September 19, 2025

ISO/IEC JTC1/SC32/WG3
ANSI INCITS DM32
Database

Title: Data exploration simplifications

Author: Gerald Venzl

Project: SQL: 202X-CD

Status: Discussion Paper

Revision	Date	ISO/IEC JTC1/SC32/WG3	ANSI/INCITS DM32	ANSI/INCITS DM32 GQL EG
Original	202x-xx-xx	XYZ-000	DM32-2025-00xxx	

References:

- [BCN-036: Select list EXCLUDE] – Peter Eisentraut

Abstract:

This paper proposes an enhancement of the `SELECT *` functionality to allow users modification of the star expanded column list of the source table(s) to simplify data exploration and reporting use cases.

1 Background

The current definition of SELECT * in 7.16 <query specification> SR 7) Case) is:

a) If the <select list> "*" is simply contained in a <table subquery> that is immediately contained in an <exists predicate>, then the <select list> is equivalent to a <value expression> that is an arbitrary <literal>.

b) Otherwise, the <select list> "*" is equivalent to a <value expression> sequence in which each <value expression> is a column reference that references a column of T and each column of T is referenced exactly once. The columns are referenced in the ascending sequence of their ordinal position within T.

Hence, in the common case, SELECT * is used to query all column of a table. This feature has long been in the standard and probably originated from a time when data volumes allowed tables to have a few columns. It has mostly been seen as a simplification feature to quickly retrieve all columns from a table, whether this is for a user to display the content of all the columns in a table or simplify data manipulation tasks such as `CREATE TABLE … AS SELECT *` or `INSERT INTO … SELECT *`.

This paper refers to this behavior as ***star expansion*** (*star* being the asterisk (*) symbol as `SELECT *` is often verbally spoken as "select star").

2 Drawbacks

Although star expansion provides benefits for some use cases, it has become an anti-pattern (bad practice) over recent decades as it has several drawbacks to the following areas:

- **Maintainability**: The query results may change unexpectedly, breaking dependent code or reports when the table structure changes, i.e., columns are added, renamed or removed.
- **Readability and Intent**: Not explicitly listing columns makes the query's purpose less clear to the reader and maintainer of the query.
- **Security Concerns**: Exposing unnecessary columns can inadvertently leak sensitive data.
- **Performance Impact and Resource Waste**: Retrieving all columns from a table, including those not needed, consumes unnecessary data transfer bandwidth, memory, and processing time.

3 Modern day requirements

Although the limitations mentioned earlier still hold true today, they primarily apply to scenarios with fixed schemas and predictable queries from applications or users expecting consistent results.

In contrast, SQL has emerged as the go-to language for exploratory data analysis and investigation, especially when dealing with unfamiliar or unreliable data structures. This data often originates from outside sources, like flat files, web resources, or temporary staging tables. People turn to SQL for these tasks because its declarative style simplifies specifying row-level filters and aggregations, allowing them to make sense of the data step by step.

However, when the data isn't fully understood yet, users frequently need to apply comparable filters to specific columns of interest. This is where SQL struggles: there's no built-in way to declaratively specify a dynamic or flexible set of columns, unlike the straightforward row filtering options available.

This issue persists even in established data warehousing environments, where denormalization leads to "wide" tables containing hundreds of columns—a technique that's widely accepted and effective. The rigid requirement to list columns explicitly becomes a major hurdle. While `SELECT *` can pull every column at once, there's no mechanism to tweak or narrow that selection declaratively. It's an all-or-nothing choice: grab everything or spell out exactly which columns you want. As a result, the convenience of `SELECT *` diminishes rapidly once you need to drop irrelevant columns. A typical data exploration process (simplified here) looks like this:

1. Retrieve all columns to get an overview.
2. Identify which ones aren't relevant to the current analysis.
3. Exclude those uninteresting columns.
4. Re-run the query with the updated column set and iterate from step 1.

Under the current SQL rules for star expansion, this works only for the initial round. After that, you must manually retype the full list of desired columns for any further tweaks, which quickly becomes tedious and error prone.

4 Data exploration simplification enhancements

Star expansion could significantly enhance data exploration and analysis if it supported dynamic column selection. Some SQL implementations have recognized this need and begun introducing features to extend star expansion, addressing its current limitations.

This paper proposes a new set of features for star expansion, enabling users to define a flexible, dynamic list of columns.

4.1 SQL implementations

Several SQL implementations provide star expansion enhancements. Paper [BCN-036: Select list EXCLUDE] listed some for the `EXCLUDE` functionality but there are some that provide additional capabilities, for example, DuckDB and Snowflake.

4.2 Enhancements

This paper proposes the discussion of the following hypothetical star expansion enhancements. A more detailed outline for each feature is provided below:

```
[<table alias>.]* (
  [ ILIKE [ NOT ] <character pattern> [, [ NOT ] <character pattern> ]…
    [ ESCAPE <escape character> ] ]
  [ EXCLUDE <column name> [, <column name> ]… ]
  [ REPLACE <column name> WITH <value expression>
    [, <column name> WITH <value expression>]… ]
  [ PREFIX <prefix> ]
  [ RENAME <column name> AS <column alias> [, <column name> AS <column alias> ] ]
)

<prefix> ::= <identifier>

<column alias> ::= <identifier>
```

The columns included in the query output will be filtered or modified based on the specified syntax before the query executes. If the filtering process removes all columns from the SELECT statement, an error will occur, similar to writing a `SELECT FROM …` statement with no items in the select list. When using multiple star expansions across different tables (e.g., `SELECT t1., t2., t3.* FROM …`), it's acceptable for all columns from one or more tables to be excluded. An error is triggered only if all columns from all tables are removed.

All operators are optional.

The operators must be specified in the order outlined, which matches their execution sequence. Notably, operators are executed from left to right, so the output of one operator affects the input of the next. This behavior offers specific advantages, which are detailed in later sections.

The operator order was debated extensively. While it's technically possible to let the execution order follow the sequence of operators as written in the query, allowing users to decide the arrangement, this approach would place a significant cognitive burden on users. They would need to carefully parse and recall the star expansion clause of each query to understand the operator sequence, especially when multiple star expansions are involved.

Ultimately, it was agreed that a fixed execution order, mirrored by a consistent operator order in the syntax, would be more user-friendly. This ensures users need to learn and understand only one standard sequence, which they can depend on consistently. This approach aligns with existing implementations that provide star expansion modification features.

If users need a different order, they can achieve this by nesting star expansion clauses within subqueries.

4.2.1 ILIKE [NOT] <character pattern>

The ILIKE operator includes columns in the projection based on one or more case-insensitive (definition of case-insensitive needs to be specified) matching patterns. The pattern(s) can be negated with the NOT keyword. Its syntax mirrors the LIKE predicate functionality but uses an “I” in front to signal that it is an “Identifier LIKE” that will match identifier names based on case-insensitive patterns instead of case-sensitive values within columns. While LIKE can be used with COLLATE to define the pattern matching of values, supporting COLLATE for column name identifiers seems excessive given that column names are generally case insensitive unless encircled in double quotes.

Just as with the LIKE operator, the ILIKE operator also provides special characters _ (underscore) to match exactly one character and % (percent sign) to match zero or more characters. The ESCAPE clause can be used to provide an escape character prior to the special character to tell the implementation that the character is to be used as literal value and not as a pattern matching character.

Multiple patterns result in a union, evaluated with OR, not AND. If multiple patterns match the same column, it’s included only once without triggering an error.

Examples

```
-- Project columns that contain the word 'TEXT' in any case,
-- for example, 'Text' or 'text' or 'tExT'
SELECT * ( ILIKE '%text%' ) FROM products;

-- Project columns that do not end with '_ID' in any case
SELECT * ( ILIKE NOT '%\_id' ESCAPE '\' ) FROM employees;

-- Pproject columns that match 'employee_id' in any case
SELECT * ( ILIKE 'employee_id' ) FROM employees;

-- Project columns that do not end with 'ID' OR 'TMS' in any case
SELECT * ( ILIKE NOT '%id', NOT '%tms' ) FROM employees;

-- Project columns that start with 'VENDOR' OR end with 'ID' in any case
-- Includes columns vendor_name, vendor_code, vendor_id, cab_id, customer_id
SELECT * ( ILIKE 'vendor%', '%id' ) from cab_rides;

-- Project columns that start with 'spare0' or not with 'spare'
-- Includes all (other) columns from the table and selectively 'spares0-9'
select * ( ilike 'spare0%', not 'spare%' ) from productsides;

-- Produces all columns that do not end with 'ID' or start with 'product'
-- Includes product_code, production_quantity
-- and also product_id because it matches the second pattern.
-- customer_id, create_tms, updated_tms, adjusted_quantity
select * ( ilike not '%id', 'product%' ) from products;
```

The final examples are written in lowercase, mimicking typical ad-hoc queries, to show the advantage of case-insensitive pattern matching. Requiring patterns to match column case adds unnecessary burden without clear benefits, as users typically don’t care about case when selecting columns for the SELECT list. Since SQL identifiers are generally case-insensitive, pattern matching for star expansion should also be case-insensitive.

4.2.2 EXCLUDE <column name>

The EXCLUDE operator removes a comma-separated list of columns from the star expansion. If the identifier does not exist in the table, an error is thrown. Unlike the ILIKE operator, which filters columns based on patterns, EXCLUDE directly specifies identifiers to omit. Users may want to select a group of columns with a pattern and then exclude specific ones. While EXCLUDE can technically come before or after ILIKE, it's more intuitive for users to first match a pattern with ILIKE and then exclude columns with EXCLUDE, rather than excluding columns from the star expansion and then applying a pattern. For clarity, defining EXCLUDE to follow ILIKE is preferred.

A question arises of what should happen when the ILIKE filtered a column via a pattern that is also referred to in the EXCLUDE operator. The consensus was that for user friendliness EXCLUDE should not raise an error if a column has already been filtered out by ILIKE as no harm has done by double filtering. It is more likely that the user started off with EXCLUDE, got to a long list and introduced ILIKE, and forgot to remove the column from the EXCLUDE, either because it wasn't obvious that the pattern matched the column or because of an oversight.

Examples

```
-- Exclude the manager_id from the result
SELECT * ( EXCLUDE manager_id ) FROM employees;


-- Exclude the employee_id, salary and bonus from the result
SELECT * ( EXCLUDE employee_id, salary, bonus ) FROM employees;


-- Combine ILIKE and EXCLUDE
-- Project all columns that end with 'name' in any case, exclude column middle_name
SELECT * ( ILIKE '%name' EXCLUDE middle_name ) FROM employees;


-- Combine ILIKE and EXCLUDE
-- Project all columns that end with 'name' in any case, exclude started_tms
-- The result of ILIKE already prunes started_tms yet the user is forgiven
-- calling out started_tms in the EXCLUDE clause as the result is the same
SELECT * ( ILIKE '%name' EXCLUDE started_tms ) FROM employees;
```

4.2.3 REPLACE <column name> WITH <value expression>

Data exploration involves examining data, which can be challenging when columns contain lengthy values, like text, descriptions, or notes. These long values often dominate the output, complicating data analysis. While the `EXCLUDE` or `ILIKE` operators can omit such columns, users may still want to view parts of these columns for context or to check if they contain values. Functions like `SUBSTRING` could truncate these values but applying them prevents the use of `SELECT *`. Similarly, for columns of type `TIMESTAMP`, users might want only the date portion to avoid cluttered output, but applying a function to format it also precludes `SELECT *`.

The `REPLACE` operator addresses these issues by allowing users to substitute a column's projection with an expression while preserving its name and position in the `SELECT` list. It supports replacing column values with any arbitrary <value expression>, even if the result is a different data type. This enables data type conversions, such as converting a numeric value stored as a string in an external source or table into a number, all while maintaining star expansion and avoiding the need to list all columns explicitly.

Examples

```
-- Replace the description and comments with only the first few characters
SELECT *
  ( REPLACE description WITH SUBSTRING(description FROM 1 FOR 10),
            comments    WITH SUBSTRING(comments FROM 1 FOR 5)
  )
 FROM products;

-- Replace timestamp columns with only the date portion
SELECT * ( REPLACE placed_tms WITH
                   SUBSTRING(
                     CAST(placed_tms AS VARCHAR(35))
                     FROM 1 FOR 10)
       )
  FROM orders;

-- Project all columns not ending in '_ID' in any case, exclude coordinates,
-- shorten description and comments, convert timestamp to day precision
SELECT *
  ( ILIKE NOT '%\_id' ESCAPE '\'
    EXCLUDE coordinates
    REPLACE description WITH SUBSTRING(description FROM 1 FOR 10),
            comments WITH SUBSTRING(comments FROM 1 FOR 5),
            placed_tms WITH
            SUBSTRING(
              CAST(placed_tms AS VARCHAR(35))
            FROM 1 FOR 10)
  )
 FROM orders;
```

4.2.4 PREFIX

When selecting columns from multiple tables, adding a prefix helps identify the source table of each column.

The `PREFIX` operator adds a specified prefix to all columns in the star expansion. It is applied after the `ILIKE` and `EXCLUDE` operators to avoid interfering with user-defined filtering. If the resulting column name exceeds the maximum allowed length, an error is triggered.

This operation may cause name collisions, which must be handled appropriately. However, due to the complexity of name collision rules in the standard, they are beyond the scope of this discussion paper.

Examples

```
-- Prefix all columns with table initials
SELECT e.* ( PREFIX e_ ), d.* ( PREFIX d_ )
 FROM employees e
   INNER JOIN departments p ON (e.dept_id=d.id)
 ORDER BY d.name;
```

The result of the above query would yield columns from the employee table `first_name`, `last_name` as `e_first_name`, `e_last_name`, and the `name` of the department as `d_name`.

```
-- Retrieve all products in stores sold in Austria
-- From the stores table:
--   1) Exclude store_id, opening_hours, coordinates
--   2) Prefix all projected store columns with s_ (table initial)
-- From products table:
--   1) Exclude product_id
--   2) Change description to substring of the first 10 characters
--   3) Prefix all projected columns with p_ (table initial)
SELECT s.* ( EXCLUDE store_id, opening_hours, coordinates
             PREFIX s_
       ),
       p.* ( EXCLUDE product_id
             REPLACE description WITH SUBSTRING(description FROM 1 FOR 10)
             PREFIX p_
       )
 FROM stores s
   INNER JOIN stores_products sp ON (s.store_id=sp.store_id)
   INNER JOIN products p ON (sp.product_id=p.product_id)
   INNER JOIN countries c ON (s.country_id=c.country_id)
  WHERE c.name = 'Austria';
```

4.2.5 RENAME <column name> AS <column alias>

Users often like to give columns a to them a more meaningful name. The 7.16 <query specification> <as clause> (column alias, i.e. `col1 AS alias1`) has long fulfilled this requirement. Yet it cannot be applied for `SELECT *` but instead requires the column to be explicitly typed.

The `RENAME` operator overcomes this restriction for `SELECT *`. The operator is applied after all other operators to avoid accidentally affecting them. Specifically, it is applied after `ILIKE` and `EXCLUDE` to avoid accidental invalidation of filtering instructions, and it is applied after the `PREFIX` operator meaning that the column to be renamed must state the prefix.

This order allows users to add a prefix to all columns but remove the prefix from selective ones.

This operation can produce a name collision which needs to be handled appropriately. However, because the name collision rules in the standard are complex, they are out of scope for this discussion paper.

Examples

```
-- Rename fname to first_name, lname to last_name
SELECT * ( RENAME fname AS first_name, lname AS last_name )
 FROM employees;


-- Replace description results with first 10 characters
-- Change column name to desc_short
SELECT * ( REPLACE description WITH SUBSTRING(description FROM 1 FOR 10)
           RENAME  description AS desc_short
       )
  FROM products;


-- Only project ID columns
-- Rename them to identifiers
SELECT * ( ILIKE '%id'
           RENAME emp_id AS employee_identifier
                  mgr_id AS manager_identifier
         )
  FROM employees;


-- Prefix all columns with table initials e_ and d_
-- Remove duplicate e.department_id and mark d.department_id as common column
SELECT e.* ( EXCLUDE dept_id PREFIX e_
       d.* ( PREFIX d_ RENAME d_department_id AS department_id)
 FROM employees e
   INNER JOIN departments p ON (e.department_id=d.department_id)
 ORDER BY d.name;
```

- End of paper -