overcast blog

13 Ways to Optimize Databricks Queries

DavidW (skyDragon) Follow

22 min read · Aug 6, 2024

Effective query optimization in Databricks is vital for several reasons. Firstly, it directly impacts the speed at which data can be processed and analyzed, which is critical in data-intensive industries where timely insights are essential. Optimized queries reduce computational overhead and resource consumption, leading to lower operational costs. Additionally, well-optimized queries improve the overall user experience by providing faster response times, which is particularly important in interactive data analytics environments.

When to Use Query Optimization Techniques

Query optimization should be an ongoing process, especially in environments where data volumes and workloads fluctuate. It is particularly important during the development and deployment of new data pipelines, as well as during major data migrations or schema changes. Optimization is

also crucial in response to performance monitoring, where identifying and addressing bottlenecks can lead to substantial improvements in efficiency.

Key Challenges with Query Optimization

One of the primary challenges in query optimization is managing data skew, where an uneven distribution of data leads to inefficient query execution. Another challenge is selecting the appropriate optimization techniques and configurations, as these can vary significantly depending on the specific characteristics of the workload and data. Additionally, maintaining optimal configurations requires continuous monitoring and adjustments, particularly in dynamic environments where data and query patterns can change frequently.

Choosing a Solution

Choosing the right optimization strategies involves understanding the specific requirements and constraints of your data environment. Factors to consider include the size and complexity of the datasets, the types of queries being executed, and the performance characteristics of the underlying infrastructure. For instance, leveraging features like Adaptive Query Execution (AQE) and the Photon engine can provide significant performance benefits in scenarios involving large-scale data analytics and machine learning workloads. Similarly, using Delta Lake optimizations such as Z-Ordering and Auto Optimize can improve query efficiency in environments with frequent data updates.

In this guide, each optimization technique is explored in detail, providing practical steps, best practices, and additional resources to help you implement these strategies effectively. Whether you are managing ETL pipelines, interactive analytics, or machine learning workflows, optimizing

your Databricks queries will enable you to harness the full potential of your data infrastructure.

1. Use Adaptive Query Execution (AQE)

Adaptive Query Execution (AQE) is a dynamic optimization feature in Apache Spark that adjusts query execution plans at runtime based on the actual data characteristics encountered. Introduced in Spark 3.0, AQE helps in improving query performance by adapting strategies such as join selection, partition coalescing, and handling skewed data. AQE works by collecting runtime statistics and using them to re-optimize the query execution plan as the query progresses.

How to Use Adaptive Query Execution

To enable AQE in Databricks, you can set the configuration parameter `spark.sql.adaptive.enabled` to true. This enables several sub-features, such as dynamic partition coalescing, adaptive join selection, and skew handling. For example, to enable AQE and configure it to coalesce small partitions, you can use the following settings:

```
SET spark.sql.adaptive.enabled = true;
SET spark.sql.adaptive.coalescePartitions.enabled = true;
SET spark.sql.adaptive.coalescePartitions.minPartitionSize = '1MB';
```

When to Use Adaptive Query Execution

AQE is particularly beneficial when dealing with unpredictable data patterns, large datasets, and complex queries with multiple joins and aggregations. It is ideal for scenarios where data distributions are unknown or can vary significantly, such as in data lakes or mixed workloads. AQE

helps optimize resource utilization and reduces query latencies by dynamically adjusting the execution strategies based on actual runtime metrics.

Best Practices for Using Adaptive Query Execution

Enable Auto-Optimized Shuffle: Set `spark.sql.shuffle.partitions` to 'auto' to allow AQE to dynamically adjust the number of shuffle partitions based on the size of the input data. This setting helps balance the load across tasks and avoid skewed data distribution.

Monitor and Adjust Skew Join Handling

Use the parameter `spark.sql.adaptive.skewJoin.enabled` to enable skew join detection and optimization. This feature identifies skewed partitions and splits them into smaller, more manageable subpartitions, thus improving performance. For instance:

```
SET spark.sql.adaptive.skewJoin.enabled = true;
SET spark.sql.adaptive.skewJoin.skewedPartitionFactor = 5;
SET spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes = '256MB';
```

Dynamic Join Re-Optimization

AQE can dynamically switch join strategies, such as converting a planned sort-merge join to a broadcast hash join if runtime data indicates it would be more efficient. This flexibility is crucial in optimizing join operations based on the actual sizes of the data partitions involved.

Further Reading

- Detailed AQE Overview: https://docs.databricks.com/spark/latest/spark-sql/aqe.html

- Performance Tuning with AQE: https://www.databricks.com/blog/adaptive-query-execution
- AQE Configuration and Tuning: https://learn.microsoft.com/en-us/azure/databricks/spark/latest/spark-sql/aqe

2. Leverage the Photon Engine

Photon is a high-performance, native vectorized query engine developed by Databricks to enhance the execution of SQL and DataFrame workloads. Built from the ground up in C++, Photon leverages modern CPU architectures, providing significant performance improvements over traditional Spark execution. It is fully compatible with Apache Spark APIs, allowing seamless integration with existing workflows. Photon accelerates various operations, including aggregations, joins, and data transformations, making it ideal for data-intensive tasks.

