June 5, 2025

ISO/IEC JTC1/SC32/WG3

ANSI INCITS DM32

Database

Title: Non-positional INSERT statement

Author: Gerald Venzl

Project: SQL: 202X-CD

Status: Discussion Paper

1 Abstract

This paper proposes an enhancement to SQL that will allow INSERT statements to leverage non-positional column arrangements. This functionality will make INSERT statements easier to write, comprehend and maintain.

2 Introduction

Today, the target columns for the INSERT statement are arranged by position. The position of the columns can either be stated explicitly or if omitted, implies *[…] all columns of T in the ascending sequence of their ordinal positions within T […].*

The following example illustrates a common INSERT statement explicitly stating the column positions:

```
INSERT INTO employees (employee_id, first_name, last_name, salary, job_id)
              VALUES (        210, 'Gerald',   'Venzl',     1000,      3);
```

The following example illustrates a common INSERT statement with implied column positions:

```
INSERT INTO employees VALUES (210, 'Gerald', 'Venzl',  1000, 3);
```

An argument could be made that because the ordinal position within the table may not be guaranteed in between multiple executions of the INSERT statement, using implied column positions is brittle.

INSERT statements also allow insertion from a source that is a subquery instead of a table values constructor.

The following example illustrates a common INSERT statement using a subquery as source and explicitly stating the column positions:

```
INSERT INTO employees (employee_id, first_name, last_name, salary, job_id)
 SELECT                     emp_id,      fname,     lname,    sal, job_id
  FROM acquired_employees;
```

UPDATE statements, however, use a non-positional arrangement of columns.
The following example illustrates a common UPDATE statement:

```
UPDATE employees
 SET first_name  = 'Gerald',
     last_name   = 'Venzl',
     salary      =  1000,
     job_id      =     3
 WHERE employee_id = 210;
```

There are implied benefits of the SET clause leveraged by the UPDATE statement:

1. **The target column is immediately followed by the source input:** Unlike the INSERT statement, the target column and source input are specified next to each other. This makes it immediately clear to the reader/maintainer of the statement which value goes into what target column.
2. **Column position in the statement is not relevant:** A column/source pair can easily be added, rearranged, or removed from the statement with no risk of removing the wrong positional column or source value.

3 Benefits of non-positional column arrangements for INSERT statement

There are several usability benefits if the INSERT statement were to provide a non-positional variant. This section will elaborate on these.

3.1 Target tables with many columns

It is not uncommon for tables to have more than a hand full of columns. In praxis, tables often have 100s or more columns. Likewise, tables often have string-based columns that store several 100 bytes of data. In such scenarios, an INSERT statement can get rather long. Here is a practical example of an INSERT statement storing US National Park information:

```
INSERT INTO parks
  (park_id, park_code, name, full_name, url, description, designation, latitude,
longitude, states, directions_info, directions_url, weather_info, country_id)
 VALUES
 ('77E0D7F0-1942-494A-ACE2-9004D2BDC59E', 'abli', 'Abraham Lincoln Birthplace',
'Abraham Lincoln Birthplace National Historical Park',
'https://www.nps.gov/abli/index.htm', 'For over a century people from around the
world have come to rural Central Kentucky to honor the humble beginnings of our
16th president, Abraham Lincoln. His early life on Kentucky''s frontier shaped
his character and prepared him to lead the nation through Civil War. Visit our
country''s first memorial to Lincoln, built with donations from young and old,
and the site of his childhood home.', 'National Historical Park', 37.5858662, -
85.67330523, 'KY', 'The Birthplace Unit of the park is located approximately 2
miles south of the town of Hodgenville on U.S. Highway 31E South. The Boyhood
Home Unit at Knob Creek is located approximately 10 miles northeast of the
Birthplace Unit of the park.',
'http://www.nps.gov/abli/planyourvisit/directions.htm', 'There are four distinct
seasons in Central Kentucky. However, temperature and weather conditions can vary
widely within those seasons. Spring and Fall are generally pleasant with frequent
rain showers. Summer is usually hot and humid. Winter is moderately cold with
mixed precipitation.', 'USA');
```

