Fast, Efficient Processing of Semi-Structured Data

snowflake

Conventional data warehouses were designed decades ago, when data arrived in very predictable structured formats. Relational data with fixed schemas was the norm because data sources were limited, controlled and changed infrequently.

Today, data arrives in diverse forms from diverse sources. The rapid decrease in the cost of storing data and the growth in distributed systems has led to an explosion of machine-generated data. This includes data from applications, sensors, mobile devices, and more.

Semi-structured data formats such as JSON, Avro, and others have become the de facto form in which this data is sent and stored. Semi-structured data is easy for these applications to create and capable of representing a wide array of information.

Two of the key attributes that distinguish semi-structured data from structured data are the lack of a fixed schema and nested data structures. Whereas structured data requires a fixed schema defined in advance, semi-structured data does not require a prior definition of a schema and can constantly evolve—new attributes can be added at any time. Also unlike structured data, which represents data in a flat table, semi-structured data can contain hierarchies of nested information.

The flexibility of schemaless design and the ability to represent a wide range of information are key reasons that semi-structured data has become so widely used. New and richer information can easily be added to the data at any time.

Snowflake handles semi-structured data as a first-class database element:

* ***Flexible-schema datatype:** load semi-structured data without transformation*
* ***Storage optimization:** transparently converted to optimized internal storage format*
* ***Query optimization:** automatic database optimizations for fast and efficient SQL querying*

Semi-Structured Data in a Relational World

However, the flexibility and expressiveness of schemaless design create challenges when the data needs to be analyzed. As the use of semi-structured data formats has increased, so has the need to analyze that data. Although a limited amount of analysis can be done on semi-structured data in isolation, the most valuable insights come from bringing semi-structured data together with other data, particularly structured relational data.

Relational databases were not designed to store and process semi-structured data. They were architected based on the assumption that a static schema could be determined in advance. That design assumption has made possible a wide array of optimizations—pruning, predicate push-down, and others—but at the cost of sacrificing the flexibility that schema-on-read offers.

To date, data warehouses have supported two options, both of which have significant drawbacks. The first option is to transform semi-structured data into a fixed schema before loading it into the data warehouse. This can be done by transforming the data in another system (e.g. an ETL tool, custom scripting, or a Hadoop system) or by extracting specific attributes from the semi-structured data at the time of loading. This approach creates a very fragile data pipeline that requires significant maintenance. Every change in the data—adding a new attribute, eliminating an attribute or adding a new level of nested information—breaks the data pipeline such that information is lost until the transformation and attribute extraction is updated to handle the new data structure.

The second option is to store semi-structured data as an uninterpreted object inside a relational table, either simply as a string or as a datatype that stores the object as a BLOB or CLOB. That approach simplifies loading of semi-structured data, but sacrifices performance. Because the semi-structured data is effectively a black box to the database engine, access to any part of that data requires a complete scan of the entire object. Even systems that support some type of indexing on these objects are only a small improvement because they can index at best a very limited number of attributes and add significant overhead for creating, storing, and maintaining those indexes.

To avoid the complexity of these options, another approach has been to use a noSQL data platform to store and analyze semi-structured data. Systems such as Hadoop can adapt to the rapid evolution in semi-structured data because they can store that data without requiring definition of a fixed schema. However they were not designed for high-performance querying, particularly querying that combines semi-structured and structured data, and come with significant complexity of their own. Attempts to extend noSQL data platforms to support relational SQL querying have come to face the reality that their core processing engines were simply not designed to provide the optimized performance and broad SQL support that is provided by a relational database.

As a result, it has been a struggle to bring together structured and semi-structured data and make that data accessible for analysis in a timely fashion.