How to Use the Photon Engine

Photon is enabled by default on Databricks clusters running Databricks Runtime 9.1 LTS and above. To manually enable or disable Photon, you can select or deselect the “Use Photon Acceleration” checkbox when creating or editing a cluster. For example, when configuring a cluster, ensure that the Photon acceleration option is checked to benefit from its optimizations:

```
CREATE CLUSTER [CLUSTER_NAME]
SET runtime_engine = 'PHOTON';
```

When to Use the Photon Engine Photon is particularly beneficial in scenarios requiring high throughput and low-latency data processing. It

excels in handling large-scale data analytics workloads, such as ETL pipelines, complex queries with heavy aggregation, and AI/ML model training that involve significant data manipulation. Photon's architecture is optimized for operations on Delta Lake and Parquet tables, making it ideal for environments that frequently process large datasets stored in these formats.

Best Practices for Using the Photon Engine

1. **Optimize Resource Utilization**: While Photon provides substantial performance gains, it may consume more Databricks Units (DBUs) compared to non-Photon clusters. It's important to balance performance improvements with cost considerations. Monitoring DBU consumption and adjusting cluster configurations accordingly can help manage costs effectively.
2. **Leverage Photon's Vectorized Processing**: Take advantage of Photon's vectorized execution by structuring workloads to maximize parallel processing. This can be particularly effective for batch operations and large analytical queries.
3. **Use with Delta Lake**: Photon's optimizations are most effective when used with Delta Lake tables. Ensure that data is stored in Delta Lake format to benefit from enhanced performance in read, write, and merge operations.

Further Reading

- Databricks Photon Overview: https://docs.databricks.com/spark/latest/spark-sql/aqe.html
- Enhancing Performance with Photon: https://www.databricks.com/blog/adaptive-query-execution

- Understanding Photon in Databricks: https://learn.microsoft.com/en-us/azure/databricks/spark/latest/spark-sql/aqe

3. Implement Dynamic File Pruning (DFP)

Dynamic File Pruning (DFP) is an optimization technique in Databricks that significantly improves query performance by reducing the amount of data read during query execution. It operates by dynamically pruning unnecessary files from being scanned based on filter conditions applied at runtime. This technique is particularly effective when dealing with Delta Lake tables, as it allows the query engine to skip over irrelevant files, thereby reducing I/O operations and speeding up query times.

How to Use Dynamic File Pruning

DFP is enabled by default in Databricks Runtime 6.1 and higher. It is automatically applied to queries that involve Delta Lake tables and utilize certain join types, such as INNER and LEFT-SEMI joins. The configuration can be adjusted using specific Spark configuration settings. For example:

```
SET spark.databricks.optimizer.dynamicFilePruning = true;
SET spark.databricks.optimizer.deltaTableSizeThreshold = '10GB';
SET spark.databricks.optimizer.deltaTableFilesThreshold = 100;
```

These settings enable DFP and determine the conditions under which it is activated, such as the size of the table and the number of files it contains. The `deltaTableFilesThreshold` setting, for instance, specifies the minimum number of files required for DFP to trigger, helping to avoid overhead for smaller tables where pruning would not be beneficial.

When to Use Dynamic File Pruning

DFP is particularly useful in environments with large datasets, especially when queries involve complex join operations or are executed on tables with high file counts. It is most beneficial for queries that include filter conditions on non-partitioned columns or are part of a star schema where filter predicates are pushed from dimension tables to fact tables. This optimization is crucial in scenarios where minimizing data scan volume is necessary for performance reasons.

Best Practices for Using Dynamic File Pruning

To maximize the benefits of DFP, it is advisable to:

1. **Use Photon-Enabled Compute:** DFP works optimally with Photon, especially for operations like `MERGE`, `UPDATE`, and `DELETE`, where filtering needs to be highly efficient.
2. **Combine with Data Clustering Techniques:** Techniques like Z-Ordering can enhance the effectiveness of DFP by organizing data in a way that makes file skipping more efficient. This is especially important when filtering on multiple columns.
3. **Adjust Configuration Parameters:** Fine-tune the DFP settings based on your specific workload and data characteristics. For instance, lowering the `deltaTableFilesThreshold` can be useful in scenarios with smaller files.

Further Reading

- Dynamic File Pruning Overview: https://learn.microsoft.com/en-us/azure/databricks/optimizations/dynamic-file-pruning
- Comprehensive Guide to Optimize Data Workloads: https://www.databricks.com/blog/optimize-data-workloads

- Detailed Explanation of Dynamic File Pruning: https://databricks.com/blog/faster-sql-queries-on-delta-lake-with-dynamic-file-pruning

4. Utilize Delta Cache

Delta Cache, also known as disk caching, is a feature in Databricks that accelerates data access by storing copies of remote data files on local storage attached to the compute nodes. When a file is first read from a remote location, Delta Cache creates a local copy using a fast intermediate format. Subsequent reads are performed locally, significantly reducing latency and improving performance. This caching mechanism is automatically applied to Parquet and Delta Lake tables, enhancing the speed of data-intensive operations such as joins, aggregations, and filtering (Databricks documentation) (MS Learn) (Databricks).

