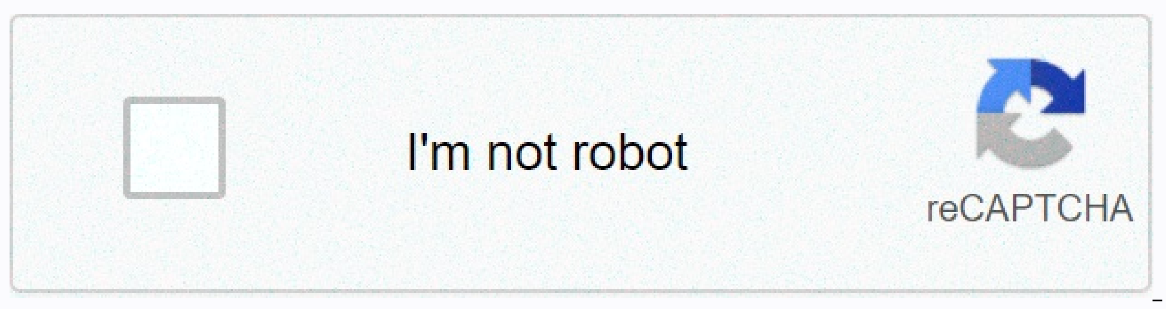
I'm not robot
reCAPTCHA

Arrays in redshift

Readers of this blog know that we love fast analytics apps. As we demonstrated in an earlier article, ClickHouse outpervised Redshift at a nyc taxi travel benchmark. However, reference queries are very simple. Real-life use cases often require much more complex data processing. It is especially true for fintech applications. This time the Paris-based startup, ICA, a capital market fintech, suggested an interesting case of using risk management that solved Clickhouse's use. ICA allows financial institutions to extract and analyze billions of rows of real-time data for Front Office and P&amp;L Risk, Market Risk and Credit, Liquidity Management, Market Data, etc. Your solution widely uses ClickHouse for all kinds of aggregations and statistics needed for risk calculation. We take the challenge and look at it from different angles, showing how ClickHouse can adapt to specific use cases unlike other traditional SQL databases like Redshift. For impatient readers, here's the punchline. CLICKHouse SQL extensions, arrays in particular, allow you to solve the business use case up to 100 times more efficiently than Redshift at 1/6 cost. We know ClickHouse is fast, but we were a little surprised by the results of the investigation. Use Case: FinTech Risk Management Consider a financial cube that is normally used for risk management purposes, e.g. profit and loss values (P&amp;L) or market-branded securities under stressed market conditions. These cubes can be quite large, containing hundreds of dimensions of different data types. Each record is associated with a measurement vector. Vectors are quite long, for example, 1000 elements. Typically, this cube will store for each financial transaction and for each date, a vector of P&amp;L values representing different market scenarios (calculated using story-based simulations or a Monte Carlo simulation). Different dimensions represent some of the business attributes (such as portfolio hierarchy, product type, or counterparty) that are required as potential aggregation levels or filtering criteria when performing risk calculation. The typical question we can ask with this data is as follows. What is the maximum loss in statistical terms that we can expect for each of my portfolios? These metrics are generally referred to as Risk Value, Expected Deficit, etc., in risk management terminology. First, we add the P&amp;L vectors for each portfolio in order to obtain a distribution of Then we can apply different statistical methods. For example, if we want to calculate the maximum loss in a 95% confidence interval (Value at risk), we can sort the resulting vectors of 1000 elements and take element 50 to get 5% quantile. Below is the diagram to illustrate the process: Since this data is usually private, we have generated it using a tool provided by ICA. It produces a dataset with 100 dimensions of different types represented as int0-int33, dttime0-dttime32, str0-str32 respectively, and P&amp;L vectors of 1000 elements. For benchmarking we used the same configuration as in the previous article: ClickHouse 20.6.4.44 running on Kubernetes within an AWS m5.8xlarge instance (32 vCPU 120 GB of RAM, Redshift on two dc2.8xlarge instances (2 x 32 vCPU 244GB RAM, $9.6 per hour). The dataset is small and the data is cached after the first run, so we are measuring the pure computational capabilities of both analytical DBMSs. ClickHouse Schema The most natural way to render vectors in ClickHouse are arrays. ClickHouse has beautifully developed functionality around arrays, including functions added in arrays, lambda expressions, and more. Arrays allow you to avoid additional tables and joins when there is a 1 to N relationship between entities. In our case each