tinybird

Materialized Views

Materialized Views are a great way to improve your endpoint performance.

If you want low latency endpoints on top of millions or billions of rows of data, you often need to find ways to improve query performance. Complex queries run over large datasets are invariably slow, so you must find ways to recover speed you lose as data size increases. Of course, writing efficient SQL helps. But when you're processing billions of rows, even a simple SUM aggregation can become unbearably slow.

One such way to improve speed is to use Materialized Views.

The example on this page shows you how to create Materialized Views in Tinybird, using the UI and the CLI, to improve endpoint performance by over 50X.

Directly below, you can read a bit more about *why* and *when* to use Materialized Views, or skip straight to the guide.

> Before you get started creating MVs, we strongly recommend that you take some time to read through how they work in Tinybird, since Tinybird takes a unique approach to materialization that is different than most databases you may have used.

Why use Materialized Views?

When you're working in real-time, speed is everything. Queries that try to aggregate or filter over large datasets are doomed to suffer unacceptable levels of latency.

Materialized Views give you a way to pre-aggregate and pre-filter large Data Sources incrementally, adding simple logic using SQL to produce a more relevant Data Source with significantly fewer rows.

Put simply, Materialized Views shift computational load from query time to ingestion time, so your endpoints stay fast.

When should I use Materialized Views?

Materialized Views are used to accomplish 2 main goals:

1. Improve endpoint performance by reducing query speed and scan size

2. Simplifying query development by automating common filters and aggregations

As such, consider using a Materialized View if you are experiencing any of the following:

- A simple query taking too much time
- You're using the same aggregations on the same Data Source in multiple Pipes
- A single query is processing too much data
- You need to change the data schema for a new use case

Here are a few real-world examples of when a Materialized View makes sense:

- **Aggregating Sales per Product:** You're ingesting real-time sales events data, but you frequently need to query the total orders or revenue by product.
- **Aggregating Truck Routes per Day:** You need to save the routes that every truck in your fleet has performed each day. You can create a Materialized View with a groupArray, so you can easily return the routes per truck per day.
- **Error Logs:** You work with logs but just want to analyze those with the type "error". You can create a Materialized View to filter by just that type.
- **Real-Time Events:** You have a real-time application that doesn't need historical relevance. You can create a Materialized View to filter only timestamps within the last minute.
- **Changing Indexes:** You want to reuse a Data Source, but you have a new use case that requires you to filter by non-indexed columns. You can create a Materialized View and index by the columns you want to filter.
- **Extract data from geolocation JSON:** Ingested data for application sessions includes JSON with various user properties (e.g. location, device, browser...) and you want to be able to query over those individual properties. You can create a Materialized View to extract the data you need from a column containing JSON, and save those as new columns.

What is a Materialized View in Tinybird?

Materialized Views aren't unique to Tinybird, but Tinybird does handle them uniquely. In Tinybird, you can use Materialized Views just like you'd use regular Data Sources. In fact, Materialized Views are simply a special type of Data Source created by transforming an existing Data Source using a Transformation Pipe.

In this case, as with any Pipe in Tinybird, you can transform the data any way you want with SQL, using calculations such as counts, sums, averages, arrays; or even transformations like string manipulations (e.g. extracting JSON data) or joins.

Unlike traditional Pipes, however, these Transformation Pipes don't terminate with an Endpoint. Rather, the final node of the Pipe is used to create a special kind of Data Source: The Materialized View.

Follow the steps below to see how to create a Materialized View in the Tinybird UI.

Do Materialized Views affect usage?

We do not charge for one-time populates on Materialized Views, which means that when you create a new Materialized View, you can populate it with all existing data without any cost.

However, on-going incremental writes to Materialized Views do count towards your Tinybird usage.

Creating a Materialized View in the Tinybird UI

For this guide, the question we want to answer is: "What was the average purchase price of all products in each city on a particular day?"

Here's how we'd do that with the original Data Source.

Baseline endpoint performance on the original Data Source

First, let's understand baseline performance if we were to create an Endpoint on top of the original ingested Data Source.

Pipe node #1: Filtering and extracting

First, we want to filter the data to include only `buy` events. Simultaneously, we will normalize event timestamps to rounded days and extract `city` and `price` data from the `extra_data` column containing JSON.

