DATA+AI SUMMIT
BY databricks

Visualizing Trillion Point Datasets with Databricks and Plotly

Sachin Seth
Date 6/13/2024

Introduction

Short overview of the presentation and topics we are going to cover

- Purpose: *Explore the use of Databricks SQL and Plotly Arrow Resampler (new) for* ***very-large-scale time-series data visualization*** *(relevant for vehicle test fleet analytics such as at Mercedes)*
    - Server side implementation - using Databricks SQL - of sophisticated downsampling algorithms
    - Pass down samples Arrow files from Databricks to Plotly Dash Enterprise using Databricks SQL Connectors
    - Client sided downsampling using Plotly Arrow Resampler for optimized real time visualization

Analyzing the Fleet

Description of the business problem

Background

- Vehicle test fleets such as Mercedes' often include 100's of vehicles (or more) generating terabytes of data (100's of billions of high-fidelity, high-frequency time series data points) daily which include telemetry, diagnostics and usage patterns.
    - Data is ingested into Databricks in varied formats and velocities (from real-time GPS feeds to batch-upload maintenance logs).

Requirement

- Multiple teams of engineers want to (locally and remotely) analyze this data daily with a high degree of interactivity and flexibility (which thus requires a very high degree of performance in order to work effectively given the scales involved)
    - Directly visualizing raw data from thousands of vehicles is currently impractical

Expanded Fleet Dataset

Description of the fleet data generation

Data Per Sample:

Each data feed from one vehicle generates one sample every millisecond

Each sample is a 32-bit (4-byte) data poin

Data Generation per Vehicle:

Each vehicle has 100 data feeds

Each vehicle generates 100 feeds x 4 bytes/sample = 400 bytes per sample

Data Generation Rate per Vehicle

Since each feed samples data every millisecond, the data rate per vehicle is 400 bytes/ms

Total Fleet Generation Rate

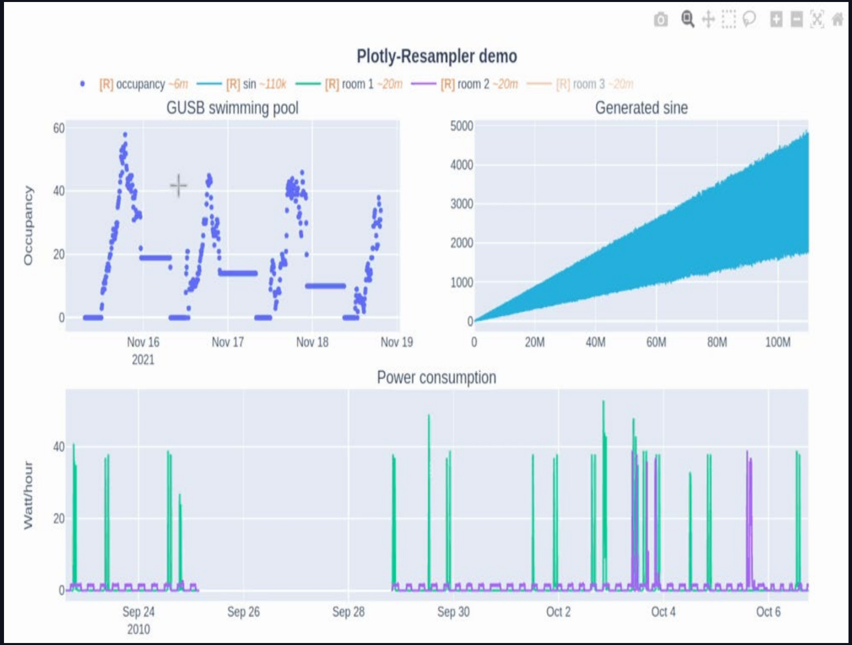
With 1000 vehicles the fleet is generating around 400 MB/s! On 42 minutes or so to generate 1TB of data

