**Technical Blog**

Explore in-depth articles, tutorials, and insights on data analytics and machine learning in the Databricks Technical Blog. Stay updated on industry trends, best practices, and advanced techniques.



# Top 10 query performance tuning tips for Databricks Serverless SQL

**kamalendubiswas**
Databricks Employee

09-06-2023 07:39 AM

Databricks Serverless SQL (DBSQL) is the latest offering from Databricks to build data warehouses on the Lakehouse. It incorporates all the Lakehouse features like open format, unified analytics, and collaborative platforms across the different data personas within an organisation. In the last couple of quarters, we have seen tremendous growth and adoption of DBSQL with a lot of customers migrating from different cloud data warehouses to DBSQL and one of the keys to their success is the server-less nature of DBSQL which provides instant and elastic compute, lower total cost of ownership and management overhead. You can further improve your experience with Databricks SQL Serverless through optimisations and performance tuning applied by different data personas. These can be broadly categorised into:

- **Resource Optimisation** - These are the knobs that can be optimised on the DBSQL warehouse compute cluster that a warehouse owner can leverage to build an optimised platform for the analytics users. It includes provisioning a proper warehouse size (S, M, L etc.) based on the workloads and setting up the correct auto-scaling factors to handle spiky workloads.

- **Storage Optimisation** - The performance of a query is significantly impacted by the underlying data files, format, and layout. Leveraging Delta can significantly improve query performance derived from its optimised, columnar data format, advanced optimisation techniques, and ACID guarantees. If you want to learn more about storage optimisation, here is a nice blog about it - Six tried and tested ways to turbocharge Databricks SQL.

- **Query Optimisation** - Oftentimes SQL users rely heavily on the engineering or platform team for most of the optimisation but it is crucial to write better queries to achieve the best query performance. It is quite possible that a poorly written query can choke a huge resourceful warehouse and thus cost a lot in terms of time and money.

In this blog post, I will share the Top 10 query performance tuning tips that Data Analysts and other SQL users can apply to improve DBSQL Serverless performance.

### Summary of Tips

- Avoid SELECT * from my_table
- Limit your search
- Integer vs String data types
- Leverage caching
- CTE vs Sub queries
- Use Photonizable functions
- Capitalise join hints
- Run ANALYZE, OPTIMIZE and VACUUM
- Understand your query plan - EXPLAIN
- Format your SQL

### Tip 1 - Avoid SELECT * FROM my_table

*SELECT * FROM my_table* will retrieve all the columns and rows from the given table and it is an Input-Output intensive operation for a large table. Usually, we use this simple query just to identify the column names of the table or to profile sample data. This naive approach can cause bottlenecks in the query execution with growing numbers of users simultaneously trying to profile the tables in this fashion. It is always recommended to select only the required columns for the analysis instead of fetching all the columns. You can use the Data Explorer option in the Databricks console to profile the table for the column details and analyse the sample data set as well. This will help you to avoid doing *SELECT **  on tables.

### Tip 2 - Limit your search

**LIMIT** - The *LIMIT* clause is used to restrict the number of rows returned by a query. We mostly use it for data profiling or retrieving arbitrary subsets of data. It specifies the maximum number of rows that should be retrieved from the database result. DBSQL Query Editor by default adds LIMIT 1000 to all the queries. Limiting the number of rows returned by a query can improve the performance of the database. By retrieving only the necessary subset of rows, the database engine can avoid unnecessary processing and reduce the amount of data transferred over the network. Large result sets can consume a significant amount of memory, especially if they are not needed in their entirety. By using *LIMIT*, you can control the amount of memory required to hold the query result.