How to Use Delta Cache

Delta Cache is enabled by default on clusters running Databricks Runtime. However, for fine-grained control, you can use the `CACHE SELECT` command to specify which data to cache. For example:

```
CACHE SELECT column1, column2 FROM my_table WHERE column3 = 'value';
```

This command caches the specified columns and rows, allowing subsequent queries to access cached data instead of reading from the source. In Databricks Runtime 14.2 and above, the `CACHE SELECT` command is ignored, as an enhanced disk caching algorithm is used automatically.

When to Use Delta Cache

Delta Cache is particularly beneficial for workloads that involve repeated access to the same data. It is ideal for interactive and BI workloads where low-latency data access is critical. This caching mechanism is also useful in scenarios involving complex analytical queries, as it reduces the overhead of reading data from remote storage multiple times (Databricks documentation).

Best Practices for Using Delta Cache

1. **Choosing the Right Instance Types:** Use instance types with local SSDs for optimal performance, as these provide the necessary local storage for caching. Databricks automatically configures these instances to utilize Delta Cache efficiently.
2. **Monitoring and Managing Cache Usage:** Regularly monitor the usage and effectiveness of the cache. Since the cache automatically manages its contents, including invalidation when the underlying data changes, ensure that your infrastructure can handle the required cache size. This can be configured using parameters like `spark.databricks.io.cache.maxDiskUsage` to control the amount of disk space allocated for caching.
3. **Optimizing Cache Configuration:** If specific queries or datasets are critical for performance, consider using the `CACHE SELECT` command to ensure they are cached. Additionally, understand that when a cluster is terminated or scaled down, the cache data is lost and must be repopulated upon reinitialization (MS Learn).

Further Reading

- Databricks Documentation on Caching: https://docs.databricks.com/optimizations/delta-caching.html

- Disk Cache Overview: https://learn.microsoft.com/en-us/azure/databricks/sql/language-manual/delta-cache
- Caching Best Practices: https://www.databricks.com/blog/faster-sql-queries-on-delta-lake-with-dynamic-file-pruning

5. Optimize Data Layout with Z-Ordering

Z-Ordering is an optimization technique in Databricks designed to improve data locality and enhance the performance of queries. It organizes data within a table such that related information is colocated in the same set of files. This co-locality is particularly beneficial for data skipping, a process that allows Databricks to skip over irrelevant data during query execution, thereby reducing the amount of data read and significantly improving query performance. Z-Ordering is especially effective when filtering or querying large datasets on high-cardinality columns (Databricks documentation) (Delta Lake).

How to Use Z-Ordering

To apply Z-Ordering in Databricks, you use the `OPTIMIZE` command followed by the `ZORDER BY` clause. This command reorganizes the data files in a Delta table based on the specified columns, optimizing the layout for query efficiency. For example, to optimize a Delta table for queries that frequently filter by the `eventType` column, you can use the following command:

```
OPTIMIZE events
WHERE date >= current_timestamp() - INTERVAL 1 day
ZORDER BY (eventType);
```

In this example, the table `events` is optimized such that data rows with similar `eventType` values are stored together, facilitating faster retrieval when these fields are queried.

When to Use Z-Ordering

Z-Ordering should be used when you have large datasets and queries that frequently filter on specific high-cardinality columns. It is particularly useful in scenarios where queries involve complex filtering conditions or require rapid access to subsets of data, such as in analytics and business intelligence applications. The technique is not idempotent; therefore, reapplying Z-Ordering periodically, especially after significant data updates, ensures continued optimization (MS Learn) (Databricks documentation).

Best Practices for Using Z-Ordering

1. **Select Appropriate Columns:** Prioritize columns that are frequently used in query predicates and have high cardinality. For example, columns like timestamps or unique identifiers are often ideal candidates.
2. **Combine with Data Skipping:** Ensure that statistics are collected on the columns used in `ZORDER BY` statements, as this enhances the effectiveness of data skipping.
3. **Monitor Performance Impact:** While Z-Ordering can significantly improve query performance, it does come with an initial cost during data reorganization. It is recommended to monitor the impact on both query performance and resource utilization to balance the trade-offs effectively.

Further Reading

- Understanding Z-Ordering: https://learn.microsoft.com/en-us/azure/databricks/optimizations/z-order

- Delta Lake Optimizations: https://docs.delta.io/latest/delta-optimizations.html
- Best Practices for Data Layout: https://www.databricks.com/blog/comprehensive-guide-to-optimize-data-workloads

6. Enable Auto Optimize for Delta Lake Tables

Auto Optimize is a feature in Databricks designed to automatically manage the optimization of Delta Lake tables. It consists of two primary functions: Optimized Writes and Auto Compaction. Optimized Writes dynamically optimize the size of data files during write operations, aiming to produce files close to a target size (typically 128 MB). Auto Compaction further refines the data layout by compacting small files into larger ones, helping to avoid the proliferation of numerous small files, which can degrade performance.

