# 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 15: Modeling Semi-Structure Data

* So far, this book has focused on modeling ***structured* data**, the kind used in RDB’s since the early 70s
* However, w/ the rise of the internet, a different style of data became prevalent: **semi-structured**
* **Semi-structured data**, such as website traffic + social media feeds, contain ***some* organizational structure but do NOT conform to the formal structure of an RDB**
* New file formats also emerged to support this new type of data, starting w/ the advent of **Extensible Markup Language** (**XML**) in the early ‘00s, followed by **JavaScript Object Notation** (**JSON**), and, w/ the rise of distributed computing, formats such as **Avro, ORC, + Parquet**
* These formats offered **a lightweight + flexible way to structure data, making them ideal for web-based and mobile app data**
* The popularity of **semi-structured data** can be attributed to its **flexibility**, **adaptability**, + **ability to handle data sources that do not fit neatly into traditional RDBs**
* As a modern cloud data platform, **Snowflake is capable of natively ingesting semi-structured data, storing it efficiently, + accessing it using simple extensions to standard SQL**
* However, to *consumers* of semi-structured data, the **challenge of wrangling nested entities and establishing meaningful relationships remains**
* This chapter will explore tools Snowflake offers to make working w/ semi-structured data easy
* Using these tools, we will learn how to analyze + deconstruct semi-structured data into a relational model that BI tools + traditional analytics can consume
* Main Topics:
* The benefits of semi-structured data
* How Snowflake makes working w/ semi-structured data easy
* The flexibility of schema-on-read
* Techniques for flattening semi-structured data into rows
* The benefits of transforming semi-structured data into a relational schema
* A rule-based method for transforming semi-structured data into relational data

#### 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/ch15>
* While the key section of each script will be explained in the latter half of this chapter, refer to the repository for the complete code required to load, query, + transform semi-structured data

#### The Benefits of Semi-Structured Data in Snowflake

* **Semi-structured data formats are popular due to their flexibility when working w/ *dynamically varying information***
* Unlike relational schemas, in which a precise entity structure *must* be known + fixed ahead of time, **semi-structured data is free to include or omit attributes as needed, as long as they are properly nested w/in corresponding parent objects**
* Think of the contact list on a phone
* It contains a list of people + their contact details but does *not* capture those details *uniformly*
* For example, some contacts may contain multiple phone numbers while others have one
* Some entries contain information such as an email address + street address, while others have only a number + a vague description in lieu of a name
* **To handle this type of data, Snowflake uses the VARIANT data type, which allows semi-structured data to be stored as a column in a relational table**
* **As w/ *all* column types, Snowflake optimizes how VARIANT data is stored internally, ensuring better compression + faster access**
* **Not only can semi-structured data sit next to relational data in the same table, but users can also access it using basic extensions to standard SQL + achieve similar performance**
* Another compelling reason to use the **VARIANT** data type for semi-structured data is **its adaptability to change**
* If columns are added or removed from semi-structured data, there is no need to modify **ELT** pipelines
* **The VARIANT data type does not care whether the schema changes, + even read operations won’t fail for an attribute that no longer exists**
* Let’s load some semi-structured data to see these features in action

#### Getting Hands-On with Semi-Structured Data

* Although we will query semi-structured JSON data as part of this exercise, **semi-structured storage still conforms to modeling best practices such as naming and standard columns**
* In this example, we will use semi-structured data containing information about pirates (such as details about crew, weapons, + their ship) all stored in a single VARIANT data type column
* **W/ relational data, a row represents a *single* entity, but in *semi-structured data*, a row is an *entire file* (although the file itself can contain single or countless entities)**
* For this reason, **metadata columns to mark individual loads + source filenames are stored alongside VARIANT**

