# Section 1: Overview

With the prevalence of IoT devices, social media sites and mobile devices all generating endless streams of JSON data, you’ve probably wondered what insights your organization can glean from this semi-structured data, along with traditional structured data you already have. Certainly, there is no shortage of various approaches and databases, offering options to work with your JSON data, but most are not geared towards programmers and technical professionals with specific skills and are not seamlessly integrated with your traditional data. One of the key differentiators in Snowflake, the “Data Warehouse Built for the Cloud”, is the ability to natively ingest semi-structured data such as JSON, store it efficiently and then access it quickly using simple extensions to standard SQL. In this tutorial, we will show some of these versatile capabilities in Snowflake.

## 1.1. What You’ll Learn

The exercises in this lab will walk you through the steps to:

* Configure your Snowflake workspace
* Access and explore JSON structures captured in Bitcoin blockchain sample data
* Query JSON data in Snowflake using versatile native functions
* Create objects within Snowflake

# Section 2: Overview of Sample Bitcoin Blockchain JSON Structure

For our sample JSON data, let’s look at some examples of Bitcoin Blockchain data. Detailed contents of the Bitcoin Blockchain JSON can be found at:

<https://blockchain.info/block-height/500613?format=json>



* The Bitcoin Blockchain consists of *blocks*.
* A block is a set *of transactions, tx,* and also contains some block-specific information. It is a many-to-many relationship as multiple blocks can include the same transaction.
* A transaction consists of *inputs* and *outputs*. The input references outputs from prior transactions, which may include transactions in the same block. The same transaction can be included in more than one block. This is common during chain splits, i.e. when more than one miner solves a block.
* A transaction’s input value is the sum of its inputs and the output value is the sum of its outputs. The difference between the input and the output is the transaction *fee* and is taken by the miner solving the block in which the transaction is included.
* The first transaction in a block is referred to as the *coinbase*.
* In a Blockchain, blocks and transactions are always referred to through their *hash*.

# Section 3: Query Bitcoin Blockchain JSON data using Snowflake SQL

In Snowflake, as soon as the JSON data is loaded, we can easily issue SQL queries to gain insight into the data without transformation or pre-processing of the JSON.

This section shows some sample queries to easily analyze the JSON data using the various JSON functions and semi-structured data querying features including the LATERAL FLATTEN function in Snowflake.

This section will guide you through the steps that enable you to do the following:

* Start with some simpler examples to pull out some basic data from the JSON column
* Understand how to access simple nested fields
* Understand how to access more complex fields

## 3.1. Configure your Workspace

Now that we are familiar with the case, navigate to the Worksheets tab of your Snowflake instance and create and name a new worksheet. On the upper right-hand side of your worksheet, you will see your worksheet configuration settings. Using these settings, configure your environment variables by changing the warehouse and database to the location of the Bitcoin Blockchain data.

## 3.2. Start Pulling Basic Data Out

Now that we’ve set up our workspace, let’s work through some examples of how to access the JSON data. We’ll start by pulling out some core attributes of a block from some of the sub-columns:

Query the data to pull the block height, number of transactions in a block, block relayer, block hash, and block size from your BLOCK table.

Hint: Similar to the Table.Column notation all SQL people are familiar with, in Snowflake, we have the ability to effectively specify a column within the column––a sub-column. However, we cannot leverage “dot” as our separator, as SQL syntax has already claimed that. So, we use a colon to reference the JSON sub-columns and navigate that hierarchy. This structural information is dynamically derived based on the schema definition embedded in the JSON string. Snowflake’s advanced metadata engine records this information at the time it loads the JSON document into the table.

For further reference: <https://docs.snowflake.net/manuals/user-guide/json-basics-tutorial-query.html>

## 3.3. Using the CAST to Model the Right Data Type

We can cast a sub-column to an appropriate data type and give it a nicer column alias, similar to what we would do with a normal column

Using your query from above, use Snowflake’s CAST function to cast each column as a different data type.

For further reference: <https://docs.snowflake.net/manuals/sql-reference/functions/cast.html>

## 3.4. Flattening JSON Data

Given that JSON schemas are flexible, e.g. they can easily change, it’s essential to have a method to extract the data dynamically, and be able to determine how many rows are in the array for any given record, at any time. In this section, we will introduce the LATERAL FLATTEN function which takes an array and returns a row for each element in the array. You can select all the data in the array as though it were in table rows; there is no need to figure out how many elements there are.

Use the LATERAL\_FLATTEN function to flatten your data, then query from your flattened data.