How to Use Auto Optimize

Auto Optimize can be enabled either at the table level or globally for all tables within a Spark session. To enable it for a specific Delta table, you set table properties like `delta.autoOptimize.optimizeWrite` and `delta.autoOptimize.autoCompact` to `true`. For example:

```
CREATE TABLE my_table (id INT, name STRING)
TBLPROPERTIES (
  'delta.autoOptimize.optimizeWrite' = 'true',
  'delta.autoOptimize.autoCompact' = 'true'
);
```

Alternatively, to enable these properties for all new tables in a session, use the following Spark configuration:

```
SET spark.databricks.delta.properties.defaults.autoOptimize.optimizeWrite = true
SET spark.databricks.delta.properties.defaults.autoOptimize.autoCompact = true;
```

These settings ensure that any new tables created in the session will automatically use optimized writes and auto-compaction features.

When to Use Auto Optimize

Auto Optimize is particularly useful in data environments where frequent updates and high transaction volumes occur. It helps maintain an efficient data layout without manual intervention, making it ideal for ETL processes and real-time data applications where maintaining optimal file sizes can significantly enhance query performance. Additionally, Auto Optimize is beneficial in scenarios involving streaming data or when frequent small updates are made to the Delta tables.

Best Practices for Using Auto Optimize

1. **Set Appropriate File Size Targets:** The default file size for Auto Optimize is set to 128 MB. Adjust this target size based on your specific use case to balance between write performance and query efficiency. For instance, larger file sizes may be beneficial for batch processing, while smaller files might be preferred for real-time analytics.
2. **Monitor the Effects of Optimization:** While Auto Optimize can significantly improve performance, it also introduces additional compute

overhead. Monitoring the impact on both write latency and query performance is crucial to ensure that the benefits outweigh the costs.

3. **Use in Conjunction with Other Optimization Techniques:** Combine Auto Optimize with other Delta Lake features like Z-Ordering and data partitioning for comprehensive data layout optimization. This combination can further reduce query times by ensuring efficient data skipping and clustering.

Further Reading

- Delta Lake Optimizations: https://docs.delta.io/latest/delta-optimizations.html
- Auto Optimize Configuration: https://docs.databricks.com/optimizations/auto-optimize.html
- Databricks Community Insights on Auto Optimize: https://community.databricks.com/

7. Use Cost-Based Optimizer (CBO)

The Cost-Based Optimizer (CBO) in Databricks is a query optimization framework that enhances query execution plans by leveraging statistical data about the tables involved. It determines the most efficient way to execute a query based on various factors, such as data distribution, cardinality, and available indexes. CBO is particularly effective in optimizing complex queries with multiple joins, aggregations, and filtering operations, where the choice of join strategy and execution path can significantly impact performance【220】【223】.

How to Use the Cost-Based Optimizer

To utilize the CBO, it is essential to collect and maintain accurate statistics for tables and columns. These statistics can be gathered using the `ANALYZE TABLE` command, which collects information like the number of rows, distinct values, and null counts. For example:

```
ANALYZE TABLE my_table COMPUTE STATISTICS FOR ALL COLUMNS;
```

This command ensures that the optimizer has up-to-date information, enabling it to make informed decisions about the query execution plan. The collected statistics are then used to estimate the costs associated with different query plans, allowing the CBO to select the most efficient one.

When to Use the Cost-Based Optimizer The CBO is especially beneficial in scenarios involving large datasets with complex joins and aggregations. It is most effective when there is a need to optimize resource-intensive queries, such as those in data warehousing and big data analytics. By providing accurate estimates of data size and distribution, the CBO helps in selecting optimal join strategies (like broadcast joins for small tables) and efficient execution paths, thus improving query performance and reducing resource consumption.

Best Practices for Using the Cost-Based Optimizer

1. **Regularly Update Statistics:** Ensure that table and column statistics are regularly updated, especially after significant data modifications, such as large inserts or updates. This practice helps maintain the accuracy of the CBO's estimations, leading to better query optimization.
2. **Verify and Analyze Query Plans:** Use the `EXPLAIN` command to inspect the query plan and verify that the CBO is utilizing statistics correctly. For instance, the output will show whether the optimizer has applied appropriate join strategies based on the collected statistics.
3. **Adjust Configurations as Needed:** Fine-tune settings such as the broadcast join threshold and the level of detail for statistics collection (e.g., including histograms for better cardinality estimation) to enhance the effectiveness of the CBO.

Further Reading

- Databricks Documentation on CBO: https://docs.databricks.com/spark/latest/spark-sql/cbo.html
- Collecting Table Statistics: https://docs.gcp.databricks.com/data-engineering/optimization/cbo.html
- Optimizing Query Performance: https://www.databricks.com/blog/adaptive-query-execution

8. Avoid Unnecessary Collect Operations

In Databricks and Apache Spark, the `collect` operation retrieves the entire dataset to the driver node from the distributed nodes where it is processed. This operation is often used to examine data locally, debug, or perform actions that require access to the entire dataset. However, this can lead to significant performance issues, especially when working with large datasets, as it can overwhelm the driver's memory and cause out-of-memory errors.

How to Use Collect Operation Appropriately

While `collect` can be useful in certain situations, such as retrieving small datasets for inspection or further local processing, it should be used with caution. For example, if you need to inspect the first few records of a dataset, consider using the `show` or `take` methods instead. These methods only bring a limited number of records to the driver node, reducing the risk of memory overload:

```
# Using take to retrieve a small number of records
small_df = df.take(10)
```

In this case, `take(10)` retrieves only 10 records from the DataFrame `df`, which is generally safe and efficient compared to collecting the entire dataset.