In contrast, below is what the statement could look like leveraging the SET clause, adding much to the statement's readability and maintainability:

```
INSERT INTO parks
 SET
   park_id         = '77E0D7F0-1942-494A-ACE2-9004D2BDC59E',
   park_code       = 'abli',
   name            = 'Abraham Lincoln Birthplace',
   full_name       = 'Abraham Lincoln Birthplace National Historical Park',
   url             = 'https://www.nps.gov/abli/index.htm'
   description     = 'For over a century people from around the world have come
                      to rural Central Kentucky to honor the humble beginnings
                      of our 16th president, Abraham Lincoln. His early life on
                      Kentucky''s frontier shaped his character and prepared him
                      to lead the nation through Civil War. Visit our country''s
                      first memorial to Lincoln, built with donations from young
                      and old, and the site of his childhood home.',
   designation     = 'National Historical Park',
   latitude        = 37.5858662,
   longitude       = -85.67330523,
   states          = 'KY',
   directions_info = 'The Birthplace Unit of the park is located approximately 2
                      miles south of the town of Hodgenville on U.S. Highway 31E
                      South. The Boyhood Home Unit at Knob Creek is located
                      approximately 10 miles northeast of the Birthplace Unit of
                      the park.',
   directions_url  = 'http://www.nps.gov/abli/planyourvisit/directions.htm',
   weather_info    = 'There are four distinct seasons in Central Kentucky.
                      However, temperature and weather conditions can vary
                      widely within those seasons. Spring and Fall are generally
                      pleasant with frequent rain showers. Summer is usually hot
                      and humid. Winter is moderately cold with mixed
                      precipitation.',
   country_id      = 'USA';
```

3.2 Inserting multiple non-uniform rows

The table values constructor allows for insertion of multiple rows in one INSERT statement. However, constructing such a statement can become unnecessarily verbose if the row values are not uniform, meaning that the new rows do not provide values for the same columns. Explicit NULL or DEFAULT keywords are needed to guarantee that the intended column positions remain intact:

```
INSERT INTO jobs (job_id, job_title, min_salary, max_salary)
  VALUES         (3, 'Product Manager', 5000, NULL),
                 (4, 'Junior Technical Consultant', NULL, 4000),
                 (2, 'Vice President', NULL, NULL);
```

In contrast, below is what the statement could look like leveraging a potential SET clause extension. In this scenario, neither NULL keywords nor the target columns need to be specified, adding much to the statement's readability and maintainability:

```
INSERT INTO jobs SET
 (job_id = 3, job_title = 'Product Manager', min_salary = 5000),
 (job_id = 4, job_title = 'Junior Technical Consultant', max_salary = 10000),
 (job_id = 5, job_title = 'Vice President');
```

3.3 Grouping of logical entities

There are certain data elements that are meaningless on their own but meaningful when combined with other elements. Addresses are such an example. A street name by itself may not mean much without information about the city, postal code and country. Today, the INSERT statement has no provisions to clarify the existence of such *logical entities*. However, it would be easy to extend the INSERT statement to add such a grouping functionality specified via `( )`, aiding the readability and maintainability of an INSERT statement, for example:

```
INSERT INTO employees
 SET  employee_id = 1,
      job_id = 5,
     (first_name, last_name) = ('Gerald', 'Venzl'),
     (street, zip, city)     = ('400 Oracle Parkway', 94065, 'Redwood Shores');
```

3.4 Positionless INSERT statement with subquery

The demonstrated hypothetical SET clause extensions do not work for INSERT statements with subqueries as subqueries are a standalone construct. There is no way to map the target columns and source columns to each other without either injecting new syntax into the SELECT statement or removing from it. However, the source and target columns could still be matched by their **names or aliases** instead of their position.