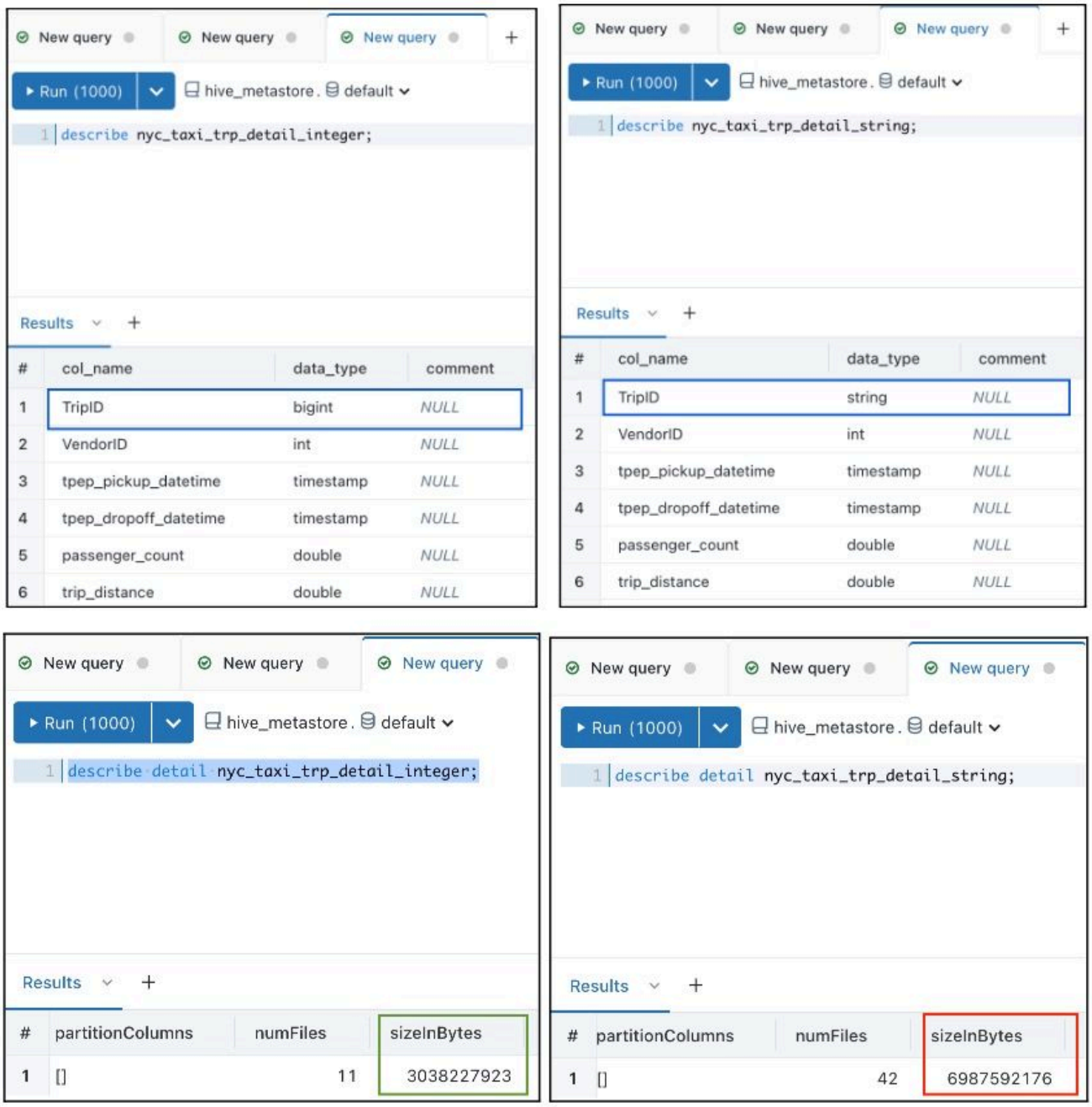
**FILTER** - The *WHERE* clause enables you to select specific rows from a table based on certain criteria. By specifying filters, you can retrieve only the data that meets the desired conditions, making your query results more relevant and meaningful. You can also leverage the capabilities of *Z-Ordering* to reduce your data scan and thus improve query performance. *Z-ORDER* is particularly beneficial for workloads that involve range-based queries, such as filtering data based on a specific date range, numerical ranges, or other column-based ranges. For more detailed information on *Z-ORDER*, please visit the documentation.