When to Avoid Collect Operations

Avoid using `collect` in scenarios where the dataset size is large or unknown. This is particularly important in production environments, where the data scale can vary significantly, and the driver node's resources are limited. Instead, consider using actions like `count`, `agg`, or writing results to distributed storage for analysis, which are designed to handle large-scale data efficiently within the distributed computing framework.

Best Practices for Avoiding Collect

1. **Limit Data Retrieval:** Always use operations like `limit`, `head`, or `sample` to work with a manageable subset of the data. This practice ensures that only the necessary amount of data is moved to the driver node.

2. **Use Distributed Actions:** Leverage distributed actions like `count`, `sum`, or `groupBy` to perform computations within the cluster, rather than bringing the data to the driver for computation.
3. **Optimize Data Pipelines:** Design data pipelines to minimize the need for driver-side operations. Where possible, complete all data transformations and analyses within the distributed nodes and persist the results to a database or distributed file system.

Further Reading

- Databricks Documentation on Dataframe Operations: https://docs.databricks.com/dataframes/dataset-operations.html
- Best Practices for Apache Spark Jobs: https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions
- Avoiding Driver Bottlenecks: https://docs.databricks.com/best-practices/avoid-driver-bottlenecks.html

9. Optimize Join Strategies

In Databricks, join optimization involves selecting the most efficient method for joining tables, a crucial aspect of improving query performance. Different join strategies, such as broadcast joins, sort-merge joins, and shuffle hash joins, can be applied depending on the data size and distribution. Choosing the right join type helps minimize data shuffling and memory usage, which are often the bottlenecks in query execution.

How to Optimize Join Strategies

Join optimization can be achieved through several techniques, including leveraging join hints, enabling adaptive query execution (AQE), and

configuring specific settings for handling skewed data. Join hints allow users to suggest preferred join strategies, guiding the optimizer toward the most efficient execution plan. For instance, using the broadcast join hint (`/*+ BROADCAST */`) can force small tables to be broadcasted to all nodes, avoiding expensive shuffle operations:

```
SELECT /*+ BROADCAST(small_table) */ *
FROM large_table
JOIN small_table ON large_table.id = small_table.id;
```

For larger datasets where broadcasting isn't feasible, sort-merge joins or shuffle hash joins might be more appropriate. AQE dynamically adjusts join strategies at runtime based on the actual data characteristics, automatically selecting the optimal strategy.

When to Use Join Optimization

Join optimization is particularly vital in big data environments where queries involve large tables with complex join conditions. It is especially useful when handling data from different sources, performing ETL processes, or executing analytics queries. Optimizing joins can reduce the computational load, lower query latencies, and improve overall resource utilization, making it critical for maintaining performance and cost-efficiency in data-intensive applications.

Best Practices for Join Optimization

1. **Leverage Join Hints**: Use hints like `BROADCAST` , `MERGE` , and `SHUFFLE_HASH` to explicitly guide the join strategy. For example, `BROADCAST` is beneficial for small tables, while `MERGE` or `SHUFFLE_HASH` can be used for larger, sorted tables.

2. **Enable AQE:** AQE automatically optimizes join strategies at runtime. Ensure AQE is enabled by setting `spark.sql.adaptive.enabled` to true, which helps in dynamically selecting the most efficient join type based on real-time data metrics.
3. **Handle Skewed Data:** Data skew, where data is unevenly distributed across partitions, can lead to inefficient joins. Use skew join hints (`/*+ SKEW */`) or enable AQE's skew handling feature to mitigate this issue by redistributing data more evenly.

Further Reading

- Databricks Join Strategies: https://docs.databricks.com/sql/language-manual/joins.html
- Optimization Recommendations: https://docs.databricks.com/optimizations/optimization-recommendations.html
- Handling Data Skew: https://docs.databricks.com/optimizations/skew-join-optimization.html

10. Cache Intermediate Results

Caching intermediate results in Databricks refers to the practice of storing temporary data in a cache to speed up subsequent operations. This technique can significantly reduce computation times by avoiding repeated calculations on the same data. Databricks offers several caching mechanisms, including the Spark Cache, Databricks Disk Cache, and the recently introduced Remote Result Cache. Each of these serves different purposes and offers distinct advantages, depending on the data and query patterns.

How to Cache Intermediate Results

To cache intermediate results, you can use the `CACHE` or `PERSIST` methods provided by Spark. For instance, to cache a DataFrame in memory, you might use:

```
df.cache()
df.count()  # Action to materialize the cache
```

Alternatively, for more control over where data is stored, such as memory and disk, you can use:

```
df.persist(StorageLevel.MEMORY_AND_DISK)
df.count()  # Materialize the cache
```

The `StorageLevel` parameter allows you to specify the storage type, which can be particularly useful if memory resources are limited.