Background on the Plotly Resampler

Description of capabilities of existing Plotly Resampler

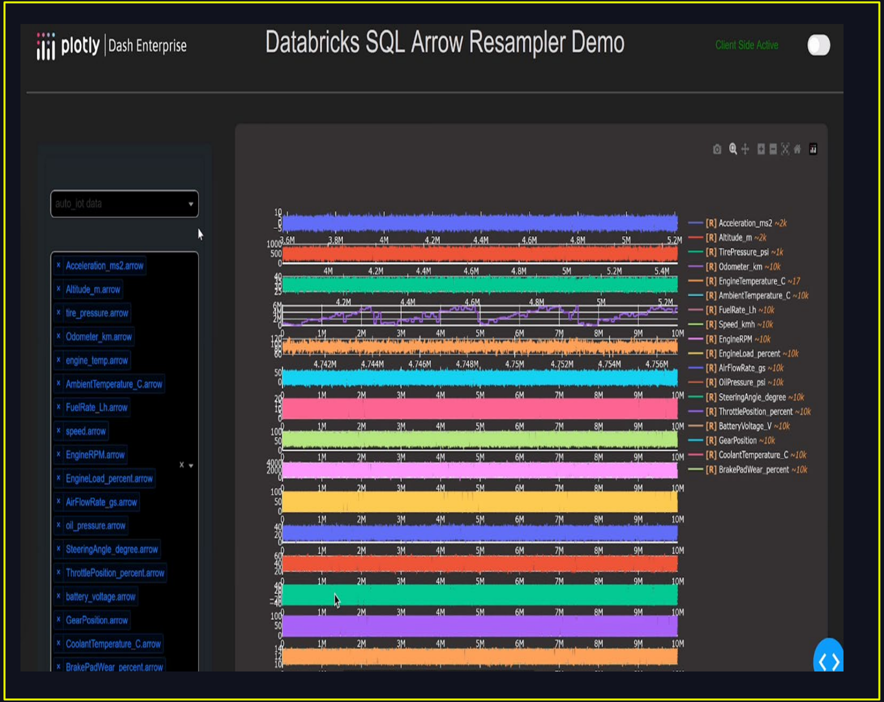
pip install plotly-resampler

- >400K downloads/mo
- developed by J & J Van Der Donckt (twins)
- **dynamically aggregates time-series data respective to a current Plotly graph view**
- *is actually a Plotly Dash AIO component !*
- leverages optimized MinMaxLTTB
- https://arxiv.org/abs/2206.08703
- good for large datasets but memory intensive with subsequent downsampling
    - effective max = 10M points on 32GB
    - not targeted for high concurrency



Intro to Plotly Arrow Resampler (new)

Introduction to Plotly Arrow Resampler



- Improves upon the existing Plotly Resampler by exclusively using Arrow supported data types
- Able to perform zero-copy on Arrow columnar memory format
- Data does not need to be copied into app memory from storage
- Faster and uses less CPU resources, supporting higher concurrency
- Features Databricks Connectivity!

Start Simple with Local Resampling

Overview of client -side downsampling with cached Arrow files

Process Data in Databricks

- Data from IoT devices, user interactions and system logs is ingested into Databricks
- Using Databricks medallion architecture to process data into discrete tables

Cache Data in Arrow Files

- Use Databricks SQL connector for Python to pull data out of Databricks as PyArrow table object
- Schedule period updates to cached Arrow file (e.g. nightly)

Use Resampler to Visualize

- Directly feed Arrow files into Plotly Arrow Resampler for visualization
- User high powered, ultra efficient Rust algorithms to explore entire dataset in real-time

What it looks like

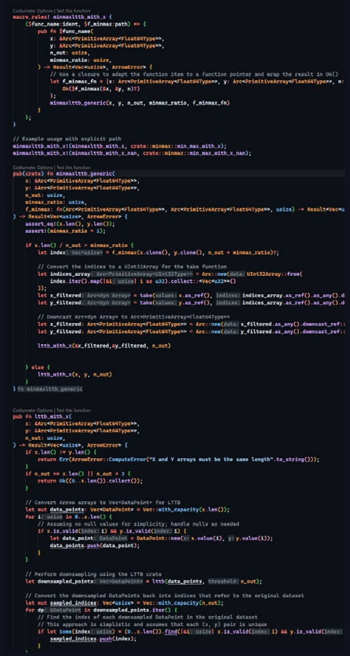
Databricks SQL connector code for pulling data out of Databricks in Arrow