### Tip 3 - Integer vs String Data Types

When designing a database table, choosing between integer and string data types depends on several factors, including the nature of the data, the usage patterns, and the specific requirements of your application. A common choice made by customers is to opt for **HASH** function based alphanumeric keys because they are deterministic. But if you think from a storage perspective, the alphanumeric keys are of string data type and that is a lot of storage as compared to integer type. This can slow down the data import to your BI tools and thus degrading the report performance.

Here is an example of the NYC Taxi Trip Data with 100+ million rows. I have two tables: ***nyc_taxi_trp_detail_string*** with ***TripID*** as string and **nyc_taxi_trp_detail_integer** with ***TripID*** as integer. The size of the table with an integer key is around 3GB whereas

the one with a string key is an astonishing 7GB.




Ultimately, the choice between integer or string columns depends on the specific requirements and characteristics of your data and application. In some cases, a combination of both types may be appropriate, such as using an integer surrogate key for internal referencing and a string natural key for external interactions. It's important to consider factors like performance, data integrity, integration needs, and the meaningful representation of your data when making this decision.

### Tip 4 - Leverage caching

Caching in DBSQL can significantly improve the performance of iterative or repeated computations by reducing the time required for data retrieval and processing. Caching allows you to avoid redundant computations by reusing previously calculated data, thereby reducing the overall processing time of your queries or operations. However, it's important to use caching judiciously and consider the memory requirements of your workload to avoid excessive memory usage or potential out-of-memory issues. Databricks offers a wide variety of caching mechanisms and DBSQL leverages those features.

- **DBSQL UI Cache** aims to improve user experience by providing quick access to the most recent queries, dashboards and result sets.
- **Query Result Cache** includes both Local Cache and Remote Cache (serverless only).
    - **Local Cache** - it stores the latest query results in-memory for the cluster's lifetime or the cache is full. It is based on the exact query text.
    - **Remote Cache** - persists the data in the cloud storage for all warehouses across a Databricks Workspace. It is based on the Logical Plan.
- **Disk cache**, previously known as Delta cache - The Disk Cache is designed to enhance query performance by storing data on disk, allowing for accelerated data reads. Data is automatically cached when files are fetched, utilising a fast intermediate format.

As an SQL user, you can cache the data as a SELECT statement for the most frequently used tables or queries in the Disk Cache.

```
CACHE SELECT * FROM boxes;
CACHE SELECT width, length FROM boxes WHERE height=3;
```

You don't need to use this command for the disk cache to work correctly, the data will be cached automatically when first accessed. But it can be helpful when you require consistent query performance.

*This construct is applicable only to Delta tables and Parquet tables.*

### Tip 5 - CTE vs Subqueries

CTE (Common Table Expressions) and subqueries are both powerful techniques in SQL for expressing complex queries and organising data. While they can achieve similar results, subqueries can sometimes be less performant than CTEs, especially when used in complex queries or with large datasets. The optimiser may choose different query execution plans for subqueries compared to CTEs. Depending on the specific query, the optimiser may or may not treat subqueries and CTEs equivalently in terms of optimisation and performance. In addition to that, CTEs can improve the readability and maintainability of complex queries by breaking them down into smaller, more manageable parts. They provide a way to give meaningful names to intermediate result sets, making the query logic easier to understand. CTEs can be referenced multiple times within the same query, enabling code reuse and avoiding the need to repeat complex subqueries. This can lead to more concise and modular queries.

### Tip 6 - Use photonizable functions

Photon is a new vectorised query engine on Databricks developed in C++ to take advantage of modern hardware and is compatible with Apache Spark APIs. DBSQL uses Photon by default which accelerates the query execution that processes a significant amount of data and includes aggregations and joins. Most of the DBSQL native functions are supported by Photon. You can read more about the scope of Photon in the coverage documentation. User Defined Functions (UDF) are not photonizable yet, be sure to explore all native photonizable functions before writing any UDF.

### Tip 7 - Capitalise on Join Hints

Join hints allow users to explicitly suggest the join strategy that the DBSQL optimiser should use. When different join strategy hints are specified on both sides of a join, Databricks SQL prioritises hints in the following order: **BROADCAST** over **MERGE** over **SHUFFLE_HASH** over **SHUFFLE_REPLICATE_NL**.

