2024년 09월 06일

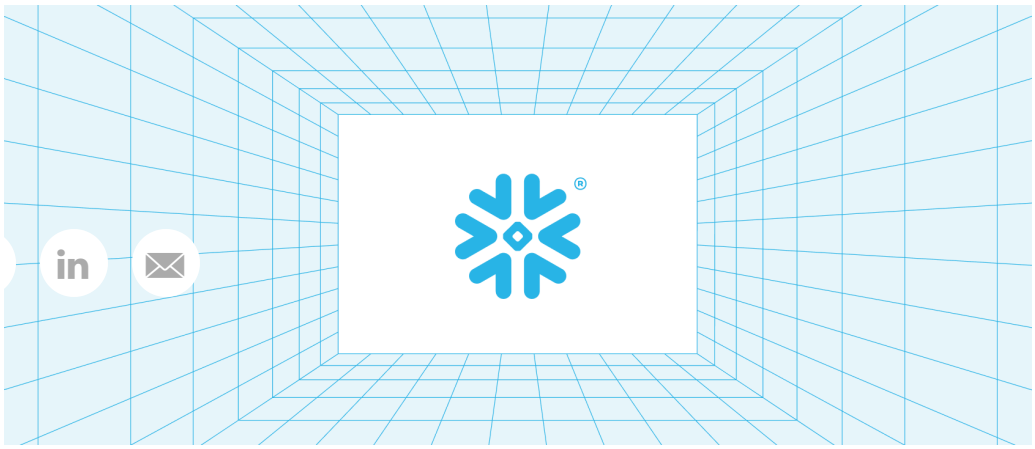
AUTHOR

Bowei Chen

SHARE

Aggregation Placement — Technical Deep-dive and Road to Production

Core Platform



In the previous blog, we presented a query optimization in the query processing layer of Snowflake: aggregation placement. Aggregation placement improves Snowflake's query engine's ability to process complex analytical queries that consist of aggregations and many joins. We illustrated how this optimization improves such workloads and showed that the optimization is being applied widely to Snowflake's production queries. This blog post is for engineers interested in the technical aspects of aggregation placement and the overall software development cycle at Snowflake.

The key differentiator of Snowflake's aggregation placement implementation is that we do not rely on compile-time statistics, which could often be very inaccurate, to decide the final execution strategy. Instead, we allow our query execution engine to adapt. The adaptiveness enables the optimization to be applied wherever it is beneficial without causing performance regressions. Also, the rewrite framework is designed to be generic enough to handle arbitrary aggregation functions and join types. In the first part of this blog post, we are going to discuss the implementation of query plan transformation and the runtime adaptive query execution strategy.

Rolling out aggregation placement to customers is challenging. Aggregation placement involves changes in both the optimizer and the query execution layer. It interacts with various components in the system and can be applied to a considerable amount of production queries. A testing and rollout plan, which we will also discuss in this blog post, is followed to allow aggregation placement to be enabled for customers safely and gradually. The rollout plan was taken as an example of rolling out complex features and inspired the standard rollout template for Snowflake's query processing features.

AUTHOR

Bowei Chen

Aggregation Placement Transformation

Aggregation placement transformation is implemented via a set of rewrite rules where the query optimizer first identifies aggregations that can be pushed below joins and adds additional aggregations below the joins in the query plan. Whether to actually apply the pushed-down aggregations are decisions made at runtime. If an aggregation can be pushed down multiple joins, we always push to the deepest possible position in the join tree. This is different from the traditional approach to aggregation placement, where the optimizer would consider alternatives and pick one, which execution is stuck with.

The rewrite rules for aggregation placement are applied in a query plan transformation phase after join orders are determined in the optimizer. In this section, we will go through the rewrite rules and discuss how they are designed to be generic for different aggregate functions and different join types.

Properties of Aggregate Functions

An aggregation might contain multiple aggregate functions. Each aggregate function could be pushed to either the left or right side of a join independently. The properties of aggregate functions are considered individually when we rewrite the query plan.