When to Cache Intermediate Results Caching is particularly beneficial when dealing with iterative processes, such as machine learning training loops or repeated queries on the same dataset. It is also advantageous in scenarios where data transformations are computationally expensive. By caching the results of intermediate steps, you can avoid re-computing these steps, leading to faster overall execution times. For example, in an ETL pipeline, caching intermediate DataFrames can speed up data processing, especially when multiple transformations are applied sequentially.

Best Practices for Caching Intermediate Results

1. **Choose the Appropriate Cache Type:** Use the Spark Cache for in-memory operations that need fast access and low latency. Opt for Databricks Disk Cache or Remote Result Cache for larger datasets that can be stored on local SSDs or cloud storage, providing a balance between performance and cost.
2. **Eviction and Lifecycle Management:** Be mindful of the cache's lifecycle and eviction policies. For instance, data in the Disk Cache is automatically managed and evicted based on a least-recently-used (LRU) policy. Understanding these policies helps in planning resource utilization and avoiding unexpected cache purges【254】【255】.
3. **Monitor Cache Usage:** Regularly monitor cache usage to ensure that critical data remains cached and that memory resources are not being over-committed. Use Spark's built-in UI or Databricks' monitoring tools to inspect cache hit rates and memory consumption.

Further Reading

- Databricks Caching Documentation: https://docs.databricks.com/optimizations/delta-caching.html
- Optimizing Data Workloads: https://www.databricks.com/blog/comprehensive-guide-to-optimize-data-workloads
- Understanding Databricks Cache: https://databricks.com/blog/databricks-cache-boosts-performance

11. Manage Cluster Resources Efficiently

Efficient cluster management in Databricks is vital for optimizing resource utilization, reducing costs, and ensuring high performance. It involves selecting the right cluster configurations, leveraging auto-scaling, and using

features like cluster pools and auto-termination to manage resources dynamically.

How to Manage Cluster Resources

Cluster management starts with choosing the appropriate cluster size and configuration for your workloads. Databricks supports various VM instance types, allowing you to tailor clusters to specific tasks, such as using GPU instances for machine learning workloads or high-memory instances for data-intensive processing. You can configure these settings using the Databricks interface or programmatically with the Databricks API. For example:

```
from databricks import sql
sql.execute("CREATE CLUSTER [CLUSTER_NAME] ...")  # Specify cluster configuratio
```

Auto-Scaling and Auto-Termination Auto-scaling enables clusters to automatically adjust the number of nodes based on the current workload. This feature helps manage resources efficiently by scaling up during peak demand and scaling down during periods of low activity. To set up auto-scaling, configure the `min_workers` and `max_workers` parameters for the cluster. For instance, setting `min_workers = 2` and `max_workers = 10` allows the cluster to dynamically scale within this range.

Auto-termination is another critical feature that helps control costs by shutting down clusters that are inactive for a specified period. This setting is particularly useful for preventing unnecessary costs when clusters are not in use. You can set auto-termination when creating a cluster, specifying the inactivity timeout:

```
CREATE CLUSTER my_cluster
SET auto_terminate = true
SET auto_termination_minutes = 30;
```

Best Practices for Cluster Management

1. **Cluster Pools:** Utilize cluster pools to reduce startup times and manage costs. Cluster pools maintain a set of ready-to-use instances, allowing for faster provisioning of clusters. This is particularly useful for environments with frequent, short-lived jobs, as it reduces the overhead associated with cluster startup times.
2. **Tailored Clusters:** Define and reuse clusters tailored to specific workloads. For instance, create separate clusters for ETL processes, data science workloads, and machine learning training. This approach not only optimizes resource allocation but also simplifies resource management by matching the cluster's capabilities with the workload's requirements.
3. **Regular Monitoring and Optimization:** Continuously monitor cluster performance and cost metrics. Databricks provides comprehensive tools for tracking compute usage and costs, allowing you to make informed decisions about resource adjustments. Regularly review and optimize cluster configurations based on workload patterns and performance data.

Further Reading

- Best Practices for Performance Efficiency: https://docs.databricks.com/best-practices/performance-efficiency.html

- Managing Compute Resources: https://docs.databricks.com/compute/cluster-management.html
- Cost Optimization Techniques: https://community.databricks.com/optimize-costs-databricks

12. Clean Up Old Configurations

Cleaning up old configurations in Databricks involves removing outdated or unnecessary setup settings, data, and metadata to improve performance and manage costs. This process is particularly important in environments where configurations and data structures frequently change, as it helps maintain an optimized system state, reduces clutter, and avoids potential conflicts or inefficiencies.

How to Clean Up Old Configurations

A key method for cleaning up in Databricks is using the `VACUUM` command, which removes files that are no longer needed for Delta Lake tables. This includes files from older versions that exceed the set retention period, thereby freeing up storage space and ensuring that outdated data is not inadvertently accessed. For example:

```
VACUUM my_table RETAIN 168 HOURS;  -- Retains files for 7 days
```

This command deletes data files older than 168 hours (7 days) from the `my_table` Delta Lake table. It's crucial to set an appropriate retention period based on your compliance and data retention requirements.

When to Perform Configuration Cleanup

Regular cleanup should be part of standard maintenance, especially after significant data updates or schema changes. For example, after deleting large volumes of data or changing schema definitions, old versions and metadata can accumulate, leading to increased storage costs and potential performance degradation. It is advisable to schedule periodic cleanups using `VACUUM` to keep the system optimized.