- **BROADCAST** - Use broadcast join. The join side with the hint is broadcast regardless of autoBroadcastJoinThreshold. If both sides of the join have the broadcast hints, the one with the smaller size (based on stats) is broadcast. The aliases for **BROADCAST** are **BROADCASTJOIN** and **MAPJOIN**. Broadcast join is applicable when one of the datasets involved in the join is small enough to fit entirely in the memory of each executor node in the cluster. By broadcasting the smaller dataset to all executor nodes, Spark can avoid the costly shuffle phase that

occurs in regular join operations, which can significantly speed up the overall computation.

- **MERGE** - Use shuffle sort-merge join. The aliases for **MERGE** are **SHUFFLE_MERGE** and **MERGEJOIN**. When both datasets involved in the join are too large to fit entirely in memory, unlike broadcast join, merge join performs efficiently. Merge join requires the input datasets to be sorted on the join key. If the datasets are already sorted, or if sorting them doesn't introduce significant overhead, merge join is a good choice.
- **SHUFFLE_HASH** - Use shuffle hash join. If both sides have the shuffle hash hints, Databricks SQL chooses the smaller side (based on stats) as the build side. When both datasets involved in the join are too large to fit entirely in memory and the data distribution is skewed, meaning some keys have significantly more records than others, shuffle hash join is performed by the spark engine.
- **SHUFFLE_REPLICATE_NL** - Use shuffle-and-replicate nested loop join. This is a cartesian product join, also known as cross join. When you perform a Cartesian join between two datasets, the resulting output will have a size equal to the product of the number of rows in each dataset. This can quickly become computationally expensive and lead to a large number of output rows, especially if the datasets involved are large.

When both sides are specified with the **BROADCAST** hint or the **SHUFFLE_HASH** hint, Databricks SQL picks the build side based on the join type and the sizes of the relations. Since a given strategy may not support all join types, Databricks SQL is not guaranteed to use the join strategy suggested by the hint.

**Tip 8 - Run ANALYZE, OPTIMIZE and VACUUM**

ANALYZE collects the statistics of a specified table. These statistics are used by the DBSQL query optimiser to generate a better execution plan.

```
ANALYZE TABLE nyc_taxi_trp_detail_integer COMPUTE STATISTICS;
```

OPTIMIZE statement optimizes the layout of the Delta Lake data files. Optionally you can Z-Order the data to colocate the related data in the same set of files.

```
OPTIMIZE nyc_taxi_trp_detail_integer;
OPTIMIZE nyc_taxi_trp_detail_integer ZORDER BY (TripID);
```

VACUUM removes unused files from the table directory. It removed all the data files that are no longer in the latest state of the transaction log for the table and are older than a retention threshold. The default threshold is 7 days. If you run Vacuum on a delta table, you lose the ability to time travel back to the older versions of the table but it helps in reducing the storage cost.

```
VACUUM nyc_taxi_trp_detail_integer DRY RUN;
VACUUM nyc_taxi_trp_detail_integer;
```

### Tip 9 - Understand your query plan - EXPLAIN

It is very important to explore and analyse your query plan before executing it. This enables you to understand how the code will actually be executed and is useful for optimising your queries. The DBSQL optimiser automatically generates the most optimised physical plan which is then executed.

```
EXPLAIN [EXTENDED|CODEGEN|COST|FORMATTED]
SELECT
VendorID, count(TripID) AS TotalTrip
FROM
nyc_taxi_trp_detail_integer
GROUP BY VendorID;
```

DBSQL also offers the feature of Adaptive Query Execution (AQE) which is a re-optimisation of the query plan that happens during the query execution. This can be very useful when statistics collection is not turned on or when statistics are stale. It is also useful in places where statically derived statistics are inaccurate, such as in the middle of a complicated query, or after the occurrence of data skew.