PYTHON

```
from databricks import sql
import os

with sql.connect(server_hostname = os.getenv("DATABRICKS_SERVER_HOSTNAME"),
                 http_path       = os.getenv("DATABRICKS_HTTP_PATH"),
                 access_token    = os.getenv("DATABRICKS_TOKEN")) as connection:

  with connection.cursor() as cursor:
    cursor.execute("SELECT * FROM sample.trips LIMIT 2")
    result = cursor.fetchall_arrow()

    for row in result:
      print(row)
```

Overview of the Downsampling Algo

Brief overview of the algorithm itself and why it stands out

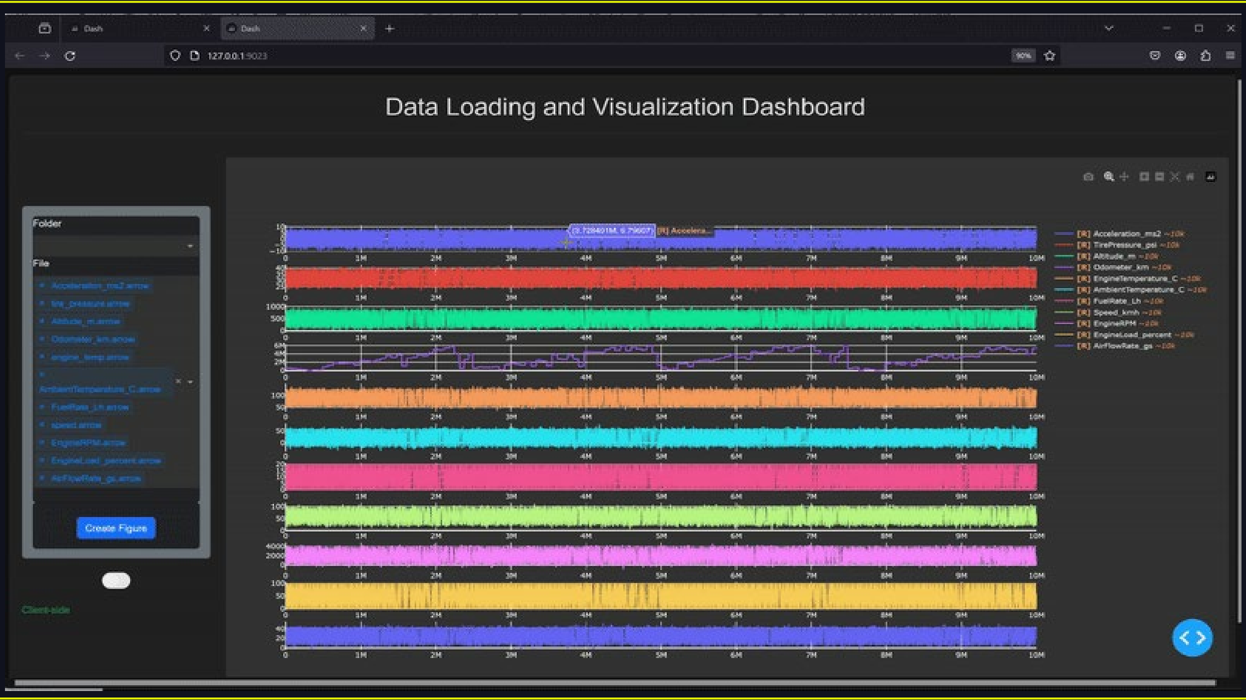


```
Codiumate: Options | Test this function
macro_rules! minmaxlttb_with_x {
    ($func_name:ident, $f_minmax:path) => {
        pub fn $func_name(
            x: &Arc<PrimitiveArray<Float64Type>>,
            y: &Arc<PrimitiveArray<Float64Type>>,
            n_out: usize,
            minmax_ratio: usize,
        ) -> Result<Vec<usize>, ArrowError> {
            // Use a closure to adapt the function item to a function pointer and wrap the result in Ok()
            let f_minmax_fn = |x: Arc<PrimitiveArray<Float64Type>>, y: Arc<PrimitiveArray<Float64Type>>, n:
                Ok($f_minmax(&x, &y, n)?)
            };
            minmaxlttb_generic(x, y, n_out, minmax_ratio, f_minmax_fn)
        }
    };
}

// Example usage with explicit path
minmaxlttb_with_x!(minmaxlttb_with_x, crate::minmax::min_max_with_x);
minmaxlttb_with_x!(minmaxlttb_with_x_nan, crate::minmax::min_max_with_x_nan);

Codiumate: Options | Test this function
pub(crate) fn minmaxlttb_generic(
    x: &Arc<PrimitiveArray<Float64Type>>,
    y: &Arc<PrimitiveArray<Float64Type>>,
    n_out: usize,
    minmax_ratio: usize,
    f_minmax: fn(Arc<PrimitiveArray<Float64Type>>, Arc<PrimitiveArray<Float64Type>>, usize) -> Result<Vec<u
) -> Result<Vec<usize>, ArrowError> {
    assert_eq!(x.len(), y.len());
    assert!(minmax_ratio > 1);

    if x.len() / n_out > minmax_ratio {
        let index: Vec<usize> = f_minmax(x.clone(), y.clone(), n_out * minmax_ratio)?;

        // Convert the indices to a UInt32Array for the take function
        let indices_array: Arc<PrimitiveArray<UInt32Type>> = Arc::new(data: UInt32Array::from(