A computer screen shot of a computer

Description automatically generated

* This example uses AUTOINCREMENT (a.k.a. IDENTITY) as the default to generate a sequential unique ID for each load/record inserted
* **In a real-world scenario, semi-structured data would be loaded into external stages + *then* to Snowflake tables using Snowpipe (streaming) or COPY** **INTO** **(bulk)**
* NOTE: Snowpipe instructions can be found on the Snowflake website:
* <https://docs.snowflake.com/en/user-guide/data-load-snowpipe-intro>
* However, in *this* example, we will learn how to create JSON data on the fly + work w/ that instead
* But first, an **overview of semi-structured data in a JSON file**:
* **JSON files** **contain nested objects, which consist of name-value pairs (a.k.a key-value pairs) and have the following properties**:
* **Data is stored in name-value pairs**
* Data is **separated by commas**
* **Curly braces hold objects**
* **Square brackets hold arrays**
* **Object + array values may *also* be (nested) objects + arrays**
* **Values (including object + array values) can be one of the following data types:**
* An object
* An array
* A Boolean
* A string
* A number
* NULL
* The following figure shows a simple example of data in JSON format:

A screenshot of a computer code

Description automatically generated

* For this exercise, we will use a slightly more advanced example
* Load the data into the previously created table using the code provided in the repo:
* INSERT INTO pirate\_json...;
* Now, we are ready to perform some analysis

#### Schema-On-Read is NOT Schema-No-Need

* W/ the rising popularity of semi-structured data, **schema-on-read** also entered the lexicon of big data, which is **the idea that, unlike in relational modeling, the schema definition for semi-structured data can be delayed until long after the data has been loaded into the data platform**
* **Delaying this task means there are no bottlenecks w/in the ETL process for generating and ingesting semi-structured data**
* ***However*, implicit in the design is that a knowable schema exists underneath the flexible semi-structured form**
* In this section, we will learn how to query JSON data + infer details about its contents using SQL and Snowflake-native functions
* Let’s begin by extracting some basic attributes for our pirate via SELECT \* FROM pirate\_json;
* Although we can query a table containing semi-structured data in a VARIANT column, a simple SELECT \* statement does not return meaningful results:



* **To access information stored inside VARIANT, Snowflake uses the colon operator**
* Using this method, let’s find out what pirate we are dealing w/
* Recall that *v* is the name of the variant column in the pirate\_json table:
* SELECT v:name AS pirate\_name\_json

, v:name::STRING AS pirate\_name\_string

, v:nickname::STRING AS pirate\_name\_string

FROM pirate\_json;

A close-up of a test

Description automatically generated

* **B/c Snowflake returns JSON data in quotes + often contains similar attribute names for nested objects, casting + aliasing the results is encouraged for readability**
* Now, let’s **move down a level and query a *sub*-column using a familiar dot notation:**
* SELECT v:name::STRING AS pirate\_name

, v:ship.name::STRING AS ship\_name

FROM pirate\_json;

A close up of words

Description automatically generated

* Remember the claim that **semi-structured data is flexible enough to handle new + deleted attributes**?
* **See what happens when we query a column that doesn’t exist (spoiler alert, there’s no error):**
* SELECT v:name::STRING AS pirate\_name

, v:loc\_buried\_treasure::STRING AS pirate\_treasure\_location

FROM pirate\_json;

A close-up of a piece of paper

Description automatically generated

* Now, let’s learn how to **interrogate an array**
* As in other programming languages, **arrays in Snowflake represent an indexed set of objects**
* **Individual elements can be selected using familiar square-bracket notation, + common functions such as ARRAY\_SIZE/CONTAINS/ADD are available**
* Let’s get some information regarding Blackbeard’s reign of terror:
* SELECT v:name::STRING AS pirate\_name

, v:years\_active AS years\_active

, v:years\_active[0] AS active\_from

, v:years\_active[ARRAY\_SIZE(v:years\_active) - 1] AS active\_to

FROM pirate\_json;

