Looker Blog

SQL Processing and Data Analysis with the EAV Model

Mike Xu, December 9th, 2014

Much of the world's data is stored in an Entity Attribute Value (EAV) model. The EAV model is a key value pattern commonly used in scientific research, medicine, healthcare, and popular open source and commercial software platforms such as Magento and Drupal. The key advantage of the EAV Model is that traditional ER schemas cannot handle extreme sparseness and volatility of data. It was designed with the intent of getting data in, with the tradeoff of being difficult to get data back out. To overcome these tradeoffs, this article will cover how to effectively analyze EAV data.

If you are not fully familiar with the structure and pros/cons of the EAV model or need a quick refresher Dr. Prakash Nadkarni has a great intro to it. In order to fully work with EAV data for traditional analysis and reporting, we will transform the EAV tables to an ER model by creating tables for their associated entities.

Directly querying EAV data to analyze cohorts, funnels, and time series is tedious and challenging. These queries are often complex and quickly become unmaintainable. The best long-term solution is to flatten the EAV dataset out into a traditional ER model. This form of data is ultimately the most flexible and intuitive. After selecting a pattern for flattening, we also will need to figure out the best method to persist and update the flattened resultsets.

Flattening the Model

There are three techniques to turning the attribute tables into a single row for each entity. The first two techniques require you to know what attributes are wanted and the last technique will allow for a "look ahead" of all possible attributes and create a table with all attributes flattened.

Lets first have an example EAV table with a single entity (patient) returned:

```
SELECT * FROM patients WHERE patient_id = 1
```

TABLE:patients		
patient_id	attribute	value
1	first_name	Betty
1	last_name	Grof
1	gender	female
1	race	human

The first flattening method is through correlated subqueries, here is an example:

```
SELECT
(SELECT value
    FROM patients
    WHERE patient_id = 1
    AND attribute = "first_name"
) as first_name
, (SELECT value
    FROM patients
    WHERE patient_id = 1
    AND attribute = "last_name"
) as last_name
, (SELECT value
    FROM patients
    WHERE patient_id = 1
    AND attribute = "gender"
) as gender
, (SELECT value
    FROM patients
    WHERE patient_id = 1
    AND attribute = “race”
) as race
```

This will return:

TABLE:patients				
patient_id	first_name	last_name	gender	race
1	Betty	Grof	female	human

Ideally, we would flatten every single entity in the EAV table in one query, one at a time is not sufficient. Let’s use a more complex dataset:

```
SELECT * FROM patients
```

TABLE:patients		
patient_id	attribute	value
1	first_name	Betty
1	last_name	Grof
1	gender	female
1	race	human
2	first_name	Simon
2	last_name	Petrikov
2	gender	male
2	race	human

Now we’re going to create a CTE first with just the distinct entities for our next query (with MySQL you will want to use a TEMPORARY TABLE which performs the same functionality):

```
WITH patients_entity AS (
SELECT
patient_id
FROM patients
GROUP BY patient_id)
SELECT
patients_entity.patient_id
,(SELECT value
    FROM patients
    WHERE patients.patient_id = patients_entity.patient_id
    AND attribute = “first_name”
) as first_name
, (SELECT value
    FROM patients
    WHERE patients.patient_id = patients_entity.patient_id
    AND attribute = “last_name”
) as last_name
, (SELECT value
    FROM patients
    WHERE patients.patient_id = patients_entity.patient_id
    AND attribute = “gender”
) as gender
, (SELECT value
    FROM patients
    WHERE patients.patient_id = patients_entity.patient_id
    AND attribute = “race”
FROM patients_entity
)
```

The CTE gives us a list of all the unique entities (patients) present in the EAV table. For performance reasons this would ideally be materialized back into the database, but it’s also fine to just compute this at run time.