Firstly, all of the aggregate functions in an aggregation need to possess the two following properties to be pushed below a join:

> Splitability: the aggregate function's input attributes come from only one side of the join.
>
> Decomposability: An aggregate function **agg** is decomposable if there exist aggregate functions **agg_parent** and **agg_child** such that **agg(X) = agg_parent(agg_child(X))**. For example, aggregate function **count** could be decomposed into **sum** and **count**.

AUTHOR

Bowei Chen

Another aggregate function property that is relevant to aggregation placement transformation is duplicate sensitivity. An aggregate function is duplicate-agnostic if the results of the function do not depend on whether there are duplicate values in its input. Examples of duplicate-agnostic functions are **min**, **max,** and **any_value**. On the other hand, if the results of the aggregate function will be sensitive to multiple occurrences of the same values, the function is duplicate-sensitive. Examples of duplicate-sensitive functions are **sum**, **count**, and **avg**.

Duplicate-sensitive aggregate functions need special handling if the aggregation could be pushed below both sides of a join. When a duplicate sensitive function **agg** is pushed to one side of the join (as **agg_child**), we need to separately maintain a **count(*)** column in the aggregation that is pushed to the other side of the join. We preserve the group cardinality information because the additional aggregation **agg_child** can reduce duplicate values in the join results. Finally, a duplicate-sensitive implementation of the original function processes results for the aggregation using the pushed-down aggregate function column and the **count(*)** column. The query transformation technique for duplicate-sensitive aggregates is discussed in various papers which this is a summary of. Interested readers can find more information in the linked paper.

Our rewrite framework could support an aggregate function if its decomposed functions and potentially its duplicate-sensitive version are implemented. One can easily add support for a new or existing function without making changes to the rewrite framework.

In the next section, we are going to use an example query to demonstrate how the query rewrite is applied to inner joins. Outer joins and semi-joins follow similar rules but have some slight differences in their join-type-specific logic. We will not discuss them in the blog.

Query Plan Rewrite

The following diagram is an example of eager aggregation placement transformation being applied to a query plan. The query returns the maximum age of the enrolled students and the sum of enrollment fees in each class.

AUTHOR

Bowei Chen

Transformation rule: pushing aggregate functions to the left side of an inner join input

The original query plan on the left has an aggregation above a join on two table scans **Student** and **Enrollment**. The group-by key is the **class_id** and the join key is the **student_id**. In the rewritten plan on the right, aggregate functions from the two tables are pushed below the join, which results in two aggregations, each above a table scan.

On top of the table scan of **Enrollment**, **AggChild1** has both the join key **student_id** and the group-by key **class_id** as its group-by key. It also ha[illegible] st[illegible] f **enrollment_fee**, which is pushed from the parent aggregation. In **AggChild2** which is on top of the table scan of **Student**, since there's no group-by key in the parent aggregation that is from **Student**, its group-by key is just the join key **student_id**. The two aggregate functions in **AggChild1** are the maximum of **age** and an extra **count(*)** as a grouping count, which is required for result correctness. There is no change in the join.

In the parent aggregation, the group-by key is the same **class_id** as it is in the old plan while the two aggregate functions are:

1. A maximum function on the results of the child maximum function
2. A duplicate-sensitive sum function that takes the sum of the enrollment from **Enrollment** and the grouping count from **Student** as input which is adding up the product of the sum results from **AggChild2** and the grouping count from **AggChild1** of each row.

Generally speaking, when applying the transformation, group-by keys, join keys, and aggregate functions are pushed below joins. If new aggregations are added to both sides, a grouping count column is required in the child aggregation on any side for result correctness if there are duplicate-sensitive aggregate functions pushed to the other side. After an aggregation is pushed below a join, the aggregation might appear on top of another join, for which we would apply the same rewrite logic to further push the aggregation down.

Runtime Adaptiveness

The transformation rules ensure we could add aggregation nodes in the query plan whenever possible. Whether the additional pushed-down aggregations are actually active is decided during query execution individually.