            index.iter().map(|&i: usize| i as u32).collect::<Vec<u32>>()
        ));
        let x_filtered: Arc<dyn Array> = take(values: x.as_ref(), indices: indices_array.as_ref().as_any().d
        let y_filtered: Arc<dyn Array> = take(values: y.as_ref(), indices: indices_array.as_ref().as_any().d

        // Downcast Arc<dyn Array> to Arc<PrimitiveArray<Float64Type>>
        let x_filtered: Arc<PrimitiveArray<Float64Type>> = Arc::new(data: x_filtered.as_any().downcast_ref::
        let y_filtered: Arc<PrimitiveArray<Float64Type>> = Arc::new(data: y_filtered.as_any().downcast_ref::

        lttb_with_x(&x_filtered,&y_filtered, n_out)


    } else {
        lttb_with_x(x, y, n_out)
    }
} fn minmaxlttb_generic

Codiumate: Options | Test this function
pub fn lttb_with_x(
    x: &Arc<PrimitiveArray<Float64Type>>,
    y: &Arc<PrimitiveArray<Float64Type>>,
    n_out: usize,
) -> Result<Vec<usize>, ArrowError> {
    if x.len() != y.len() {
        return Err(ArrowError::ComputeError("X and Y arrays must be the same length".to_string()));
    }
    if n_out >= x.len() || n_out < 3 {
        return Ok((0..x.len()).collect());
    }

    // Convert Arrow arrays to Vec<DataPoint> for LTTB
    let mut data_points: Vec<DataPoint> = Vec::with_capacity(x.len());
    for i: usize in 0..x.len() {
        // Assuming no null values for simplicity; handle nulls as needed
        if x.is_valid(index: i) && y.is_valid(index: i) {
            let data_point: DataPoint = DataPoint::new(x: x.value(i), y: y.value(i));
            data_points.push(data_point);
        }
    }

    // Perform downsampling using the LTTB crate
    let downsampled_points: Vec<DataPoint> = lttb(data_points, threshold: n_out);

    // Convert the downsampled DataPoints back into indices that refer to the original dataset
    let mut sampled_indices: Vec<usize> = Vec::with_capacity(n_out);
    for dp: &DataPoint in downsampled_points.iter() {
        // Find the index of each downsampled DataPoint in the original dataset
        // This approach is simplistic and assumes that each (x, y) pair is unique
        if let Some(index: usize) = (0..x.len()).find(|&i: usize| x.is_valid(index: i) && y.is_valid(index:
            sampled_indices.push(index);
        }
    }
```