### Tip 10 - Format your SQL codes

Last but not least, formatting your SQL queries is always a best practice to follow. It won't improve the query performance but a well-formatted code can definitely improve your efficiency in understanding someone else's code, debugging, refactoring or building up a new business logic. Some of the best practices are:

- Follow similar naming conventions for all tables and columns.
- Add comments if needed, this will help to understand the business requirements.
- Avoid extrapolating lists of values within an IN clause.

- Explicitly define data types and avoid mixing different date formats
- For parameterized queries, add comments for the parameters.
- Terminating a query with a semicolon doesn't harm anyone!

### Conclusion

Implementing these best practices enhances query processing speed and also contributes to efficient resource utilisation, scalability, and overall system stability, enabling organizations to handle growing data volumes and user demands.

28 Kudos

## 6 COMMENTS

**Axel_Schwanke**

Contributor

11-05-2023 12:41 AM

11-05-2023 12:41 AM

regarding LIMIT your search:

In my experience, if the tables are partitioned, you should first try to use the partitioning columns for filtering.

4 Kudos

**AG2**

New Contributor III

11-27-2023 02:07 AM

11-27-2023 02:07 AM

Some more **Performance optimization** techniques one should follow

i. *Use the latest LTS version* of Databricks Runtime, The latest Databricks runtime is almost always faster than the one before it(DBR 14.0 is GA with Spark 3.5.0, UDAF for Python, Row-level concurrency, Spark Connect)
ii. *Use Photon* - fastest Spark execution
iii. *Restart long-running clusters periodically*
iv. *Use Cluster Policies* to enforce best practices!

6 Kudos

**shadowinc**

New Contributor III

08-18-2024 04:25 AM

08-18-2024 04:25 AM

Thanks for the tips. Will try those. Formatting huge queries is not a good experience for me as I have to scroll up and down for any troubleshooting and it is a huge pain tbh. Rather I separate CTEs with a space and in CTE everything is in a line (Can't work for everyone though)

0 Kudos

**BapsDBC**

New Contributor III

08-25-2024 10:08 PM

08-25-2024 10:08 PM

I have a quick one.

For Server-less SQL Warehouses, Databricks claims that performance of such warehouses and the objects within is handled by Databricks itself. Which is believable based on the fact that Databricks still uses its Spark Engine to manipulate data in the SQL Warehouse (please correct me if I am wrong).

If the above statement is correct, then why do we need to think of all the query optimisations ? Table optimisations using OPTIMIZE and VACUUM (on delta tables) is still understandable. But query optimisation like using of CTE, Hints etc. ? Is that really necessary ? If yes, why ? What happened to the claims made for Spark SQL Engine and it's newfound sibling the AI ?

1 Kudo

**Ajay-Pandey**

Databricks MVP

09-08-2024 08:57 AM

09-08-2024 08:57 AM

@kamalendubiswas Thanks for sharing

0 Kudos

**Mantsama4**

Valued Contributor

02-06-2025 07:56 PM

02-06-2025 07:56 PM

This is a great solution! The post provides an in-depth, structured approach to optimizing Databricks SQL Serverless, highlighting key tips such as resource optimization, query performance improvements, and the best practices for data types and caching. The emphasis on leveraging Delta Lake and Photon for enhanced performance is particularly valuable for those looking to maximize efficiency while maintaining scalability.

I'm curious—could you kindly share some examples of how organizations have successfully implemented these performance tuning tips in their real-world use cases? I would love to learn more about the practical application of these strategies!

