snowflake®

DATA MODELING IN THE AGE OF JSON

How to Convert Schema-on-Read into Schema-on-Write

Author: Kent Graziano

CHAMPION GUIDES

TABLE OF CONTENTS

INTRODUCTION

Over the past decade, the number and variety of data sources has grown exponentially. With the rise of mobile devices and applications, as well as the advent of the Internet of Things, we now have data coming in from many new sources in addition to the data generated by our enterprise applications (for example, enterprise resource planning [ERP] applications), customer relationship management (CRM) systems, and weblogs. In fact, IDC recently estimated there will 175 zettabytes of data created by 2025.[1]

While big data technology has helped address how we will store this massive increase in data volume, an important question remains: What are we going to do with the data? How are we going to understand it, extract value from it, and take advantage of the insights that lie within it?

Effective data modeling is key to the successful use of these massive new data streams and formats. Far from being an outmoded skill, supplanted by the rise of automation tools, data modeling is more important than ever—especially for those tasked with business intelligence or analytics. Data modeling helps define the structure and semantics of the data and make it understandable, so business users and data scientists can be properly query, manipulate, and analyze it.

In the pages that follow, we'll examine the ways data is stored in this new era, particularly the Schema-on-Read and Schema-on-Write methods. We'll also look at new data formats such as JSON (JavaScript Object Notation) and show how you can transform them into a standard relational data model that can generate business value.

After all, in order for data to be useful, it must first be understood.

[1] IDC. The Digitization of the World From Edge to Core.

SCHEMA-ON-WRITE VS. SCHEMA-ON-READ

Since the inception of relational databases in the 1970s, the standard method for loading data into a database to be stored or analyzed was Schema-on-Write. In this method, a data modeler or database designer creates a structure, or "schema," for the data before it is loaded, or "written," into the system. The downside to this approach is that it can be time-consuming, requiring much upfront work before data can be loaded and analyzed or a business can extract value from it. However, when you *know* the structure of a data source (think: tables and columns in an ERP source system) and if you need to do little or no redesign to make the data ready for reporting and analytics, then this approach does not present any major problems.

Over the past decade, however, a new concept has developed, particularly in the "NoSQL" world: Schema-on-Read. The goal in this new method is to load data into the system as quickly as possible without all the upfront design and modeling for the database, thus shortening the time required to create an application for inputting the data. In an effort to be more agile, application developers gravitated to Schema-on-Read to deliver working software faster. Schema-on-Read allows applications to store data in semi-structured formats such as JSON and allows applications to be rapidly iterated without breaking the database behind the applications.

Although this was a benefit for the application developers of the world, it presented a hurdle to those who wanted to report off the data or use it for analytics—until now.

Still, to make data understandable to business users, this semi-structured data must be translated into a more understandable form. In database terms, that means we need to have a data model, or schema. This is especially true when it comes to conducting business intelligence or analytics functions that span many data streams or involves a variety of semi-structured and structured data sources.

Although many have equated Schema-on-Read with unstructured data and no data model, that actually is not correct. The term includes the word "schema," which most database engineers would interpret to mean a model or design. So Schema-on-Read does not indicate there is no model for the data (that is, that it is *unstructured*). Schema-on-Read means there is a flexible model and that the model need not be discerned until the data needs to be read.

THE RISE OF JSON DATA

With the rise of the internet and its myriad new web-based applications came the advent of XML (Extensible Markup Language), a flexible way of describing data and sharing structured data via the public internet or via private networks. Although XML remained the standard of choice for nearly a decade, as data became more complex and voluminous, some of XML's limitations became apparent.

As a language, many developers found it unnecessarily verbose and complex. Mapping XML to typed systems of programming languages or databases could also be difficult, especially when the data structure was tightly coupled to one application. Many developers also felt there were too many "tags," which took up too many characters and resulted in slower response times. Ironically, while XML came to the forefront with the rise of the internet, it had become too "heavy" for web speeds.

Enter JSON, which was introduced as an alternative to XML and was designed as a minimal, readable format for structuring data. The prevalence of JSON is directly connected to the nearly ubiquitous presence of JavaScript applications throughout the internet.

JSON is primarily used to transmit data between a server and a web application. As one CIO.com headline declared in 2016, "XML is toast; long live JSON." JSON has two additional benefits for users: It is easily readable (and understandable) to a human and it is programming language-independent, making it widely applicable for many use cases. JavaScript is a leading language for mobile, web-based, and IOT applications, so these applications tend to output JSON. Social media data, such as the hashtag and tweet data available from the Twitter API, is similarly output as JSON.