Best Practices for Cleaning Up Configurations

1. **Set Appropriate Retention Policies:** Define and enforce retention policies that align with your data governance requirements. This helps in determining how long historical data should be kept before it is eligible for removal.
2. **Avoid Over-Partitioning:** Over-partitioning data can create a large number of small files, which can be inefficient to manage and clean up. Instead, use partitioning strategies that balance query performance with manageability.
3. **Use Predictive Optimization:** Enable predictive optimizations for Unity Catalog-managed tables, which automate the running of `VACUUM` and other maintenance tasks. This feature helps maintain optimal performance and reduces manual intervention.
4. **Monitor and Adjust**: Regularly monitor the impact of cleanup operations on both storage and query performance. Adjust retention periods and cleanup schedules based on the system's evolving needs and the nature of the workloads.

Further Reading

- Best Practices for Delta Lake Management: https://docs.databricks.com/optimizations/delta-lake.html

- Using the VACUUM Command: https://docs.databricks.com/delta/vacuum.html
- Managing Delta Lake Files and Metadata: https://kb.databricks.com/data/vacuum-best-practices.html

13. Monitor and Optimize Data Skew

Data skew occurs when data is unevenly distributed across partitions, leading to some partitions containing significantly more data than others. This imbalance can cause certain tasks to take much longer to complete, resulting in poor utilization of cluster resources and prolonged query execution times. Skew is particularly problematic in operations like joins and aggregations, where large partitions can become bottlenecks.

How to Identify and Handle Data Skew

To detect data skew, you can use the Spark UI or Databricks' monitoring tools to inspect task durations. In a healthy system, task durations should be relatively uniform. However, if a few tasks take much longer than others, it indicates skew. For example, when reviewing the Spark UI, check if the maximum task duration significantly exceeds the 75th percentile, which suggests skew.

To address data skew, Databricks provides several strategies:

1. **Skew Join Hints**: Utilize skew join hints in your SQL or DataFrame operations to explicitly indicate which columns are skewed. For example:

```
df = spark.read.table("large_table")
```

```
df_with_hint = df.hint("skew", "skewed_column")
```

This hint informs the optimizer to handle skew by redistributing data more evenly across partitions during join operations.

Adaptive Query Execution (AQE): Ensure that AQE is enabled, which is the default setting in Databricks. AQE automatically adjusts the execution plan at runtime, optimizing for data skew by dynamically redistributing data and applying better join strategies.

Salting: Introduce a "salt" column that adds randomness to the data distribution. This method helps spread data more evenly across partitions, especially useful in scenarios where a single column causes heavy skew.

When to Monitor for Data Skew Regular monitoring is crucial, especially after data updates or schema changes. It's also important during initial data ingestion and transformation processes, where the data distribution may not yet be well understood. Continuously checking for skew in long-running stages or tasks is vital for maintaining optimal performance.

Best Practices for Optimizing Data Skew

1. **Proactive Monitoring:** Use Databricks' monitoring tools and the Spark UI to proactively identify skew-related issues. Look for imbalances in task durations and resource usage.
2. **Use Appropriate Partitioning:** Optimize partitioning schemes based on data characteristics and query patterns. Avoid over-partitioning, which can exacerbate skew issues.

3. **Combine Multiple Techniques**: Utilize a combination of skew join hints, AQE, and salting to address different aspects of skew, ensuring a balanced approach to data distribution.

Further Reading

- Managing Data Skew: https://kb.databricks.com/data/skew-hints-in-join
- Adaptive Query Execution: https://docs.databricks.com/spark/latest/spark-sql/aqe.html
- Best Practices for Data Management: https://docs.databricks.com/data-management/index.html

Conclusion

optimizing Databricks queries is essential for improving performance, reducing costs, and maximizing the efficiency of data workflows. The strategies outlined in this guide, such as using Adaptive Query Execution, leveraging the Photon engine, and implementing Z-Ordering, provide practical ways to enhance query execution times and resource utilization. By understanding when and how to apply these techniques, data engineers can tailor their approaches to the specific needs of their workloads, ensuring that data is processed efficiently and at scale.

Key takeaways include the importance of proactive monitoring, regular cleanup of old configurations, and the careful management of cluster resources. Addressing challenges like data skew and selecting the appropriate join strategies are critical steps in maintaining optimal performance. By implementing these best practices, organizations can build a robust and scalable data infrastructure, capable of handling complex analytics and data processing tasks with ease.

Learn More

13 Ways to Optimize Databricks Autoscaling

Databricks autoscaler is a critical feature for managing cluster resources efficiently, ensuring optimal performance...

overcast.blog

13 Ways to Reduce Databricks Costs in 2024

Controlling and cutting Databricks costs is a strategic imperative as data volumes and processing demands continue to...

overcast.blog

11 DataBricks Tricks That Will Blow Your Mind

Unlocking the full power of Databricks can lead to dramatic improvements in data processing efficiency, scalability...

overcast.blog

Databricks | Data | Data Engineering | Programming | Software Development

Published in overcast blog

862 followers · Last published Dec 4, 2025

Follow

Cloud-Native Engineering: Kubernetes, Docker, Micro-services, AWS, Azure, GCP & more.