cube cell 'stores' a vector of values, so we can have a reference to a separate table, or store the vector in an array. We'll start with the array approach, but later we'll examine the stand-alone table as well, as we need to do it for the Redshift schema anyway. Therefore, we have the following schema in ClickHouse (see repository for a complete set of schema and query examples): CREATE TABLE factTable (index Int64, int0 Int64, ... , int33 Int64, dttime0 DateTime, ... , dttime32 DateTime, str0 String, ... , str32 String, arrFloat Array(Float32), partition6) ENGINE ? MergeTree PARTITION BY partition ORDER BY tuple() The table has the following columns: Index — unique row ID Int0-int33 , dttime0-dttime32, str0-str32 are dimension columns arrFloat — vector partition P&amp;L — partition column, it is usually a day No primary key and order, since this is just a cube. We loaded the table with dataset_gen.py. Generated 1,72M rows taking 6.8GB of storage. Once you're ready, let's take a look at the test queries suggested by our ICA friends. Queries The basic question we want to answer is the maximum loss for a portfolio with a certain confidence level. ClickHouse can add arrays, adding elements position by position with the 'sumForEach' function. This is an aggregate function that takes an array column as an argument and results in an array, where the 1st element is a sum of 1st element of all arrays in the group, 2nd element is a sum of 2nd elements of all arrays in the group, and so on. In order to calculate the quantums for the values stored in an array of a known size, we can simply sort a and take a corresponding element. Since we have 1000 elements, 5% quantile is element 50. Q1. Maximum loss with 95% confidence (5% quantile) for a single dimension. SELECT str0, arraySort(sumForEach(arrFloat))[50] AS arr1 FROM factTable GROUP BY str0 Q2. Maximum loss with 95% confidence (5% quantile) for a group of 6 6 SELECT str0, str1, int10, int11, dttime10, dttime11, arraySort(sumForEach(arrFloat))[50] AS arr1 FROM factTable GROUP BY str0, str1, int10, int11, dttime10, dttime11 Q3. Maximum loss with 95% confidence (5% quantile) for a 12-dimensional group. SELECT str0, str1, str2, str3, int10, int11, int12, int13, dttime10, dttime11, dttime12, dttime13, arraySort(sumForEach(arrFloat))[50] AS arr1 FROM factTable GROUP BY str0, str1, str2, str3, int10, int11, int12, int13, dttime10, dttime11, dttime12, dt1time. Query with a filter and display the results in rows. SELECT str0, num, pl FROM (SELECT str0, sumForEach(arrFloat) AS arr1 FROM factTable WHERE str1 ? 'KzORBHFRuFFOQm' GROUP BY str0) ARRAY JOIN arr1 AS pl, arrayEnumerate(arr1) AS num Note the ARRAY JOIN clause in the last query. Used to display multiple arrays of the same size in rows. It is often used in conjunction with the arrayEnumerate function that generates a list of indexes (1,2,3 ...) for a given array. ClickHouse can also calculate quantities in arrays directly, so Q1 can be executed as follows: SELECT str0, arrayReduce('quantilesExact(0.05)',sumForEach(arrFloat)) AS arr1 FROM factTable GROUP BY str0 Here we are using the arrayReduce function that comes from functional programming. Applies the quantileExact aggregate function to array elements. Results of both approaches are shown in the following table (query time in seconds): See ClickHousem5.8xlarge arraySort ClickHousem5.8xlarge arrayReduce Data size 6.83GB 6.83GB Q10.73 0.72 Q2 1.72 1.04 Q3 2 1.05 Q4 0.45 0.45 Note that the arrayReduce version is faster! The difference is because arrayReduce is vectorized, while arraySort is not! Going Without Arrays Not all databases support arrays, so considering Redshift we decided to test it with a more traditional approach in ClickHouse first. Delete the arrFloat column and place a vector of values in a separate