A close-up of a number

Description automatically generated

* Now, let’s learn how to ***dynamically* handle multiple repeating values**
* **Although arrays can be queried *directly*, their nested values remain in VARIANT format and are NOT treated as rows**
* Let’s try and get to know two of Blackbeard’s crew members:
* --query multiple elements

SELECT v:name::STRING AS pirate\_name

, v:crew::VARIANT AS pirate\_crew

FROM pirate\_json;

A close up of a name

Description automatically generated

* Unfortunately, the result is not user-readable
* **If we’d like to pivot the elements of an array into intelligible rows, we must use 2 Snowflake features *in conjunction*: lateral and flatten**
* <https://community.snowflake.com/s/article/Using-lateral-flatten-to-extract-data-from-JSON-internal-field>
* **Lateral join** – A **LATERAL join behaves similarly to a loop in a correlated subquery + can reference columns from a table expression**:
* SELECT ...

FROM <left\_hand\_table\_expression>, LATERAL (<inline\_view>)

* for each row in left\_hand\_table LHT:

execute right\_hand\_subquery RHS using values from current row in LHT

* <https://docs.snowflake.com/en/sql-reference/constructs/join-lateral>
* **Flatten** – This is a **table function that takes a VARIANT, OBJECT, or ARRAY** **column and produces a lateral view (i.e., an inline view that contains correlation referring to other tables that precede it in the FROM** **clause)**
* More information on FLATTEN + its parameters can be found in the Snowflake documentation at <https://docs.snowflake.com/en/sql-reference/functions/flatten>
* Please review this documentation + familiarize yourself w/ its list of **output columns** before continuing

A screenshot of a computer

Description automatically generated

* With these 2 tools, let’s transform those crew members into individual rows:
* SELECT v:name::STRING AS pirate\_name

, c.VALUE:name::STRING AS crew\_name

, c.VALUE:nickname::STRING AS crew\_nickname

FROM pirate\_json, LATERAL FLATTEN(v:crew) c;

A close up of a list

Description automatically generated

* In this example, **c is the output inline view of the FLATTEN function, which is then joined to the columns selected from the table**
* We **use the dot operator to reference its VALUE property, which contains a VARIANT data type (where a colon is used to extract contents)**
* Note that NULL values are returned as empty strings
* **Using the same technique, we can handle *multiple* nested arrays**
* For example, what weapons did each of Blackbeard’s crew mates employ?
* SELECT v:name::STRING AS pirate\_name

, c.VALUE:name::STRING AS crew\_name

, w.VALUE::STRING AS crew\_weapons

FROM pirate\_json, LATERAL FLATTEN(v:crew) c

,LATERAL FLATTEN(c.VALUE:weapons) w;

* A close up of a computer screen

  Description automatically generated
* Remember that **working w/ information *inferred* from a semi-structured schema is like running standard SQL on relational tables**
* For example, if we wish to know how many different weapons Israel Hands employed, we can turn to **familiar SQL filters + aggregates**:
* SELECT COUNT(crew\_weapons) AS num\_weapons FROM (

SELECT c.VALUE:name::STRING AS crew\_name

, w.VALUE::STRING AS crew\_weapons

FROM pirate\_json, LATERAL FLATTEN(v:crew) c

,LATERAL FLATTEN(c. VALUE:weapons) w

WHERE crew\_name = 'Israel Hands'

);

A black text on a white background

Description automatically generated

* As these examples demonstrate, **Snowflake makes schema-on-read easy thanks to simple SQL extensions + the VARIANT** **data type, which is optimized for semi-structured data**
* ***However*, inferring a schema + transforming semi-structured data into cleanly formatted rows and columns takes time**
* **In more advanced cases, inferring schema-on-read requires input + guidance from the business users who specialize in it, just as in logical and conceptual modeling**
* Remember that **schema-on-read only *delays* the work of relational modeling, which *must* be done eventually to allow the organization to query it through traditional methods**
* The next section will examine a simple method for converting semi-structured data into structured, relational schema