Although people have described JSON data as unstructured, because it isn't represented by traditional tables and columns, this description is inaccurate. Rather, JSON data is semi-structured, and you can transform into more traditional relational data model types for effective use in a Schema-on-Write system. Once modeled as such, JSON data can be integrated with a business's more traditionally structured data (for example, CRM data), providing additional richness and insight for analytics.

JavaScript Object Notation (JSON)

A minimal, readable format for structuring data, primarily used to transmit data between a server and a web application, as an alternative to XML.

TRANSFORMING JSON DATA INTO TRADITIONAL RELATIONAL MODELS

You can transform JSON data into any style of relational model, but for the purposes of this ebook, we will focus on two main types: 3NF (third normal form) and Data Vault. 3NF is an architectural standard designed to reduce the duplication of data and ensure the referential integrity of a database, whereas Data Vault models were developed specifically for data warehouse design to address agility, flexibility, and scalability issues found in other approaches.

A closer look at how to transform JSON data into each of these models reveals that not only is the process achievable, but that the JSON data is itself really does have a structure with meaning.

Below is an example of a simple JSON document, describing a list of colors. The JSON document is enclosed by curly brackets. (***Figure 1***)

Most JSON documents are self-describing. As you can see in Figure 2, the JSON document contains a number of attributes that are essentially Key Value pairs (for example, color: white, category: hue, type: primary). In database terms, we can think of the key—everything in double quotes on the left side of a colon in this list—as a column or column name. The values are everything in double quotes to the right side of a colon within the list, and they would typically be stored in the database as the value for that key. (***Figure 2***)

JSON documents also have the ability to present nested keys, delineated here by more curly brackets, and arrays, delineated by square brackets. The example in Figure 3 contains both. Here, the key code consists of two value parts, rgba and hex, and an array of values is assigned to the rgba key, namely [0, 255, 0, 1]. (***Figure 3***)

Figure 1 — Simple JSON document

```
{
   "colors": [
            {
                        "color": ”white",
                        "category": "hue",
                        "type": "primary",
                        "code": {"rgba": [255,255,255,1],
                                    "hex": "#FFFFF”
                                    }
            },
            {
                        "color": "green",
                        "category": "hue",
                        "type": "secondary”,
                        "code": {  "rgba": [0,255,0,1],
                                    "hex": "#0F0"
                                    }
             },
            ]
}
```

Figure 2 — Example of Key Value pairs

```
{
   "colors": [
            {          Key :  Value
                        "color": ”white",
                        "category": "hue",
                        "type": "primary",
                        "code": {"rgba": [255,255,255,1],
                                    "hex": "#FFFFF”
                                    }
            },
            {
                        "color": "green",
                        "category": "hue",
                        "type": "secondary”,
                        "code": {  "rgba": [0,255,0,1],
                                    "hex": "#0F0"
                                    }
             },
            ]
}
```

Figure 3 — Example of nested keys and arrays

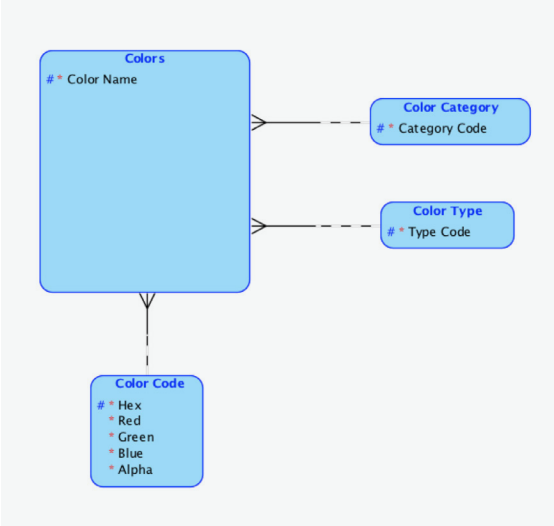
```
{
   "colors": [
            {          Key :  Value
                        "color": ”white",
                        "category": "hue",
                        "type": "primary",
                        "code": {"rgba": [255,255,255,1],
                                    "hex": "#FFFFF”
                                    }
            },
            {
                        "color": "green",
                        "category": "hue",
                        "type": "secondary”,
                        "code": {  "rgba": [0,255,0,1],
                                    "hex": "#0F0"
                                    }
             },
            ]
}
```

TURNING JSON INTO 3NF

Now that you understand the elements of the JSON document, we can transform it into a 3NF model. Because this document is about **color**, we define color as the primary entity, and we define **category**, **type**, and **code** keys as lookup, or reference, entities.