TABLE:patients_entity
patient_id
1
2

By using the entity CTE for the outer SELECT, we can subquery into the EAV table multiple times for each of our attributes while matching the inner entity id to the outer entity id. Here are our results:

TABLE:patients				
patient_id	first_name	last_name	gender	race
1	Betty	Grof	femaler	human
2	Simon	Petrikov	human	male

The second is through JOINs. We will use the exact same sample data from above to produce the same results. Here is the query:

```
WITH patients_entity AS (
SELECT
patient_id
FROM patients
GROUP BY patient_id)
SELECT
patients_entity.patient_id
, p1.value AS first_name
, p2.value AS last_name
, p3.value AS gender
, p4.value AS race
FROM patients_entity
JOIN patients AS p1
  ON patients_entity.patient_id = p1.patient_id AND p1.attribute = “first_name
JOIN patients AS p2
  ON patients_entity.patient_id = p2.patient_id AND p2.attribute = “last_name”
JOIN patients AS p3
  ON patients_entity.patient_id = p3.patient_id AND p3.attribute = “gender”
JOIN patients AS p4
  ON patients_entity.patient_id = p4.patient_id AND p4.attribute = “race”
```

The third technique is through your database’s procedural language. For Oracle it is PL/SQL, PostgreSQL it is PL/pgSQL, and for SQL Server it is TSQL. This technique involves using the functions of the procedural language to take all the distinct attributes of the EAV table and LOOP through them to systematically generate the necessary SQL for each.. This will dynamically create a flattened table with every possible attribute as a column.

Performance Considerations

Now lets cover some of the nuances and tradeoffs of these techniques as well as how to persist the end results for future use.

Subquery vs JOINs

Knowing whether to use the sub-query vs. join approach comes down to whether your underlying database is a row store or a column store.A column store will perform better with joins (excepting Redshift, which is limited in its ability to group after a correlated sub-select). Other than that the same query to flatten for column or row is about the same if all the indexes and partition keys are set up optimally. It is important for the database to scan only a single record on the EAV table for a single attribute for a single entity, this is the key to performance.

The final consideration for performance is to only query for exactly which attributes you need if the flattening occurs during querytime. Correlated subselects will be more performant for databases that support scanning only a single column per row compared to JOINs. A JOIN will calculate and impact memory more because it is using an entire row instead of just a single field before the projection.

If the data is being ETLed to a columnar or analytical database, it is better to flatten it in the ETL process instead of doing it at query time. It’s ok to have a lot of NULL values in a row since columnar databases are extremely efficient with sparse matrices and wide rows.

Setting up Indexes

For row store databases such as MySQL, Postgres, and SQL Server, the there will need to be an index set on the entity id and the attribute id. For our patient EAV tables, the columns that needed indexes would be patient_id and attribute. This will guarantee for each entity’s value for a single attribute will only scan one row.

For columnar databases such as Redshift and Vertica, there should be a tuple sortkey set on the entity id and attribute id as well. This will again limit the database from doing a sequential scan. Also we will want to DISTKEY(entity id) for Redshift or PARTITION BY entity id for Vertica. The primary entity table and other tables we want to join should follow the same strategy. This will limit queries from going across nodes to do JOINs.

Persisting Results

If your dataset is too large to flatten at querytime, it is important to create a persistence strategy for the flattened results.

For transactional datasets where attribute values are volatile and may change frequently, it is advised to DROP the persisted table and then regenerate it whenever a new attribute is added. This guarantees that state changes will be reflected in the persisted table.

Frequency of persisting depends largely on the performance needs and the real time needs of the use case. The flattened data should only be regenerated as frequently as the use case requires. If availability is a concern, the regeneration process should only DROP and UPDATE the new table to replace the old one after regeneration is complete. This makes sure that the flattened set is always available.

For big datasets, it is not feasible to regenerate the entire flattened set. If this data is immutable or stateless and static, it is important to do INSERTS or bulk INSERTS of new flattened rows into the flat table. This can be achieved with a script that logs when the last record or entity id that was flattened and then go out and flatten new ones to INSERT.

Further Considerations

When handling sparse datasets, it may make sense break apart the EAV table such that each entity type is stored in its own table. This will reduce the number of empty columns and may greatly improve performance for row store databases. We only want the entities on the same table if it makes sense for your specific analytical needs. When handling with multiple entity types in an EAV table that have different primary entity tables, it is often useful to break apart the EAV tables for each entities. Some attributes are shared for different primary entity tables for example address entity may be utilized for both customers and orders, and those should be kept on tables together. This is the concept of normalization for EAV tables, separate as much of the EAV tables as possible depending on which primary entities utilize them.

When considering whether or not your operational system should use EAV Model, it is important to look at the vast numbers of modern technologies for sparse data. Modern row store databases today are developing native handling of unstructured or sparse matrices. For example Postgres has extensive support for Key Value Pair.