Written by DavidW (skyDragon)

12.6K followers · 7 following

Follow

Into cloud-native architectures and tools like K8S, Docker, Microservices. I write code to help clouds stay afloat and guides that take people to the clouds.

No responses yet

A Angshuman Bhattacharya

What are your thoughts?

More from DavidW (skyDragon) and overcast blog

 In overcast blog by DavidW (skyDragon)

 In overcast blog by DavidW (skyDragon)

Multi-Environment Deployments with Docker: A Guide

Deploying applications across multiple environments—development, testing,...

Feb 6 32 2

Optimizing Docker Networking Performance: Reducing Latency...

Efficient Docker networking is essential because containerized applications rely on...

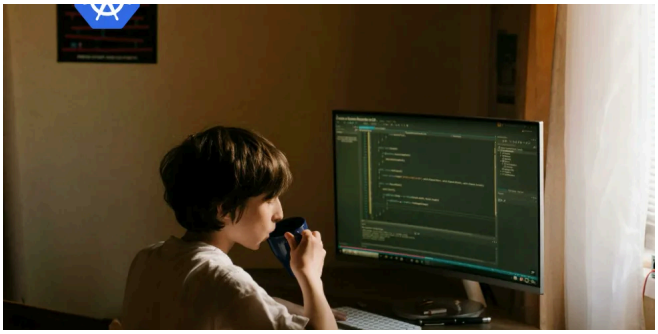
Oct 7, 2024 7

In overcast blog by DavidW (skyDragon)

Provisioning Kubernetes Local Persistent Volumes: Full Tutorial

In the Kubernetes ecosystem, efficient management of storage resources is pivotal...

Feb 12, 2024 6 1



In overcast blog by DavidW (skyDragon)

13 Kubernetes CLI Tools You Should Know

Kubernetes, the leading container orchestration platform, provides a rich...

Jun 18, 2024 19

See all from DavidW (skyDragon) See all from overcast blog

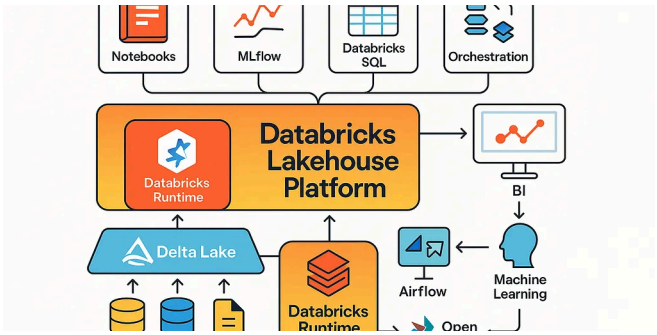
Recommended from Medium

Mayuran

Handling Repetitive Data Loads in a Medallion Architecture with...

How we balanced historical retention, deduplication, and storage efficiency when...

Jul 12 36 1



In Tech with Abhishek by Abhishek Kumar Gupta

🔥 50 Databricks Interview Questions & Answers: The Ultima...

Unlock your Databricks interview success with expert questions and practical, detailed...

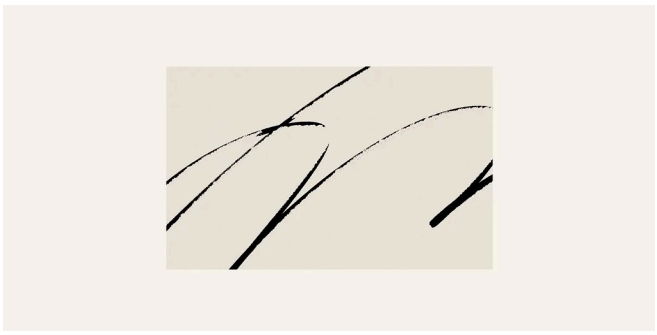
Aug 23 82 2

Ankur Gupta

Data Engineering Interview Prep Series—Part 5: Performance...

How to fix slow SQL queries, optimize PySpark jobs, and reduce cloud costs in dat...

Dec 6 22

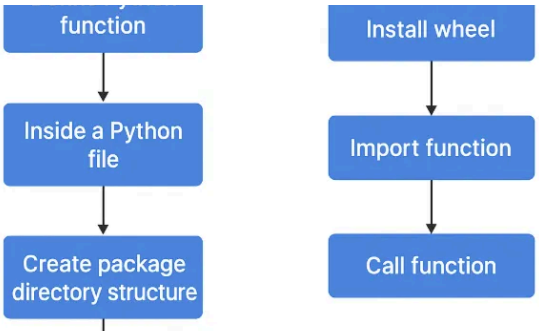


Karthik

Streaming Data with Apache Spark Structured Streaming

The Evolution from Batch to Streaming

Oct 28 25



Data AI

Python Wheels in Databricks

When working with Databricks across multiple notebooks, environments, or teams,...

Nov 4 10

Saurabh Mahajan

My PwC Data Engineer Interview Experience

— A Story of Technical Firefights, Architecture Debates & A Little Bit of Anxiety

Dec 3 82 1

See more recommendations