table as rows instead: CREATE TABLE factTable_join as factTable; ALTER TABLE factTable_join DROP COLUMN arrFloat; CREATE TABLE factCube (index Int64, position Int16, Value Float32) ENGINE to MergeTree ORDER BY index; To calculate quantums without arrays, SQL analytical functions such as PERCENTILE WITHIN GROUP or window functions are traditionally required. At the time of writing, ClickHouse does not support them. However, it has some other features that do the job, and they do it in a more efficient way: quantileExact aggregate function that we use with already unique ClickHouse feature arrays: LIMIT BY. This SQL extension allows you to return N rows for the group, with an optional offset. So we'll use the syntax 'LIMIT 49, 1 &lt;group_columns&gt;BY', which will return the 50 in a group. For example, query Q1 can be rewritten as follows with quantileExact: SELECT str0, quantileExact(0.05)(val) FROM (SELECT &lt;/group_columns&gt; &lt;/group_columns&gt; sum(value) AS val FROM factCube INNER JOIN factTable_join USING (index) GROUP BY str0, position) GROUP BY str0 ORDER BY str0 The same query using LIMIT BY is more compact: SELECT str0, sum(value) AS val FROM factCube INNER JOIN factTable_join USING (index) GROUP BY str0, position ORDER BY str0, val LIMIT 49, 1 BY str0 We have tested the performance of those two queries , and did not find a noticeable difference: most of the time is invested in dimension aggregation and position column, and this part is the same in both cases. Therefore, we will use a syntactically more compact version of LIMIT BY. Q2 and Q3 can be rewritten in a similar way, and Q4 will look like this: SELECT str0, position, sum(value) AS val FROM factCube INNER JOIN factTable_join USING (index) WHERE str1 to 'KzORBHFRuFFOQm' GROUP BY str0, position With this schema and queries we achieved significantly worse performance compared to arrays: Query ClickHousem5.8xlarge arraySort ClickHousem5.8xlarge arrayReduce ClickHousem5.8xlarge join data size 6.83GB 6.83GB 7.23GB Q1 0.73 0.72 14 Q2 1.72 1.04 60 Q3 2 1.05 99 Q4 0.45 0.45 1.33 There is another approach if arrays are not available. We can put all values in a simple table as rows. The table definition must be modified: CREATE TABLE factTable_plain (index Int64, int0 Int64, ... , int33 Int64, dttime0 DateTime, ... , dttime32 DateTime, str0 String, ... , str32 String, Int64 partition, Float32 value, position Int16) ENGINE - MergeTree PARTITION BY partition PRIMARY KEY tuple() ORDER BY (int0, ... , str32, position) Since the number of rows is multiplied by a size, the new table contains 1000 times more rows reaching 1.7 billion! In order to store it effectively we put all dimensions to the ORDER BY. This groups the same values for each dimension column. ClickHouse demonstrates a remarkable compression ratio of 95x in this case! Tenga en cuenta que no aplicamos ningún códec en absoluto, ClickHouse utilizó la compresión predeterminada de LZ4: ┌-tabla-----------┬--suma(filas)-┬-sin comprimir-┬-comprimido-comprimido-┬-relación-┬-tamaño---┬-piezas-┐ - factTable - 1720000 - 8306992324 - 7332770292 - 1.13 - 6.83 GiB - 37 factTable_plain 6992324 - 1720000000 á 1423552324000 á 14839550571 - 95,93 - 14,31 GiB - 66 - └ - , á , ┴ , ┴ , ┴ , ┴ Las consultas ┴-┴-┴-------┴ ┘ son muy similares a lo que teníamos antes, pero sin unirse , but without joining, but without joining, but without joining, but without joining the union without joining, for example, Q1 looks like: SELECT str0, sum(value) AS val FROM factTable_plain GROUP BY str0, position ORDER BY VAL LIMIT 49, 1 BY str0 Query ClickHousem5.8xlarge arraySort ClickHousem5.8xlarge arrayReduce ClickHousem5.8xlarge join ClickHousem5.8xlarge plain Data size 6.83GB 6.83GB 7.23GB 13.82GB Q1 0.073 0.72 14 10 Q2 1.72 1.04 60 48 Q3 2 1.05 99 77 Q4 0.45 0.45 1.33 2.1 