An INSERT statement could provide a new `BY POSITION | BY NAME` clause that would determine whether the source and target columns are matched by their position (the current behavior) or by their name or alias. A column present in the SELECT list but not in the target table would generate a parsing error. The `BY NAME` extension would furthermore eliminate the need for specifying the target columns altogether. Difference in name between a source and target column can be overcome by providing an alias for the source column that matches the target column, for example:

```
INSERT INTO job_history
  BY NAME
 SELECT employee_id, job_id, department_id,
```

```
       hire_date AS start_date, CURRENT_DATE AS end_date
  FROM employees
   WHERE employee_id = 206;
```

Because SELECT lists can be arbitrary long and complex, a more sophisticated example may better demonstrate the usefulness of matching by name, for example:

```
INSERT INTO employee_olap_results
 BY NAME
-- complex OLAP query
SELECT
 employee_id,
 salary,
 department_id,
 AVG(salary) OVER (PARTITION BY department_id)
   AS dept_avg_salary,
 RANK() OVER (PARTITION BY department_id ORDER BY salary DESC)
   AS salary_rank_in_dept,
 SUM(salary) OVER (PARTITION BY department_id ORDER BY employee_id)
   AS running_total_salary,
 RATION_TO_REPORT(salary) OVER (PARTITION BY department_id) * 100
   AS pct_of_dept_salary,
 salary - AVG(salary) OVER (PARTITION BY department_id)
   AS salary_diff_from_dept_avg,
 FIRST_VALUE(salary) OVER (PARTITION BY department_id ORDER BY employee_id)
   AS first_salary_in_dept,
 LAST_VALUE(salary)
     OVER (PARTITION BY department_id
       ORDER BY employee_id
       ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
   AS last_salary_in_dept,
 LEAD(salary, 1, salary)
     OVER (PARTITION BY department_id ORDER BY salary) - salary
   AS salary_gap_to_next,
 AVG(salary)
     OVER (ORDER BY employee_id ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING)
   AS moving_avg_salary,
 PERCENTILE_COUNT(0.5) WITHIN GROUP (ORDER BY salary) OVER ()
   AS median_salary
 FROM
  employees
   ORDER BY
    department_id, employee_id;
```

3.5 Rigorous column list

There is one apparent benefit of today's INSERT statement with an *explicitly* specified target column list for both, the table value constructor and subquery sources: The specified target column list tells a user in one place of the statement into which columns values are being inserted into rather than in multiple places throughout the statement. This makes is easy for a user to comprehend the columns touched by the statement and easy to remove a column from the INSERT, if providing a value is no longer desired.

However, the hypothetical SET clause and BY NAME extensions could provide the same capability via a *rigorous* target column list. With such an extension, the column list is provided in the same way as it already can be, but the list would now ensure that the only columns that can be manipulated by the SET clause are those specified in the target column list. Extra columns listed or columns omitted in the SET clause will generate a compile time error.

The following example uses the hypothetical rigorous column list on a single row INSERT, ensuring all columns stated in the target column list need to be present in the SET clause:

```
INSERT INTO jobs (job_id, job_title, min_salary)
  SET job_id = 3,
      job_title = 'Product Manager',
      min_salary = 5000;
```

The following example uses the hypothetical rigorous column list on a multi-row row INSERT, ensuring all columns stated in the target column list need to be present in the SET clause and hence the rows need to be uniform:

```
INSERT INTO jobs (job_id, job_title, min_salary) SET
 (job_id = 3, job_title = 'Product Manager', min_salary = 5000),
 (job_id = 4, job_title = 'Junior Technical Consultant', min_salary = NULL),
 (job_id = 5, job_title = 'Vice President', min_salary = NULL);
```

The following example uses the hypothetical rigorous column list on an INSERT with subquery, ensuring all columns stated in the target column list are present in the SELECT list:

```
INSERT INTO job_history (employee_id, job_id, department_id, start_date, end_date)
  BY NAME
 SELECT employee_id, job_id, department_id,
        hire_date AS start_date, CURRENT_DATE AS end_date
  FROM employees
   WHERE employee_id = 206;
```

4 Conclusion

Providing the capability of a non-positional INSERT statement greatly enhances its usability, whether that's for the initial writer of the statement, its reader or maintainer. It would make INSERT statements "self-documenting code". Special care has been taken in the proposed syntax to match as much of the existing constructs as possible. With the outlined `SET` clause extensions, the syntax of an INSERT statement and UPDATE statement are remarkably similar, making the new capability appear natural to long-time SQL users and newcomers alike. The minimal addition of the `BY NAME` keywords for subquery INSERT statements add a lot of usability benefits with minimal impact.

- End of paper -