The nested key **code** requires one more step. As it turns out, the term "rgba" refers to "red, green, blue, alpha," and the value for hex is a hexadecimal number that defines web-based colors. Because the **rgba** values comprise an array, we must first separate them into their component parts and turn them into attributes for the reference entity **color code**.

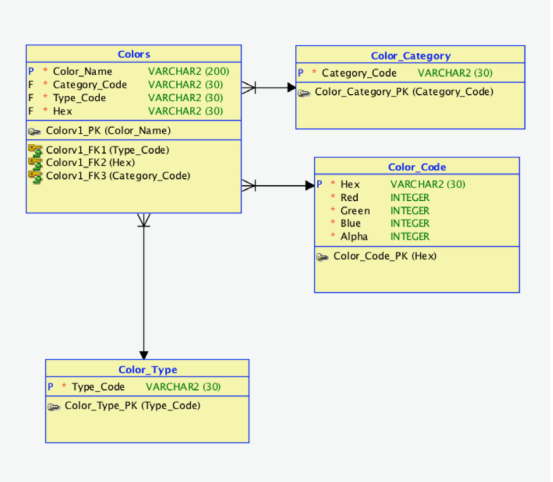
Here is a logical representation of the transformation so far:



Note: The diagram above represents a logical diagram or entity relationship diagram (ERD) using the Barker Notation popularized in the Oracle*CASE product in the early 1990s and still used in the Oracle SQL Developer Data Modeler tool.

Now we are ready to turn this logical representation into a 3NF schema model of tables. The **colors** entity becomes a **Colors** table with the primary key (denoted by **P**) of **Color_Name**. We also have **Color_Category, Color_Type**, and **Color_Code** tables with the inherited relationships from the logical diagram above. For **Color_Category** and **Color_Type**, we have tables with primary keys of **Category_Code** and **Type_Code**" respectively.

For the **Color_Code** table, we have a table with the primary key **Hex** as well as **Red,Green,Blue**, and **Alpha** as the mandatory attributes.



Through this short process, we have turned the structure of this JSON document into tables and columns as a 3NF data model diagram. Report and application developers who are tasked with building user friendly analytics systems can now use and understand this model.

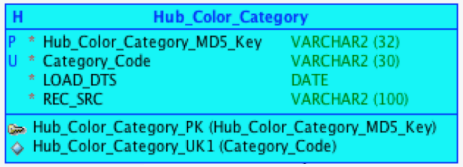
TURNING JSON INTO A DATA VAULT MODEL

Data Vault models encompass three basic structures:

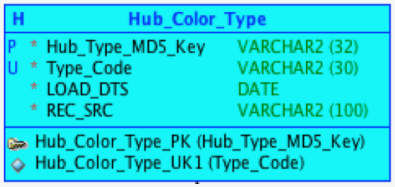
- **Hubs**, which have a unique list of business keys
- **Links**, which document a unique list of relationships across keys
- **Satellites**, which house descriptive data, including changes over time

It is important to note that in a Data Vault model, hubs cannot be "child" tables, and satellites have one and only one parent table and cannot be "parents" to other tables.[2]

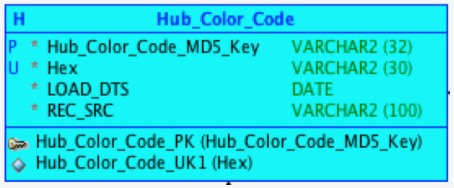
First, we define **`Hub_Color_Category`** as a hub. In the Data Vault 2.0 method, all hubs have a primary unique identifier, denoted by P" which is a calculated key. The hub also has a natural unique identifier (U), or business key (BK), in this case **`Category_Code`**. Every table in a Data Vault design also carries two pieces of metadata with it—the load date and the record source—making this the first hub defining the JSON document in a Data Vault manner.



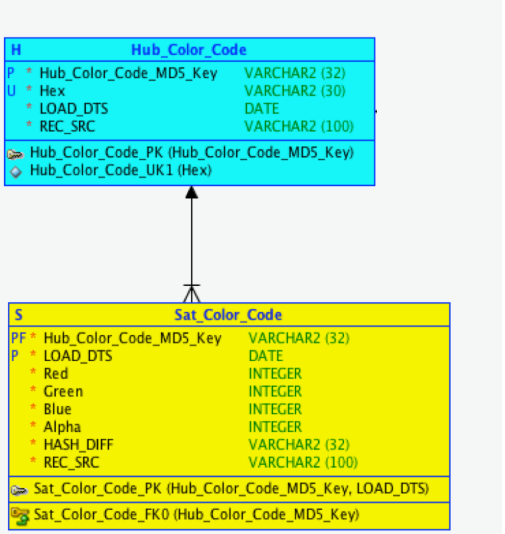
The next hub is **`Hub_Color_Type`**. Again, the primary key is **`Hub_Type_MDS_Key`** and **`Type_Code`** is the unique identifier, along with the load date and record source.