Performance is slightly better than the table version joined at a cost of 2 times the expansion data. Testing Redshift Redshift does not support arrays, so we tested the same arrayless approaches as before: with a JOIN table and a raw table without JOIN. The table structure in Redshift is similar to ClickHouse, we just had to change the data types that are slightly different between two databases. Here is a table definition for the JOIN case (see repository for complete examples): CREATE TABLE IF NOT EXISTS factTable (index BIGINT PRIMARY KEY, int0 BIGINT, ... , int33 BIGINT, dttime0 timestamp, ... , dttime32 timestamp, str0 varchar(20), ... , str32 varchar(20), partition BIGINT) DISTKEY(index); CREATE TABLE IF NOT EXISTS factCube (index BIGINT, position SMALLINT, value FLOAT4, PRIMARY KEY (index, position)) DISTKEY (index) SORTKEY (index, position); We didn't apply any Redshift-specific codecs as well, and we were surprised to see that Redshift couldn't use any defaults effectively. The size of the data was 3-5 times larger than the ClickHouse in the worst case, but small enough to be effectively cached. Without practical ClickHouse feature queries they had to be rewritten in a more traditional way using window functions. For example, Q1 looks like this: SELECT str0, val FROM (SELECT str0, sum(value) as val, row_number() OVER (PARTITION BY str0 ORDER BY val) as the FROM factTable INNER JOIN factCube USING (index) GROUP BY str0, position) WHERE number a 50; Q2 and Q3 are very similar with more columns in the PARTITION BY and GROUP BY sections. The Q4 is exactly the same as in ClickHouse. We also tested the aggregate function PERCENTILE_DISC, but we couldn't get reasonable performance. All query times are summarized in the following table: Query ClickHousem5.8xlarge arraySort ClickHousem5.8xlarge arrayReduce ClickHousem5.8xlarge join ClickHousem5.8xlarge plain Redshiftdc2.8xlarge x2 join Redshiftdc2.8xlarge x2 Simple Data Size 6.83GB 6.83GB 7.23GB 13.82GB 40.06GB 62.00GB Q1 0.73 0.72 14 10 4 4 Q2 1.72 1.04 60 48 112 1 14 Q3 2 1.05 99 77 200 200 Q4 0.45 0.45 1.33 2.1 6.6 2 As we can see , ClickHouse with arrays significantly exceeds Redshift in all queries. It's 100-200 times faster for Q2 and Q3! The data stored in ClickHouse is also very compact, which takes up 6 times less disk space than in Redshift. This is very important at scale. But even if we decide not to ClickHouse arrays for some reason and using other SQL functions instead, Redshift is still far behind. Interestingly, Redshift shows very little difference between the join and simple table approach. The analysis would be if we don't mention the cost. The ClickHouse instance costs us only $1.54/hour on AWS, while for Redshift Amazon it charges $9.6/hour.   Conclusion This project was an interesting investigation for us. It clearly demonstrated that ClickHouse's performance is not only driven by well-optimized code. Actual speed can be achieved when using specialized functions and data types, arrays in particular. Arrays are very powerful for calculations and make ClickHouse extremely efficient in applications where data is naturally represented as vectors. Statistical analysis in finance is a good example. ClickHouse was able to calculate complex P&amp;L vectors 100 times faster than Redshift at 1/6 price! We would like to thank our ICA colleagues for raising the risk management issue explored in this blog post. (Let's check them if you are a financial user!) We'd also like to challenge Redshift experts to see if they can beat ClickHouse in this use case as well as others. Friendly competition helps everyone learn. Finally, we praise ClickHouse arrays, but we don't discuss them in sufficient detail for such an important feature. Let's fill the void soon with another blog post. Stay tuned! Tuned!