- Largest Triangle Three Buckets Algorithm
- MinMax LTTB
- Optimized for Arrow
- Handling NaN values
- Importantly*** this algorithm does not compute an average to reduce the dataset, instead it selectively chooses those points in the original dataset ensuring the detail is preserved (makes it good for subsequent calculations)

Arrow Resampler in Action

Visual demonstration of Arrow Resampler acting in real-time



BUT WHAT IF YOU HAVE *EVEN MORE* DATA AND USERS?

The Challenges of Scaling Up

Outlining scaling challenges when local solution falls short

- Local Caching Limitations
    - Impractical to cache terabytes with of data locally before processing
- Constrained Resources
    - Local systems often lack the computation power and memory capacity, subsequent concurrency issues
- Data Freshness and Synchronization
    - Data is coming in much faster than we can cache it, especially with real-time IoT data

Advancing to Server-Side Downsampling

Use the power of Databricks SQL to downsample server -side

- We can use Serverless SQL warehouse cluster to host the compute, utilizing Photon in the process
- Integrates with data in your warehouse without an intermediate step; performing calculations across the entire dataset
- Dramatically decreases the load on the local server, increasing concurrency
- The ability to handle resampling datasets which are continuously updating- streaming data into Databricks

MinMax LTTB in Databricks SQL

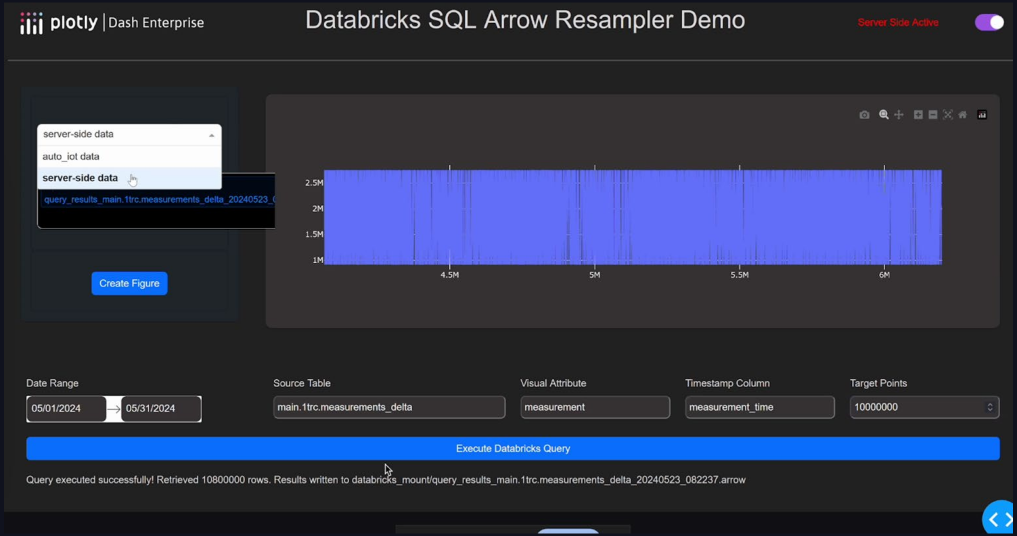
Sample SQL Code

SQL

```sql
SET use_cached_result = false;

-- Step 1 : Generated Filter Raw Dataset to minimize data we need to do the resampling on
-- this is historically what ONLY happened Server Side
WITH raw_data AS (
    SELECT
    IDENTIFIER({{ts_col}})::timestamp AS timestamp,
    IDENTIFIER({{visual_attribute}})::decimal AS value -- you can dynamically get this from client
    FROM IDENTIFIER({{source_table}}) AS s
    -- Filters generated by client resampling app (min/max boundaries of visual and filter selections in app)
    WHERE
    IDENTIFIER({{ts_col}})::timestamp BETWEEN '{{start_ts}}'::timestamp AND '{{end_ts}}'::timestamp
    -- Other filters from selctions in the app - FILTER as EARLY as possible
```