 0 Kudos

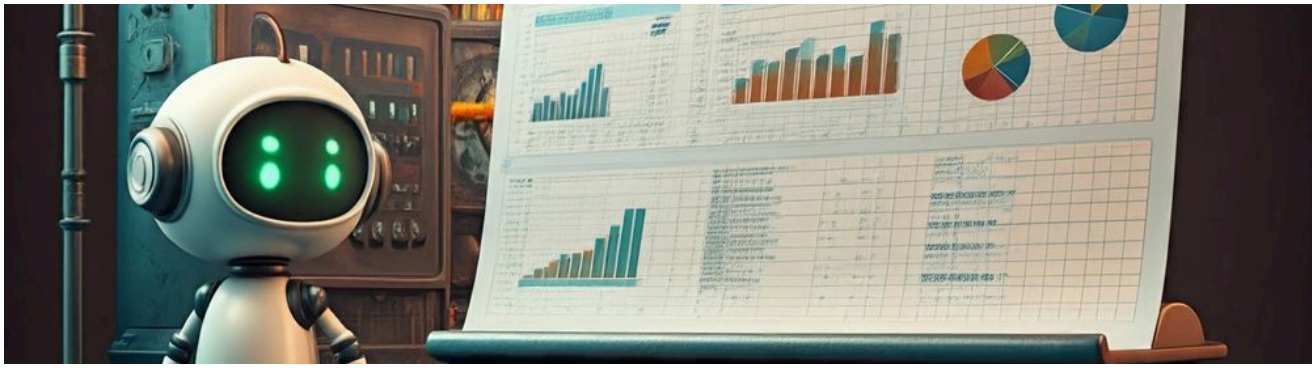
**You must be a registered user to add a comment. If you've already registered, sign in. Otherwise, register and sign in.**

**Comment**

## Contributors

 **kamalendubiswas**

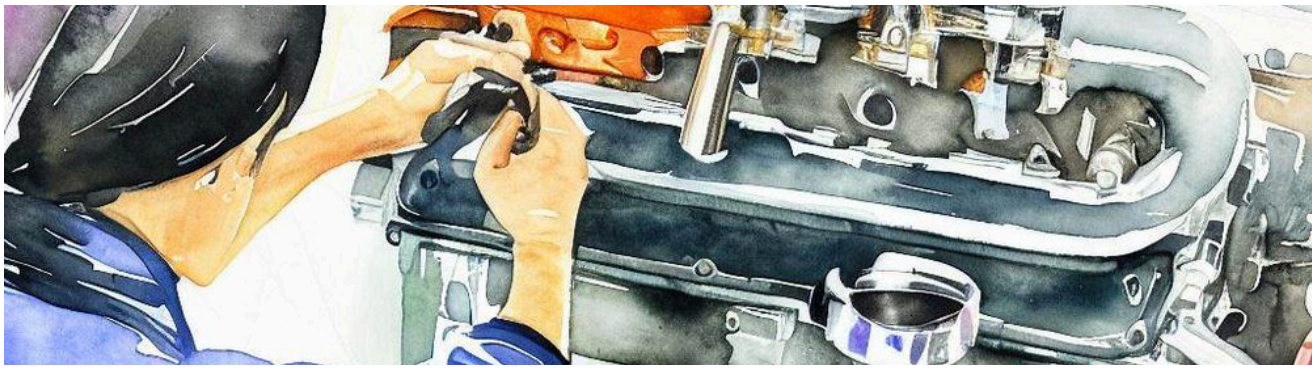
## Popular Articles



## Metadata-Driven ETL Framework in Databricks (Part-1)

by **Rjt_de** •
Databricks Employee

223250 27 46



## Top 10 query performance tuning tips for Databricks Serverless SQL

by **kamalendubiswas** •
Databricks Employee

137687 6 28

## Best practices for safe data experimentation with Databricks

by **Mahavir_Teraiya** •
Databricks Employee

15655 5 24

## Related Content

**Declarative Pipelines meets foreachBatch: Custom Streaming for Advanced Pipelines**

in **Technical Blog** yesterday

---

**[PARTNER BLOG] Building an Agent-Native Data Quality Manager with Databricks Apps and DQX**

in **Technical Blog** yesterday

---

**🌟 Community Pulse: Your Weekly Roundup! December 22, 2025 – January 04, 2026**

in **Announcements** Tuesday

---

**Solution Accelerator Series | Scale cybersecurity analytics with Splunk and Databricks**

in **Announcements** Tuesday

---

**Designing Reliable Stream–Stream Joins with Watermarks in Databricks**

in **Community Articles** 2 weeks ago

---



Databricks Inc.
160 Spear Street, 13th

Floor
San Francisco, CA 94105
1-866-330-0121