For further reference: <https://docs.snowflake.net/manuals/user-guide/json-basics-tutorial-flatten.html>

<https://docs.snowflake.net/manuals/sql-reference/functions/flatten.html>

# Section 4: Summary

This tutorial explores how Snowflake enables business analysts to work with JSON data using SQL. In particular, it includes some sample real-world JSON data sets and explores a set of examples using Snowflake JSON functions and semi-structured data querying capabilities. Now you have some ideas for how to use Snowflake to load and query JSON data such as the Bitcoin Blockchain data set.

To expand your Snowflake skills and learn more, visit and work through examples included in the Snowflake product documentation. You can also query data and explore Snowflake’s SQL functions, asking questions to our team along the way.

APPENDIX

# Section 3: Query Bitcoin Blockchain JSON data using Snowflake SQL

In Snowflake, as soon as the JSON data is loaded, we can easily issue SQL queries to gain insight into the data without transformation or pre-processing of the JSON.

This section shows some sample queries to easily analyze the JSON data using the various JSON functions and semi-structured data querying features including the LATERAL FLATTEN function in Snowflake.

This section will guide you through the steps that enable you to do the following:

* Start with some simpler examples to pull out some basic data from the JSON column
* Understand how to access simple nested fields
* Understand how to access more complex fields

## 3.1. Configure your Workspace

Now that we are familiar with the case, navigate to the Worksheets tab of your Snowflake instance and create and name a new worksheet. On the upper right-hand side of your worksheet, you will see your worksheet configuration settings. Using these settings, configure your environment variables by changing the warehouse and database to the location of the Bitcoin Blockchain data.

## 3.2. Start Pulling Basic Data Out

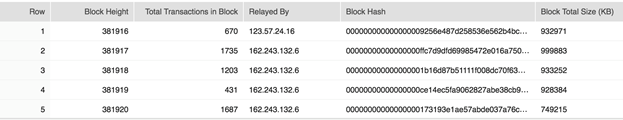
Now that we’ve set up our workspace, let’s work through some examples of how to access the JSON data. We’ll start by pulling out some core attributes of a block from some of the sub-columns:

Run the following query:

**SELECT block:height AS Height, block:n\_tx AS Transactions,**

**block:relayed\_by AS "Relayed By", block:hash AS "Hash",**

**block:size AS "Size (KB)" FROM blocks;**

Result:****

Similar to the Table.Column notation all SQL people are familiar with, in Snowflake, we have the ability to effectively specify a column within the column––a sub-column. However, we cannot leverage “dot” as our separator, as SQL syntax has already claimed that. So, we use a colon to reference the JSON sub-columns and navigate that hierarchy. This structural information is dynamically derived based on the schema definition embedded in the JSON string. Snowflake’s advanced metadata engine records this information at the time it loads the JSON document into the table.

## 3.3. Using the CAST to Model the Right Data Type

We can cast a sub-column to an appropriate data type and give it a nicer column alias, similar to what we would do with a normal column:

Run the following query:

**SELECT block:height::int AS Height,**

**block:n\_tx::int AS Transactions ,**

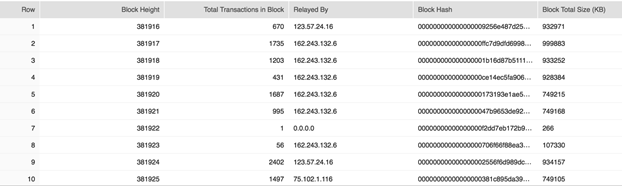
**block:relayed\_by::string AS "Relayed By",**

**block:hash::string AS "Hash",**

**block:size AS "Size (KB)"**

**FROM blocks;**

Result:



Again, simple SQL and the output are similar to the results from any table you might have built in a traditional data warehouse.

At this point, you could look at a table in Snowflake with a VARIANT column and quickly start “shredding” the JSON with SQL.

## 3.4. Flattening JSON Data

Given that JSON schemas are flexible, e.g. they can easily change, it’s essential to have a method to extract the data dynamically, and be able to determine how many rows are in the array for any given record, at any time. In this section, we will introduce the LATERAL FLATTEN function which takes an array and returns a row for each element in the array. You can select all the data in the array as though it were in table rows; there is no need to figure out how many elements there are. The following query shows this example:

Run the following query:

**SELECT txs.value:tx\_index AS "Transaction Index",**

**txs.value:weight AS "Transaction Weight",**

**txs.value:size AS "Transaction Size (KB)" FROM blocks,**

**LATERAL FLATTEN( INPUT => block:tx ) txs;**

Result:

