Execute formulas stored in database fields with (almost) standard SQL

10 jan 2024

Harness the Potential of Storing and Executing Formulas from Database Fields for Expanded Analytical Power

In current Database Management Systems (DBMS), the inability to store and execute formulas at the individual record level poses a significant challenge.

This practice is generally discouraged owing to security issues, such as susceptibility to SQL injection vulnerabilities, performance-related concerns as well as the need for robust error handling mechanisms and careful validation and sanitation of user input.

Moreover, there exists a lack of tidy implementations seamlessly incorporating this functionality into SQL.

Colbert, however, disrupts this norm by seamlessly integrating formula storage and execution into SQL in a secure and straightforward manner. Notably, this integration adheres to the principle of orthogonality, whereby different language features can be combined or used independently in a consistent and predictable manner.

This article provides a concise technical exploration of Colbert's capabilities by delving into the mechanics of storing and executing formulas within SQL.

What we are familiar with

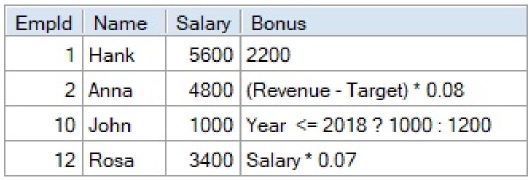
We are already familiar with *calculated fields* that apply to an entire column.
So in the following SQL statement, each employee is assigned a bonus, calculated according to the same formula: (Revenue – Target) * 0.05.

```
Select EmpId,Year,Name, (Revenue - Target) *
0.05 as bonus
from Employees natural join Sales
```

What we would like

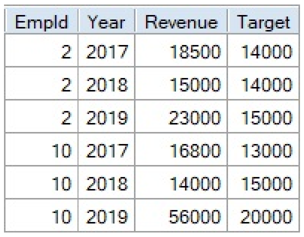
But what if every employee could have a different bonus agreement?
In that case we should be able to enter a bonus formula for each employee using the same syntax as when defining a *calculated field*.
Colbert facilitates this by introducing a novel category of field, denoted as a *Formula Field*.

Formula Field



EmpId	Name	Salary	Bonus
1	Hank	5600	2200
2	Anna	4800	(Revenue - Target) * 0.08
10	John	1000	Year <= 2018 ? 1000 : 1200
12	Rosa	3400	Salary * 0.07

Employees



EmpId	Year	Revenue	Target
2	2017	18500	14000
2	2018	15000	14000
2	2019	23000	15000
10	2017	16800	13000
10	2018	14000	15000
10	2019	56000	20000

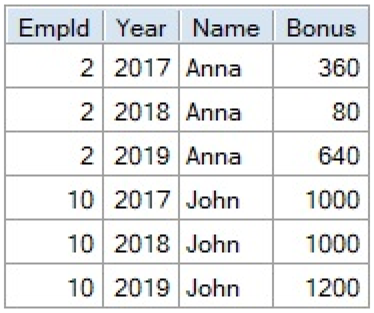
Sales

The Employees table has a *formula field* called Bonus. The Sales table contains sales figures per year.

To determine the annual bonus per employee using the Bonus Formula, Colbert enables the execution of the following simple SQL statement.

```
Select EmpId, Year, Name, Bonus
from Employees natural join Sales
```

This Query produces the next result:



EmpId	Year	Name	Bonus
2	2017	Anna	360
2	2018	Anna	80
2	2019	Anna	640
10	2017	John	1000
10	2018	John	1000
10	2019	John	1200

Annual bonus per employee calculated by the bonus formula

Design Principles: Simplicity, Power, and Safety.

Colbert follows certain fundamental design principles to achieve optimal solutions that embody simplicity, power, and safety. *Formula fields* should seamlessly integrate, feeling as though they've always been a natural part of SQL.

- **No extension of SQL**: The solution should refrain from introducing additional complexities to SQL.
- **Intuitive integration of formulas**: The implementation should prioritize a user-friendly and instinctive utilization of mathematical expressions.
- **Robust type checking**: The solution should incorporate a robust type mechanism to enhance data integrity and prevent type-related errors.
- **Orthogonality within SQL**: The solution should be orthogonal meaning that *everything fits on everything* within the defined syntax for SQL. The solution must avoid unnecessary constraints on the use of *formula fields*.

The concept behind this: The novel Expr<T> type.

The Necessity of One Slight Extension of SQL.

The underlying concept is inherently straightforward and consequently, highly intuitive. The system is completely type-checked, while formulas can contain any number of parameters of any type.
Contrary to the previously mentioned design principle, the declaration of a *formula field* necessitates a slight extension of the SQL syntax.
Colbert introduces a novel field type, Expr<T>, where T can be any of the known data types, such as int, string, datetime, bool, or double. The SQL statement for creation of the Employees table is now:

```
Create table Employees
(EmpId int, Name string, Salary double, Bonus
Expr<double>)
```

This marks the initiation of a discussion on the type mechanism: Conventionally, the type of a formula is determined by the result type and the number and type of

each formal parameter.
However, in the context of defining this formula field Expr<T>, there is a lack of information regarding the formal parameters. At this stage, it is intentionally preferred not to define anything about them and allow any set of formal parameters. However, it makes sense to define the result type T so that it works just like any other field.

In the context of the example regarding Bonus calculation, the Bonus formula can be any expression with any number of formal parameters, as long as it yields a value that can be implicitly typecast to double, when executed with the correct actual parameters. This part of type checking is enforced on entering an expression in a formula field.

> *For a more in-depth understanding of type checking and error handling, please refer to the upcoming blog post.*

Two manifestations of a formula field

The Dual Nature of Formula Fields in SQL

A *formula field* manifests itself in two facets:

Code:	the formula itself, represented by an expression
Calculated:	the calculated value derived from that expression in an certain context af parameters.

Upon invoking the name of the *formula field* in SQL, the calculated manifestation is
evoked, with the exception of the SQL insert clause and update clause.
In all other SQL instances where the *formula field* remains unmentioned, such as in *Select * from Employees*, the code is evoked.

Enforcement of either manifestation can be achieved through the utilization of the prefix operators ***code*** *<formula field>* or ***calculated*** *<formula field>*, though their necessity is infrequent.

```
Insert into Employees set
EmpId = 18,
Name = ‘William’,
Salary = 4000,
Bonus = (Revenue /
Target) * Salary
```

*Bonus refers to the **code** of the formula field*

```
Update Employees
set Bonus = Salary * 0.04
where EmpId = 18
```

*Bonus refers to the **code** of the formula field*

```
Select EmpId, Year, Name,
Bonus
from Employees natural
join Sales
```

*Bonus evokes the **calculated** value of the formula field*

```
Select
EmpId, Year, Name,
Revenue, Target,
code Bonus, calculated
Bonus
from Employees natural
join Sales
```

*First Bonus evokes the **code** while the second evokes the **calculated** value.* (*)

(*) This last SQL statement will produce the following output (first rows only).
The operator *calculated* might be omitted as it is default here. It will produce the same result when omitted.

code of the formula and calculated value of the formula are invoked

Parameter binding

The scope of parameters is defined by the place of invocation and the parameters are identified by their respective names.

As mentioned before, when entering an expression in a *formula field* we do not know anything about the parameters. They might be in the same table as the f*ormula field* but the power of *formula fields* lies in the ability to apply the formulas dynamically to data in in whatever tables at whatever moment. Moreover, these formulas are designed to accommodate any number of parameters, regardless of their type or origin.

The binding of parameters occurs when the *formula field* is called by its name in a SQL statement. The scope of parameters is then defined by the place of invocation and the parameters are identified by their respective names.

Consider the next SQL statement :

```
Select Name, Year, Bonus
from Employees natural join Sales
```

The scope the *formula field* Bonus in this SQL statement are all the fields from Employees and Sales. But instead of providing actual parameters to the formal parameters when the formula is called, Colbert searches for corresponding names of the formal parameters within the scope of the called formula.