#### Converting Semi-Structured Data into Relational Data

* As we saw in the previous exercise, **semi-structured data is flexible + can accommodate any amount of densely or sparsely nested elements**
* However, **in nested objects, it can be inferred that lower-level elements are *attributes* of their immediate parents**
* Observe the following simplified example of semi-structured data w/ 3 levels of nesting + use the indentation to count the depth

A screenshot of a computer program

Description automatically generated

* Here, we can see that a (*ship) type*, of depth 2, is an attribute of ship, which is a level-one attribute of the **root object**
* By this logic, if we follow the levels of a semi-structured object to its maximum depth N, those elements become attributes of an entity N - 1
* Then, N - 1 entities become attributes of N - 2, repeating recursively until arriving at the root
* In the current example, boarding axe and blunderbuss become instances in the weapon entity, which is an attribute of crew member, which is an attribute of pirate
* ***However*, there are several nuances to consider:**
* **Due to its *dynamic* nature, *any* object in a semi-structured file is free to vary + include whatever attributes or nested objects it needs**
* This means that, unlike in structured data, **looking at one instance of an entity**, such as crew member, **tells us *nothing* about the attributes of another**
* Notice how Stede Bonnet has a nickname attribute while Israel Hands has a Boolean parrot indicator
* **The only way to fully know ALL the levels + attributes of a semi-structured file is to scan it to the very end**
* **Even then, there is NO guarantee that new attributes won’t appear tomorrow**
* The good news is that **the process of determining the depth of a semi-structured file can be automated through Snowflake functions**
* Using the **FLATTEN function** covered earlier, we can **set the RECURSIVE parameter to automatically expand every element to its ultimate depth**
* The output of recursive flattening of an entire JSON file can be seen below:

A screenshot of a computer

Description automatically generated

* Notice that this output does NOT select from the VARIANT column v, but from the *result* of the lateral FLATTEN operation
* **Using the PATH column, we can calculate the depth by counting the number of dots and end-level array elements to give us a complete list of elements + their level in the semi-structured hierarchy**

A screenshot of a computer

Description automatically generated

* Now that we have the structure, we can **deconstruct the semi-structured data into normalized relational tables**
* **Starting w/ the *maximum* depth** (weapons at depth 3), we can **create a *dimension* w/ metadata columns + a surrogate key** (some elements, such as account numbers, contain natural keys, which can be used instead)
* **Start by creating a table for the weapon dimension, using a sequence to generate a surrogate key** (but a hash or a natural key can be used instead):
* CREATE OR REPLACE TABLE weapon

(

weapon\_id number(38,0) NOT NULL AUTOINCREMENT START 1 INCREMENT 1,

name varchar NOT NULL,

\_\_load\_name varchar NOT NULL,

\_\_load\_dts timestamp\_ntz NOT NULL,

CONSTRAINT pk\_weapon\_weapon\_id PRIMARY KEY ( weapon\_id ),

CONSTRAINT ak\_weapon\_name UNIQUE ( name )

)

COMMENT = 'weapons used by pirates'

;

* Now, **merge the weapon values from the JSON in the latest load + insert them if they don’t already exist**:
* MERGE INTO weapon w...;
* Have a look at the table contents:
* SELECT weapon\_id, name FROM weapon;

A screenshot of a computer code

Description automatically generated

* Now, **repeat the process for the elements at depth 2**
* The ship dimension poses no challenge because it has no child relationships
* However, some questions need to be answered before being able to model crew and years\_active**, which DO have child relationships**
* Is a crew member a *separate* entity from a pirate captain or is it a **subtype**?
* Do years\_active refer to how long a crew member has worked under the current captain, or how long they have been active over their entire pirating career?
* **Here, just as w/ modeling relational entities, only business experts can help determine the nuances of what the data represents + how it is used**
* Suppose our domain experts confirm that a crew member is a subtype of pirate + all attributes, including years\_active, are shared as part of a single pirate dimension
* This is the logical-to-physical rollup scenario described in Chapter 11
* **As part of the pirate rollup, we must make sure to include an FK reference to itself to store the relationship between the crew and captain**
* First, we create the table structure:

A screenshot of a computer

Description automatically generated

* As we have **fused a level-2 entity w/ its supertype**, we **must now load the data starting with the top-level dimension** (as crew will need an existing pirate\_id to report to)
* First, load the level-1 pirate entities (i.e., Blackbeard):
* MERGE INTO pirate p USING (< select level 1 pirates>)...;
* Once the top-level object has been loaded, we can load the crew details, referencing their captain’s surrogate key in the crew\_of column, + then verify the results
* MERGE INTO pirate p USING ( < select level 2 crew > )...;
* The output will look something like this:

A screenshot of a computer

Description automatically generated

* Look carefully + you will notice that one of the attributes, years\_active, is *missing* from the table
* **Because this data is multi-valued, it cannot be included in the pirate dimension without violating 1NF**
* For this, we **must create + load a separate** pirate\_years\_active **entity**
* The last missing attribute is pirate *weapons*
* In this scenario, we have a **many-to-many relationship** between the pirate and weapon dimensions
* As described in Chapter 8(Logical Modeling), **modeling many-to-many relationships requires an associative table in the physical layer**
* This table **holds the PKs for the associated entities** (i.e., pirate and weapon), **as well as the metadata fields that tell us when a certain relationship was first loaded**
* **Follow** the steps in the accompanying code to load the pirate\_weapons and pirate\_years\_ active tables
* W/ the associative table + the separate multi-valued entity added, the relational model is complete, as seen in the following diagram:

A screenshot of a computer

Description automatically generated

* Once loaded, we are free to analyze the data using traditional relational methods, for example:
* SELECT

p.NAME AS pirate\_name

, nvl(p.nickname, 'none') AS nickname

, s.type AS ship\_type

, nvl(w.NAME , 'none') AS weapon\_name

FROM pirate p

INNER JOIN ship s USING (ship\_id)

LEFT JOIN pirate\_weapons pw USING (pirate\_id)

LEFT JOIN weapon w USING (weapon\_id)

;

A close-up of a computer screen

Description automatically generated

* This exercise demonstrates that **although Snowflake features ease the technical burden of schema-on-read, the process is not trivial**
* **The fact that Snowflake can easily handle semi-structured data does NOT guarantee that your organization will have the functional or technical capacity to make sense of it**
* However, **by using a repeatable step-by-step process to transform semi-structured data into relational data, Snowflake users benefit from better organization + data consistency**

#### Summary

* **W/ the rising popularity of web apps + IoT data, semi-structured data has gained prominence for its flexibility in creating + loading dynamically changing objects w/out affecting ELT pipelines**
* Semi-structured formats, such as JSON, **can handle any amount of variable nested data, which doesn’t need to conform to a pre-defined structure**
* **Snowflake makes working w/ semi-structured formats easy thanks to its VARIANT data type, which is optimized for storage + analytical queries using easy-to-learn extensions to ANSI-standard SQL**
* **Querying a VARIANT data type provides the same performance as standard relational data types *w/out needing to analyze the structure ahead of time* in an approach known as schema-on-read**
* This means Snowflake users can **work w/ semi-structured + relational data on the same platform using familiar SQL commands**
* *However*, although Snowflake gives users all the tools necessary for analyzing semi-structured data, **schema-on-read only *delays* but does NOT eliminate the need for relational-style modeling**
* **Once the elements of semi-structured data have been understood, they can be converted into a relational schema by looking at element depth to determine its attributes and relationships**
* As w/ traditional modeling, **engaging the business teams for their expertise will be required to create a normalized schema that meets an organization’s analytical needs**
* Having familiarized ourselves w/ Snowflake’s semi-structured features, we can use many of the same techniques to handle another kind of variable structure: the **hierarchy**