

#### Summary

* **In a DW, fact tables present an additional challenge on top of merely capturing the latest values, as they must *also* be able to capture + reconcile historical changes in a way that allows users to flexibly + cost-effectively query them to resolve business questions**
* **B/c when it comes to *operational* analytics, analyzing changes, variations, + what didn’t happen can be just as valuable as the current state of truth**
* **Various types of fact tables exist** to help an organization meet these demanding analytical needs, such as **transactional**, **snapshot**, and **accumulating snapshot** fact tables, among others
* These fact tables **must differentiate between various kinds of measures they store (additive, semi-additive, non-additive), b/c each is treated differently when updating or recording changes**
* To help data teams construct + maintain these tables in Snowflake, this chapter dissected some of the toughest challenges in maintaining fact tables in a DW + came up w/ cost-effective recipes for learning to tackle the challenges using Snowflake-native features
* Even if you don’t encounter these exact scenarios in your work, the concepts + features demonstrated in these examples will surely provide a sound first-principles framework that can be applied to similar situations
* Having tackled the traditional elements of RDBs, such as facts + dimensions, we will focus on semi-structured data next, to see how elegantly Snowflake allows users to handle it with table-like performance