This parsing, compiling, and parameter binding occur once for each formula when it is called for the first time, and then the compiled and binded formula is stored in the cache.
If the parameter cannot be found in that scope or if the parameter has a type incompatible with the formula evaluation, the *formula field* yields an error value. This part of type checking is enforced on executing a *formula field*.

Type error values function analogously to null values, and Colbert incorporates effective mechanisms for their handling.

As a general guideline, any expression that could be inputted in a calculated field within a SELECT statement is permissible in a *formula field*, given the presence of the relevant parameters within the current execution scope.

> *For a more in-depth understanding of type checking and error handling, please refer to the upcoming blog post.*

Orthogonality

> ***It is like playing with Lego blocks.***

The concept of *formula field*, mirroring the overall design philosophy of Colbert, is implemented without unnecessary restrictions and adheres to the principle of *orthogonality*. It's like building with Lego blocks where each piece can connect to any other piece, allowing for a wide range of combinations and flexibility. This design principle can contribute to cleaner, more modular code and make it easier for developers to understand and work with the feature.

A *formula field* therefore applies not only within a Select clause, but also in a c*alculated field*, a *Where* clause, an *order-by*-clause or within window functions. Actually a *formula field* can be applied just like any other field. Consequently, all subsequent SQL statements are valid within Colbert.

```
Select Empid, Year, Name,
Bonus div 100
from Employees natural join
Sales;
```

in calculated field

```
Select Empid, Year, Name
from Employees natural join
Sales
where Bonus > 1000;
```

in where clause

```
Select Year, Name, Bonus
from Employees natural join
Sales
order by Bonus desc;
```

in order by clause

```
Select Year, sum Bonus, max
Bonus
from Employees natural join
Sales
group by Year;
```

in aggregation

Examples

It's rather odd that mathematical expressions couldn't be stored in database fields, considering these expressions can represent relevant attributes of objects.
Consider, for example:

- the bonus calculation mutually agreed upon with an employer
- the search criteria for finding a new car
- tax calculations
- criteria for the segmentation of a population
- identification of risk factors.

A subsequent blog post will provide detailed insights into additional use cases. Meanwhile, the following examples serve to illustrate instances efficiently solvable through the application of *formula fields*. In the absence of *formula fields*, resolving these cases requires extensive programming efforts external to the database, yielding a less flexible solution.

1) Example: matching cars and buyers

In the context of a database containing information about used cars and another table with buyers specifying their criteria for car searches, the power of a boolean *formula field* becomes evident.
The table representing used cars resembles:

Cars

As a *formula field* is capable of producing results of any data type, a boolean *formula field* could, for example, define an individual's interest in purchasing a particular used car.
The SQL statement for creating a table for this purpose appears as follows:

```
Create table Buyer (Name String, Find Expr<Bool>);
```

Inserting a record in SQL:

```
Insert into Buyer
set Name = ‘Alex’, Find = Brand = ‘Porche’ and Price <
200500;
```

Entering values through Colbert's user interface is more convenient. Subsequently, the populated Buyer table could have the following appearance:

Colbert permits specific operators to have an identifier as an operand, treating it as a string. This applies for the right operand of the **==** *operator.*

Buyer

Colbert features an extended SQL known as Slim SQL, which is detailed in the upcoming blog post about Slim SQL.

Given the tables **Cars** and **Buyer** one can effortlessly extract relevant information by utilizing the *formula field* **Find**.

Matching cars to the buyer's specified criteria

```
Select Name, Cars.* from
Buyer Join Cars where Find
```

Buyer matched Cars

Buyers for whom there is no corresponding matching car:

```
Select Buyer.* from
Buyer Outer Join Cars on Find
where Brand is null
```

Buyer no matched Car

Cars with multiple possible buyers:

```
Cars.*, Count desc as Lead from
Cars Join Buyer where Find and Lead > 1
```

*Colbert lets you **sort in the from clause**, skip **group by**, and apply **where** to aggregated values.*

Cars with multiple buyers

Colbert features an extended orthogonal SQL known as Slim SQL, which is detailed in the upcoming blog post about Slim SQL.