Semi-Structured Data as a First-Class Database Citizen

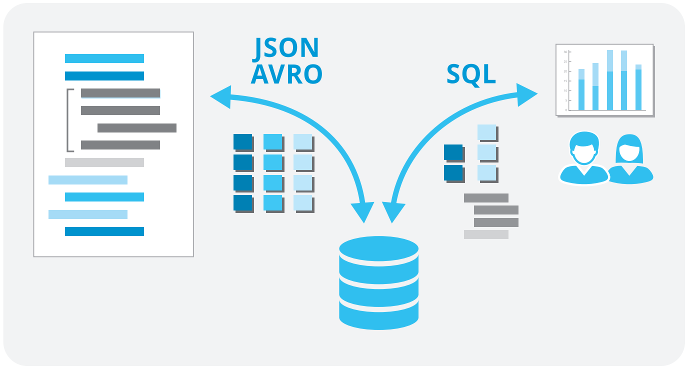
When we architected the Snowflake data warehousing service, we designed it from the start to handle semi-structured data without the trade-offs of current approaches. Our patent-pending approach makes it possible to have both the schema flexibility of semi-structured data and the performance optimizations of relational data. To do that, we designed our database engine to handle both semi-structured data natively without transformation.

We started by making it possible to load semi-structured data in its native form, without first needing to flatten, transform, or extract attributes to convert it into a relational form. We did this by creating a new datatype (called VARIANT) that can handle a variety of semi-structured data types including JSON and Avro.

Unlike datatypes for semi-structured data in conventional databases, the VARIANT data type transparently interprets and stores semi-structured data efficiently. As semi-structured data is loaded, Snowflake automatically examines each section for repeated data attributes. Based on an assessment of potential performance improvement, Snowflake breaks out and stores repeated attributes independently, similar to the way that a columnar database stores individual columns independently. Snowflake makes this assessment independently for each section of the data in order to make these optimizations even when data attributes are present in some records but not others.

All of these optimizations occur "under the hood" such that, to users and queries, the data in Snowflake remains in semi-structured form. Snowflake provides query operators that allow SQL statements to reach into semi-structured data to access individual attributes, including nested attributes and arrays. This makes it possible for a single query to access and combine both structured and semi-structured data in all of the ways supported by SQL.

Snowflake also supports the creation of relational views on top of semi-structured data such that users and tools who need to see data in strict relational form, or who are unaware of Snowflake's SQL extensions, can access the data in a form that they already understand, but without requiring a data pipeline that first transforms that data into a new relational table.



Snowflake transparently optimizes semi-structured data for fast querying with SQL.

Another part of Snowflake's patent-pending design is a database engine that natively understands how to optimize performance for queries on semi-structured data in the same way it optimizes queries on structured data, without requiring the user to transform data, create and manage indexes, or tune settings. As semi-structured data is loaded and stored in optimized form inside Snowflake, Snowflake records metadata about the structure of the data and how it has been stored. This metadata, which Snowflake automatically updates even as data changes, makes it possible for Snowflake to optimize query plans and query execution—filtering, pruning, and other database optimizations can be applied to VARIANT data in the same way that they are applied to relational data. This delivers significant performance enhancements for queries on semi-structured data, without manual tuning and optimization.

Using Semi-Structured Data in Snowflake

In the simplest scenario, all that is needed to load semi-structured data is to create a table with a single column of type VARIANT and then execute Snowflake's COPY command to load data from one or more files containing the semi-structured data. It is also possible to create tables with multiple columns that are a mix of standard data types and VARIANT data types, as well as to use commands such as INSERT and CREATE TABLE AS SELECT to populate tables that contain VARIANT columns.