The third hub is **`Hub_Color_Code`**, with Hex as the unique identifier, along with the load date and record source.



One of the key differences between Data Vault models and other models is that the attributes that are related or potentially changeable over time reside in satellite structures. The satellite hanging off **`Hub_Color_Code`** is called **`Sat_Color_Code`** and houses the **`Red, Green, Blue`**, and **`Alpha`** attributes.



Within this satellite, you can see that the primary key not only includes the inherited key from the related hub, but also the load date. This is helpful for tracking changes to the data over time. When changes are detected in the inbound data to a Data Vault style of warehouse, those changes are loaded into the satellite, which is where change data capture occurs.

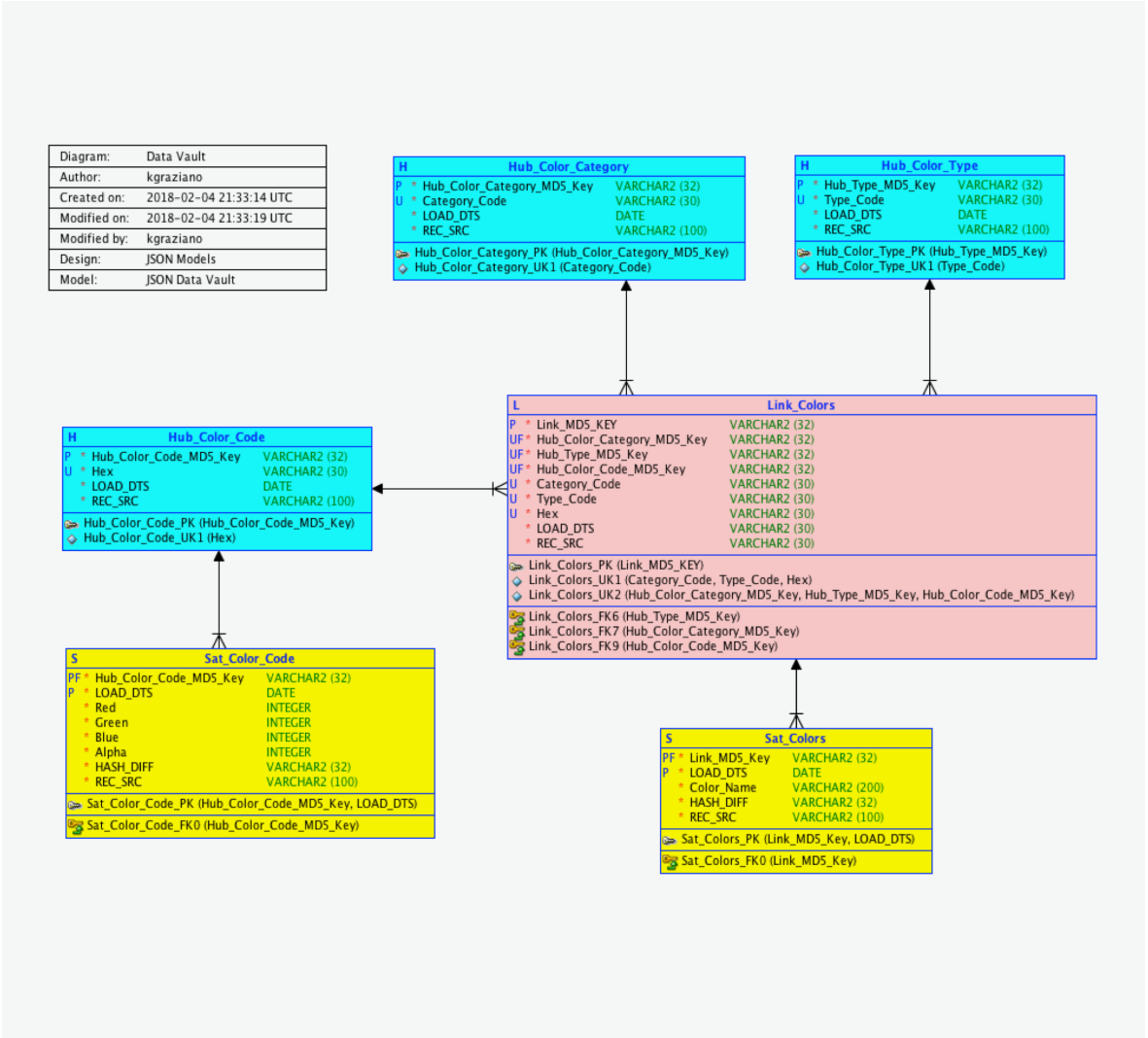
[2] Kent Graziano, Better Data Modeling: An Introduction to Agile Data Engineering Using Data Vault 2.0.

The **`HASH_DIFF`** column in the satellite table is an optional component of the Data Vault model that is used for change detection. All the non-key columns are "hashed" together to get a unique string. If any of the components change, a different string is generated. Comparing the new string to the existing string makes change detection simpler and faster when you load data to the Data Vault.

Finally, the relationship between all of these hubs is expressed as a link (**`Link_Colors`**), with a satellite (**`Sat_Colors`**) hanging off of it. *(See diagram to the right.)*

The ultimate definition of color as defined by the JSON document is essentially an intersection of a color code, a color category, and a color type. In this link structure, you see a calculated **`Link_MDS_KEY`**, followed by a number of unique identifier components, which are the inherited keys from each of the hubs, and, of course, the mandatory metadata columns for the load date and record source. We also denormalized the business keys from those hubs to help optimize queries against the link when all we need to know is the business keys.

This link structure is effectively a many-to-many intersection entity, which is how relationships are recorded in a Data Vault model. This allows you to load data into the data warehouse regardless of its cardinality, which is particularly useful when what was a static relationship between data sets later changes its cardinality. The Data Vault model absorbs those changes without changing the structure of the data model itself.



HANDLING CHANGES IN SCHEMA-ON-READ

Source data structures and feeds can change over time. This often results in broken data ingestion processes when the incoming data structure does not match the target database schema. One of the benefits of Schema-on-Read is the ability to absorb changes in the data contents without having to change the database schema.

If you are using a modern cloud-built data warehouse such as Snowflake, there are techniques to marry the concepts of Schema-on-Read to Schema-on-Write so that if an inbound JSON document changes, the ingestion process requires no changes. When you use Snowflake's VARIANT data type, for example, to ingest the raw JSON data, the data load won't break. This makes your load process more resilient so existing downstream reports and applications do not break.

However, to expose the new data in a revised JSON document, you need to adapt the relational view of that structure, which means potentially changing the data model. If you add a new attribute, such as a description of the color, to the JSON document, handling the new data is easily manageable:

- **In a 3NF model, simply add the new column and alter the relevant table (or view).**
- **In a Data Vault model, add the new column and alter the relevant table (and associated load process).**
- **Alternatively, save yourself the trouble of managing these changes and retesting, and simply create a new satellite table off an existing hub, allowing you to keep loading the data as it was, while expressing the new structure with a revised table structure.**

CONCLUSION

In a world of big data, data lakes, and data warehousing where both Schema-on-Read and Schema-on-Write exist, data modelers and data architects remain extremely important, particularly when it comes to business intelligence and data warehousing. As flexible and fast as Schema-on-Read is, it still requires a model of some kind in order to allow business users to make sense of the data. Semi-structured data such as JSON documents make application development more agile and are a powerful new source of data for businesses. However, in order to extract the greatest value from them, data warehouse architects must transform them into an understandable relational model that can be used for reporting and analytics.

You can transform semi-structured data into a relational model by applying good data modeling design practices such as the simple steps described in this ebook. Whether you choose a 3NF, star schema, or Data Vault model, the goal is the same: an actionable model that can be queried and understood and used to derive business value from an ever-increasing stream of data.

ABOUT THE AUTHOR

Kent Graziano is the Chief Technical Evangelist for Snowflake Inc. He is an award-winning author and recognized expert in the areas of data modeling, data architecture, and Agile data warehousing. He is an Oracle ACE Director - Alumni, a member of the OakTable Network, a certified Data Vault Master and Data Vault 2.0 Practitioner (CDVP2), and an expert data modeler and solution architect with more than 30 years of experience.

ABOUT SNOWFLAKE

Snowflake Cloud Data Platform shatters the barriers that prevent organizations from unleashing the true value from their data. Thousands of customers deploy Snowflake to advance their businesses beyond what was once possible by deriving all the insights from all their data by all their business users. Snowflake equips organizations with a single, integrated platform that offers the only data warehouse built for any cloud; instant, secure, and governed access to their entire network of data; and a core architecture to enable many other types of data workloads, including a single platform for developing modern data applications. Snowflake: Data without limits. Find out more at **snowflake.com.**