2) Example: Evaluating Populations

When assessing a dataset, several goals may be pursued.

- The population could be categorized into cohorts.
- Risk factors might be identified
- Rules could be applied.

In the following example, each record in the Population is assigned a Score based on specific criteria Def.

```
Create table Ranking
(Mnemonic string, Def Expr<bool>, Score
Expr<int>, Rank int)
```

Ranking

Population

Given the tables *Population* and *Ranking*, one can effortlessly extract relevant information by utilizing the *formula fields Def* and *Score*.

Aggregating the sum of all scores that satisfy the Def conditions:

```
Select Name, sum (Mnemonic + ‘ ‘), sum Score
from Population outer join Ranking on Def
```

Colbert automatically designates a default nomenclature for Calculated Fields like SumMnemonic.

Colbert features an extended orthogonal SQL known as Slim SQL, which is detailed in the upcoming blog post about Slim SQL.

Population with Ranking Mnemonic and Scores

Mnemonic for the match possessing the lowest rank.

```
Select Name, Rank, Mnemonic
from Population outer join first (Ranking by
rank)
on Def
```

*Colbert supports a **first** operator and allows sorting in subquery. Omitting **Select * from** is allowed and **order by** may be abbreviating to **by** .*

Colbert features an extended orthogonal SQL known as Slim SQL, which is detailed in the upcoming blog post about Slim SQL.

First match by Rank

A quick summary

forget the summary

In summary, Colbert proposes a straightforward approach to handling formulas stored in database fields, consisting of the following elements:

1. Introduction of a novel type, Expr<T>.
2. Implementation of a twofold type-check mechanism.

3. Invocation of the name of a *formula field* triggers the evaluation of the corresponding expression.
4. Parameter binding is determined by the name of the formal parameter in the expression.
5. The scope of formal parameters is defined by the location of invocation.

Nevertheless, there is no need to memorize these elements because the utilization of *formula fields* is intuitive and simple

Various aspects of formula fields

Is it all sunshine and rainbows ?

There are various other aspects related to executing and storing formulas in database fields. Here are some common aspects related to this topic:

- Best practice for **storing** formulas in database fields.
- How to **parse, type check, compile and execute** formulas stored in database fields.
- Implementing *formula field* calculations **in SQL queries**.
- **Security risks** and best practices for handling user-defined formulas.
- Is SQL injection a hazard when using *formula fields*?
- Optimizing **performance** of database queries involving *formula fields*.
- **Caching** strategies for formulas.
- Real-world **examples** of scenarios where storing and executing formulas in database fields is beneficial.
- **Existing approaches** to storing formulas in database fields.

Upcoming blog

Certain aspects have been addressed in the present blog. Subsequent discussions in a forthcoming blog will delve more extensively into several of the aforementioned subjects.

Videos

Experience the Magic of Formulaic Concepts with Colbert

There are some videos available so you can see how easily formulas can be applied in Colbert's versatile user interface.

Bonus calculation:

- Every employee has a different bonus formula.

Video (https://www.youtube.com/watch?v=ohk_qIG5ULY)

More examples

- Criteria matching: Buyers define what second hand car they are looking for.
- Risk analysis: A set of risk criteria is defined with corresponding risk scores.
- Cohort definition: A population is defined into cohorts according to a set of rules.

Video (https://www.youtube.com/watch?v=_Rn-7xzwFUY)

Contact

Colbert

Netherlands

info@colbert.nl(https://www.colbert.nl/mail-form)

Colbert (https://www.colbert.nl/)

About Me(https://www.colbert.nl/about)

Mail form(https://www.colbert.nl/mail-form)

Articles(https://www.colbert.nl/#Articles)