Server Side Downsample Demo

Combining the methods for the ultimate solution



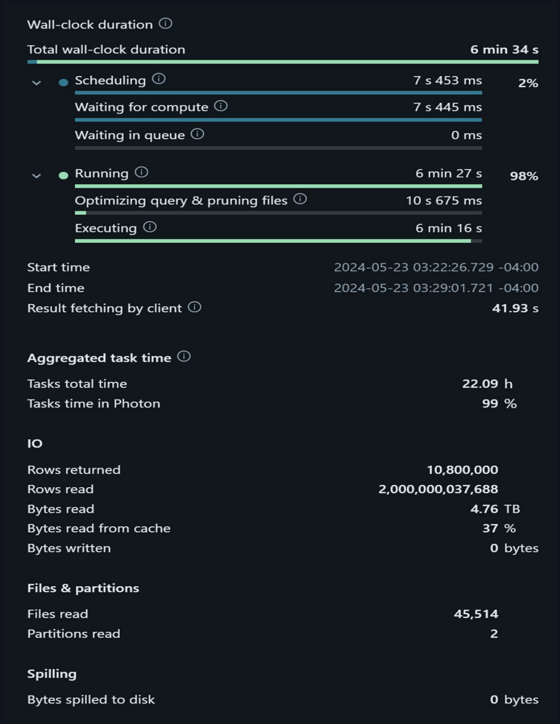
Date Range: 05/01/2024 → 05/31/2024
Source Table: main.1trc.measurements_delta
Visual Attribute: measurement
Timestamp Column: measurement_time
Target Points: 10000000

Query executed successfully! Retrieved 10800000 rows. Results written to databricks_mount/query_results_main.1trc.measurements_delta_20240523_082237.arrow

- Using a dataset of 2 trillion points of time-series IoT sensor data across 240 vehicles
- The dataset is 2.76 TB in size across approximately 512 partitioned files
- **How long does it take to downsample in Databricks SQL with Photon?**

Benchmarking the Algorithms

Contains an overview of the performance of downsampling on different kinds of compute



```
Wall-clock duration ⓘ
Total wall-clock duration                                  6 min 34 s
  Scheduling ⓘ                              7 s 453 ms      2%
    Waiting for compute ⓘ                   7 s 445 ms
    Waiting in queue ⓘ                            0 ms
  Running ⓘ                                 6 min 27 s     98%
    Optimizing query & pruning files ⓘ     10 s 675 ms
    Executing ⓘ                             6 min 16 s
Start time                      2024-05-23 03:22:26.729 -04:00
End time                        2024-05-23 03:29:01.721 -04:00
Result fetching by client ⓘ                               41.93 s

Aggregated task time ⓘ
Tasks total time                               22.09 h
Tasks time in Photon                              99 %

IO
Rows returned                             10,800,000
Rows read                          2,000,000,037,688
Bytes read                                     4.76 TB
Bytes read from cache                            37 %
Bytes written                                     0 bytes

Files & partitions
Files read                                    45,514
Partitions read                                    2

Spilling
Bytes spilled to disk                             0 bytes
```

Databricks SQL Warehouse (Photon)

- Resample 2 trillion rows representing 4.76 TB of Data to 10 million in about 6 mins
- Subsequent resampling of overlapping data set takes even less time thanks to Delta cache
- On XL warehouse numbers similar for cheaper warehouses

