Tip: How to Load Data with Duplicates into a Vertica Table with a UNIQUE Constraint

Moshe Goldberg - OpenText Vertica Principal Solutions Consultant

Data Analytics - VERTICA - Blog

When working with large datasets, it's common to encounter duplicate values. If you're loading such data into a Vertica table with a UNIQUE constraint, a single duplicate can cause the entire load to fail. Fortunately, Vertica offers powerful SQL features that let you filter and control what gets inserted.

Disclaimer:
This code is provided "as is" without any warranties or guarantees. It is intended for educational and demonstration purposes only. Always review and test in a QA or non-production environment before using in a live system.

The full demo script used in this article is available on GitHub here:
👉 https://github.com/mogomo/load_with_unique_filter/tree/main

Let's walk through a practical example of how to safely load only unique rows - and optionally keep track of duplicates - using staging tables.

🔧 Step-by-Step Guide

✅ Step 1: Define the Target Table

We begin with a fact table that enforces uniqueness on column f1.

```
CREATE TABLE my_fact_table (
    f1 INT CONSTRAINT my_constraint_name UNIQUE ENABLED,
    f2 VARCHAR(100),
    f3 FLOAT)
ORDER BY f1
SEGMENTED BY hash(f1) ALL NODES;
```

📄 Step 2: View the Source Data

The CSV data file includes duplicates on column f1:

```
cat /path_to_data/data.csv
1,new_one,1.1
2,new_two,2.1
1,new_three,3.1
3,new_four,4.1
4,new_five,5.1
2,new_six,6.1
```

❌ Step 3: What Happens If We Load Directly?

Trying to COPY the data into the target table fails due to duplicate keys:

```
COPY my_fact_table
FROM '/path_to_data/data.csv'
DELIMITER ',';
vsql:test.sql:25: ERROR 6745:  Duplicate key values:
'f1=1' -- violates constraint 'UNIQUE_SCHEMA.my_fact_table.my_constraint_name'
DETAIL:  Additional violations:
Constraint 'UNIQUE_SCHEMA.my_fact_table.my_constraint_name':
duplicate key values: 'f1=2'
```

🛟 **Step 4: Use a Staging Table Without Constraints**
We load all rows into a temporary staging table that has no constraints:

```
CREATE TEMPORARY TABLE staging_table_1 (
    f1 INT,
    f2 VARCHAR(100),
    f3 FLOAT
)
ON COMMIT PRESERVE ROWS KSAFE 0;
```

Loading all rows (including duplicates) into the staging table:

```
COPY staging_table_1
FROM '/path_to_data/data.csv'
DELIMITER ','
ABORT ON ERROR;
 Rows Loaded
-------------
     6
```

📌 **Step 5 (Optional): Separate Unique and Duplicate Rows**
We can flag which rows are unique and which are duplicates using an analytic query.
This is useful for logging, validation, or audit purposes.

```
CREATE TEMPORARY TABLE staging_table_2 ON COMMIT PRESERVE ROWS AS
WITH
  unique_list AS (
    SELECT *, TRUE AS _unique
    FROM staging_table_1
    LIMIT 1 OVER (PARTITION BY f1 ORDER BY f1)
  ),
  reject_list AS (
    SELECT d.*, FALSE AS _unique
    FROM staging_table_1 d
    LEFT JOIN unique_list u
      ON d.f1 = u.f1 AND d.f2 = u.f2 AND d.f3 = u.f3
    WHERE u.f1 IS NULL
  )
SELECT * FROM unique_list
UNION ALL
SELECT * FROM reject_list;
```

Rows that are considered unique and will be loaded:

```
SELECT f1, f2, f3 FROM staging_table_2 WHERE _unique ORDER BY 1;
 f1 |    f2    | f3
----+----------+-----
  1 | new_one  | 1.1
  2 | new_two  | 2.1
  3 | new_four | 4.1
  4 | new_five | 5.1
(4 rows)
```

Rows that are considered duplicates and will be excluded:

```
SELECT f1, f2, f3 FROM staging_table_2 WHERE NOT _unique ORDER BY 1;
 f1 |     f2      | f3
----+-------------+-----
  1 | new_three | 3.1
  2 | new_six   | 6.1
(2 rows)
```

🚀 **Step 6: Insert Only Unique Rows into the Target Table**
You can skip the duplicate logging step above and directly insert only the first row per f1 from the first staging table:

```
INSERT INTO my_fact_table
SELECT * FROM staging_table_1
LIMIT 1 OVER (PARTITION BY f1 ORDER BY f1);
COMMIT;
```

This is the content of the target table after loading rows with unique "f1" values:

```
SELECT *
FROM UNIQUE_SCHEMA.my_fact_table
ORDER BY f1;
 f1 |     f2     | f3
----+------------+-----
  1 | new_one  | 1.1
  2 | new_two  | 2.1
  3 | new_four | 4.1
  4 | new_five | 5.1
(4 rows)
```

🛟 **Avoid failing the load or inserting duplicates**
An additional challenge arises when some of the "f1" values already exist in the target fact table,
And we want to avoid failing the load or inserting duplicates.
In such cases, we can extend the filtering logic by performing an anti-join from the staging table to the fact table.
To demonstrate this example, we will first clean the fact table and then load one million values into it.
The one million values are all greater than 3, to demonstrate "f1 = 4" as a duplicate value.

```
TRUNCATE TABLE my_fact_table;

\set DEMO_ROWS 1000000
\set MIN_VALUE 4

INSERT INTO my_fact_table
with myrows as (select
row_number() over() + :MIN_VALUE -1 as f1
from ( select 1 from ( select now() as se union all
select now() + :DEMO_ROWS - 1 as se) a timeseries ts as '1 day' over (order by se)) b)
select f1, 'old_' || f1 as f2, f1 + 0.1 as f3
from myrows
order by f1;
COMMIT;
```

🚀 **Ensuring Only Truly New and Unique Keys Are Inserted**
The following query ensures that only truly new and unique keys are inserted.

Unique keys are defined as f1 values that appear only once in the loading data and do not already exist in the target table.

```
INSERT INTO my_fact_table
SELECT s.*
FROM (
  SELECT * FROM staging_table_1
  LIMIT 1 OVER (PARTITION BY f1 ORDER BY f1)
) s
LEFT JOIN my_fact_table t ON s.f1 = t.f1
WHERE t.f1 IS NULL;
COMMIT;
```

📊 Final Result

This is the final content of the first 6 lines in the target table after loading "new_" rows with unique "f1" values:

```
SELECT * FROM my_fact_table ORDER BY f1 limit 6;
 f1 |    f2     | f3
----+----------+-----
  1 | new_one  | 1.1
  2 | new_two  | 2.1
  3 | new_four | 4.1
  4 | old_4    | 4.1
  5 | old_5    | 5.1
  6 | old_6    | 6.1
(6 rows)
```

✅ Summary

- Direct loads into tables with UNIQUE constraints fail on duplicates.
- A constraint-free **staging table** allows safe loading of all rows.
- Use **analytic functions** like LIMIT 1 OVER (PARTITION BY ...) to select the first occurrence per key.
- Optional: log or inspect rejected duplicates before committing.