To illustrate, imagine that you have a data set of orders stored in JSON format. A typical record could have the following form:

```
{
    “id”: 920384503849,
    “customer_id”: 92923,
    “name”: “online order”,
    “date_utc”: 1417203819,
    “order_details”: {
        “name”: “order”,
        “type”: “summary”,
        “record_id”: 122994,
        “items”: [
            {
                “id”: 5492,
                “name”: “jeans”,
                “model_id”: 1221,
                “brand_id”: 12,
                “price”: 123.12,
                “quantity”: 10,
                “color”: “blue”
            },
            {
                “id”: 122,
                “name”: “polo shirt”,
                “model_id”: 23,
                “brand_id”: 1,
                “price”: 34.95,
                “quantity”: 1,
                “color”: “red”
            },
        ]
    }
}
```

To work with this data in Snowflake, you would first define a table that includes a column of type VARIANT to use for storing this data. That table can be as simple as a table with a single VARIANT column, or could also have other columns of any type as well. For this example, we will use a table named “ORDERS” with a single VARIANT column named “DETAIL”:

ORDERS

DETAIL
{ order1 }
{ order2 }
{ order3 }
…

Data files in JSON format containing the order information can be loaded in parallel in a single step using Snowflake's bulk COPY command. No data preparation, transformation, schema definition, or attribute extraction is required to load the data.

Accessing Individual Attributes

Once semi-structured data has been loaded into Snowflake, SQL extensions allow access to attributes within the data. Standard dot notation paths are used to specify attributes within a semi-structured data object, separated from the relational path by a single colon (':').

For example, the top-level “id” attribute in the record shown above would be identified as `ORDERS.DETAIL:id` and the “record_id” attribute nested within “order_details” would be identified as `ORDERS.DETAIL:order_details.record_id`.

Arrays can also be accessed in a similar way using square bracket ('`[]`') notation to identify individual entries within an array. For example, the “id” attribute in the first element of the “items” array would be accessed as `ORDERS.DETAIL:order_details.items[0].id`.

Because many semi-structured data formats are not explicitly typed, Snowflake supports explicit casting in two ways: using the CAST() function, or using a trailing double-colon ('`::`') operator. For example, to cast the attribute “record_id” to an integer, a query would specify `ORDERS.DETAIL:order_details.record_id::integer`.

Accessing Repeated Attributes

Snowflake also provides SQL extensions that make it possible to transform repeated elements from an array into repeated records (rows). In Snowflake, we use the table function FLATTEN() for this purpose. FLATTEN() takes an array as input and returns one row per array element.

In this basic example, we use FLATTEN() to transpose the elements an array of numbers into individual rows:

```
WITH t1 AS (
  SELECT parse_json(column1) AS c1
  FROM VALUES (‘[1,2,3,4]’)
)
SELECT t2.value AS flattened_column
FROM t1, TABLE(FLATTEN(t1.c1)) t2;

------------------+
 FLATTENED_COLUMN |
------------------+
 1                |
 2                |
 3                |
 4                |
------------------+
4 rows in result
```

We can use the FLATTEN() table function, passing in the array of items, to calculate the total amount a given customer spent with the following query:

```
SELECT SUM(
  order_items.value:price::decimal(5,2) *
order_items.value:quantity::int
  ) AS “total spend”
FROM orders,
  TABLE(FLATTEN(orders.detail:order_
details.items)) order_items
WHERE orders.detail:customer_id::int =
92923;
```

CONCLUSION

Snowflake’s architecture makes it possible to query semi-structured data and structured data together using SQL. You can join, window, compare and calculate structured and semi-structured data in a single query. This makes it possible to eliminate extra systems and steps while realizing superior performance, simplifying data pipelines and reducing the time from when data is generated to when it can be accessed and analyzed.

About Snowflake

Snowflake Computing, the cloud data warehousing company, has reinvented the data warehouse for the cloud and today’s data. The Snowflake Elastic Data Warehouse is built from the cloud up with a patent-pending new architecture that delivers the power of data warehousing, the flexibility of big data platforms and the elasticity of the cloud – at a fraction of the cost of traditional solutions. The company is backed by leading investors including Altimeter Capital, Redpoint Ventures, Sutter Hill Ventures and Wing Ventures. Snowflake is headquartered in Silicon Valley and can be found online at snowflake.net.

snowflake

www.snowflake.net | @SnowflakeDB