Snowflake's query engine is implemented using a push-based execution model. Batches of data flow through operators in a query plan in a way that each operator pushes the results to the downstream operators. Multiple operators could form an execution pipeline in which data could be processed without materializing intermediate results. The child aggregations produce partial aggregation results that can be pushed to the downstream operators before termination, so they form an execution pipeline with the downstream joins, other pushed-down aggregations, and the parent aggregation.

AUTHOR

Bowei Chen

Aggregation placement adapts by checking the runtime statistics in each execution pipeline. We collect statistics like the number of rows and the

number of distinct values for group-by keys during query execution. A runtime cost model based on these statistics is used to determine whether each pushed-down aggregation should be enabled.

The following diagrams illustrate how the child aggregations adapt in different scenarios. In each diagram, a child aggregation is added to the join's right input. The pipeline consists of the child aggregation, the join, and the parent aggregation. For each example, we compute a cost with the child aggregation enabled and another cost with it disabled. For simplicity, the cost of each operator is its input row count in our illustrations.

In the first example, the child aggregation reduces its output rows to 10% of its input rows. The join is exploding, producing 10x rows compared to its right input. When child aggregation is enabled, it makes the join and parent aggregation 90% cheaper, reducing the total cost of the pipeline. Because the child aggregation is considered beneficial, it would be kept enabled during query execution.

Adaptiveness: cheaper when the child aggregation is enabled

Although the child aggregation could group multiple input rows into one and make the downstream pipeline cheaper to process, processing itself adds an additional overhead to the pipeline and can make the pipeline more expensive overall when the overhead outweighs the savings.

In the following example, the child aggregation reduces its output rows to 70% of the input rows. The join does not explode, producing the same output rows as its input. The child aggregation makes the downstream pipeline a little cheaper, but the total cost is higher compared to the cost when it is disabled. When the child aggregation detects this is the case, it dynamically disables itself to prevent performance degradation.

AUTHOR

Bowei Chen

Adaptiveness: more expensive when the child aggregation is enabled

Road to Production

One of Snowflake's core values is to put customers first. We always pay attention to make sure features are delivered to production safely and transparently. We utilized different automation tools to prevent customers from experiencing wrong results, query failures, and performance regressions when their queries run with newly developed features. In this section, we walk through aggregation placement's road to production as an example of Snowflake's feature rollout process, which shows our dedication to providing the best user experience to our customers.

Pre-Production Testing Pipelines

The first automation tool to catch any potential issues is our regression test pipeline where unit tests and SQL tests are added to when we commit features or bug fixes. We add a set of comprehensive tests for aggregation placement that cover a variety of basic and corner cases for the functionalities we supported. The first step to rolling out a feature is to enable it in the pipeline that runs all the regression tests we have and make sure none of the new or existing test cases fail.

Query rewrites result in plan changes, which could lead to query failures or wrong results issues if a rewrite is not done properly. To cover as many query plan shapes as possible and gain confidence before rolling new

features to customers, we enabled the feature in our QA test pipeline where we run randomly generated queries that enumerate various shapes of query plans and different SQL features on a daily basis. Aggregation placement was enabled in our QA pipeline for months and we only considered moving forward with the rollout after there are no outstanding or new issues in the pipeline for over a month.

AUTHOR

Bowei Chen

Regression tests and the QA test pipeline cover many potential failures and wrong results issues. Making sure new features do not introduce performance regressions is another really important aspect in successfully delivering them to customers. For aggregation placement, we focus on ensuring the run-time cost model is accurate in that aggregation placement would be enabled only when it is improving query performance. We implemented a set of atomic queries with different performance characteristics. These queries usually have one to two joins and an aggregation on top which can be pushed down. The number of distinct values for the join keys and group-by keys varies for different queries. We run the atomic queries to make sure the cost model always works as expected. Afterward, they are added to a test suite to run with new commits periodically to prevent regressions over time.