Local Resampler using Rust

- Only limited by the amount of data we can store in the arrow files
- Can store approx 1 billion points in 20 GB of arrow file
- Can navigate to the point level with next to no latency. (Resamples millions of points with ms latency)

Evaluating Levels of Server-Side

Taking a look at the trade-off at different levels of integration

Minimal Server-Side	Optimized Server-Side	Only Server-Side
• Data is processed on the Databricks side using a MinMaxLTTB downsampling algorithm just once before passing that data to the client • Good for limited concurrency and when you want to plug a larger time frame into the client	• Data is downsampled on the server to meet the computing limitations of the client to ensure the user always has a smooth experience • Only run the the downsampling algo when it needs to (when you zoom out) • Good for higher concurrency and best balance between resource allocation and latency.	• The data is only ever downsampled on Databricks. • This solution is ideal for analyzing data sets which are being streamed into databricks in real-time and so can't utilize regular caching methods • **SQL queries are created dynamically from interacting with Plotly charts in all three of these implementations**

Synchronizing Databricks and Dash

The Advantages of using Plotly Dash Enterprise with Databricks

Advantages of Plotly

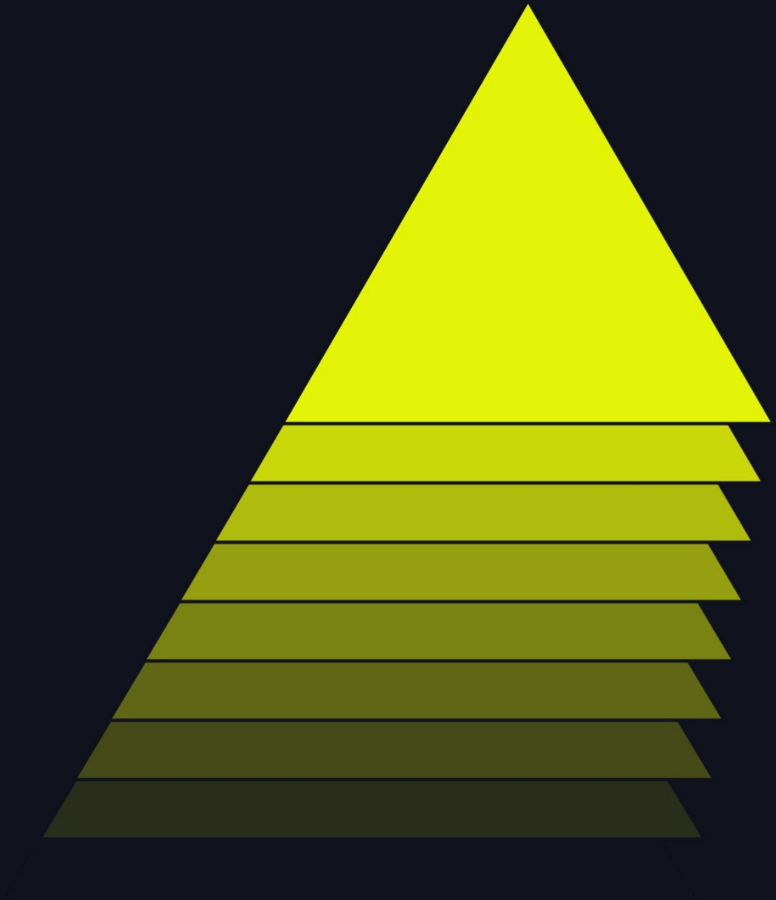
- Highly customizable and scalable production-grade data apps with access to an array of Databricks APIs to handle advanced workflows
- Persistent file storage system with robust caching capabilities
- Awesome collaborative IDE workspace experience

Chemistry with Databricks

- Databricks SQL Connector supports pulling data from Databricks into DE persistent file storage as Arrow
- Utilizes Photon for superfast resampling inside of Databricks so Plotly visuals can refresh without too much latency
- Can balance workload between Databricks and DE compute for optimal efficiency

DATA+AI SUMMIT
BY databricks

Wrapping Up



Sachin Seth