TPC-DS Benchmark

TPC-DS is a standard benchmark targeted for various analytical scenarios. Benchmark performance is sometimes a good indicator of how impactful a performance feature is for real-world workloads. In addition, the benchmark could be utilized to detect potential performance regressions for new features. After making sure the cost model works as expected with the atomic tests, we started benchmarking aggregation placement on the TPC-DS 10TB dataset. When we ran the benchmark for the first time, we got the results as shown in the following figure.

Each blue bar is the ratio between the runtime with aggregation placement enabled and disabled for one of the 99 queries in TPC-DS. The performance of a single query is better if the blue bar is lower. A query regresses if its blue bar's value is more than 1 (taller than the yellow bars). On the right side of the chart, some queries were gaining up to 3x improvements, which is fairly decent. However, on the left side, we also identified that more than 10 queries had noticeable degradation.

One of our first priorities is to “do no harm” to customer workloads. The performance degradation in the initial testing was because the overhead of the query rewrite was greater than expected and our runtime cost model was not able to identify these cases. We made improvements to address the issues and kicked off another run of the benchmark. The end result is that we don’t always get as much improvement as we could, but in exchange for that, we do no harm.

AUTHOR

Bowei Chen

Production Test

Snowflake’s cloud-native architecture opens up the opportunity to test customer queries on a larger scale before rolling out new features. The automation testing framework Snowtrail is a powerful tool that could be configured to test production workloads of different characteristics to identify issues that could affect feature rollouts. We instrumented very detailed telemetry to detect scenarios where the optimization could be applied, so we can comprehensively perform tests internally on a variety of customer workloads in different accounts to detect any possible issues that we could hit in the feature rollout process.

The first run is on a large set of queries that could be impacted to detect potential wrong results or failures. The focus is the correctness of our implementation. We finally tested with more than 20,000 queries, which we selected to maximize coverage across different customer accounts and different query performance characteristics. We proceed to the next step after the Snowtrail run succeeds with no failures or wrong results.

Snowtrail also supports a performance run mode, where each query is run multiple times under different configurations, to compare the query performance of those configurations. The diagram below shows the results of one performance run. In this run, we targeted long-running queries where execution is dominated by aggregations and joins so they are most likely to be impacted by aggregation placement. The workload consists of more than 150 such production queries. As in the previous section, the lower the blue bar, the better the improvement in performance for a single query.

AUTHOR

Bowei Chen

The results show that aggregation placement is able to improve query performance significantly on some queries without introducing performance regressions on other queries. One nice finding in the run is aggregation placement usually makes query processing much faster when there are exploding joins that often degrade query performance drastically. The way we implemented aggregation placement enables it to improve the stability of query execution out of the box. In a couple of queries, the chart shows an increase in runtime (blue bars above the yellow). These are not performance regressions caused by the feature, but some natural variance of execution time in our experiments.

Feature Rollout

Aggregation placement gets applied for every 1 in 5 production queries. The rollout plan for aggregation placement was designed to limit the impact and blast radius of any potential problems in our production systems. Over the course of 3 months, we gradually enabled the feature every week for a new portion of our customer accounts. We built a dashboard to monitor the feature rollout, making sure it does not break important health metrics of the system. Finally, aggregation placement is enabled in all production accounts after a nearly half-year effort of thorough testing and delivery.

Conclusion

In this blog, we went into the technical details of aggregation placement. Snowflake's aggregation placement implementation leverages adaptive query execution techniques to provide the best out-of-the-box performance to customers. The query optimization could be applied wherever it may improve query performance and we adapt during query execution to avoid performance regressions. We also showcase the testing and feature rollout process for aggregation placement as they represent the standard of delivering new features at Snowflake. The attention to detail we put into customer experience has made Snowflake's data platform an industry leader in out-of-the-box query performance and enterprise-class stability.

References

1. Including Group-By in Query Optimization

2. Eager Aggregation and Lazy Aggregation

SHARE

AUTHOR

Bowei Chen