NODE 1: FILTERING & EXTRACTING

```
SELECT
  toDate(date) day,
  JSONExtractString(extra_data, 'city') as city,
  JSONExtractFloat(extra_data, 'price') as price
FROM events
WHERE event = 'buy'
```

Pipe node #2: Averaging

Next, we aggregate, getting the average purchase price by city per day:

```
NODE 2: AVERAGING
SELECT
  day,
  city,
  avg(price) as avg
FROM only_buy_events_per_city
GROUP BY day, city
```

Pipe node #3: Creating a parameterized endpoint

Finally, we create a node that we will publish as an endpoint, adding a parameter to filter results to a particular day.

```
NODE 3: CREATING A PARAMETERIZED ENDPOINT
%
SELECT *
FROM avg_buy_per_day_and_city
WHERE day = {{Date(day, '2020-09-09')}}
```

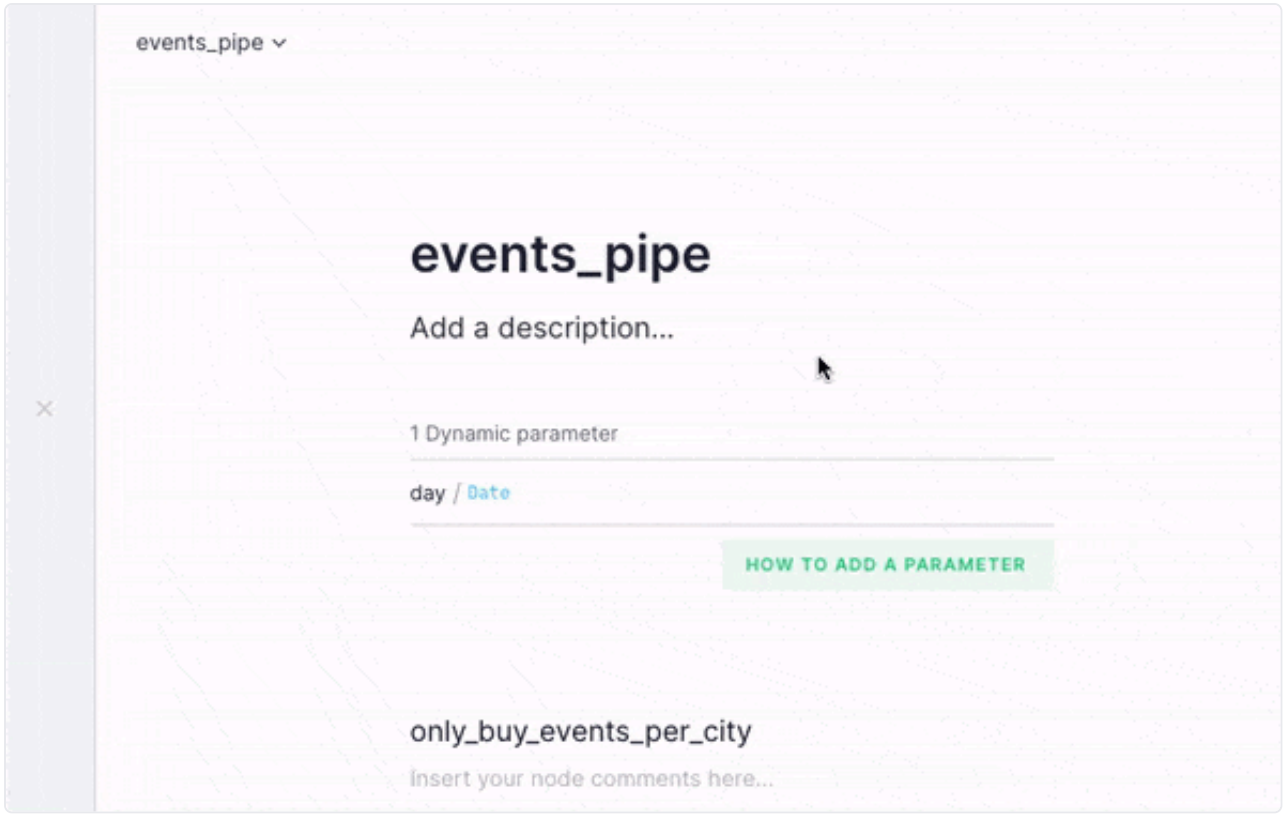
If you try this on your own, you'll see that this query processes 6.82MB every time we call it through the endpoint.

Now that we understand baseline performance, let's see how we can improve that with a Materialized View.

Creating the Materialized View

Remember, a Materialized View is simply formed by a Pipe that ends in the creation of a new Data Source (the Materialized View), instead of an Endpoint. So let's start by duplicating the existing pipe.

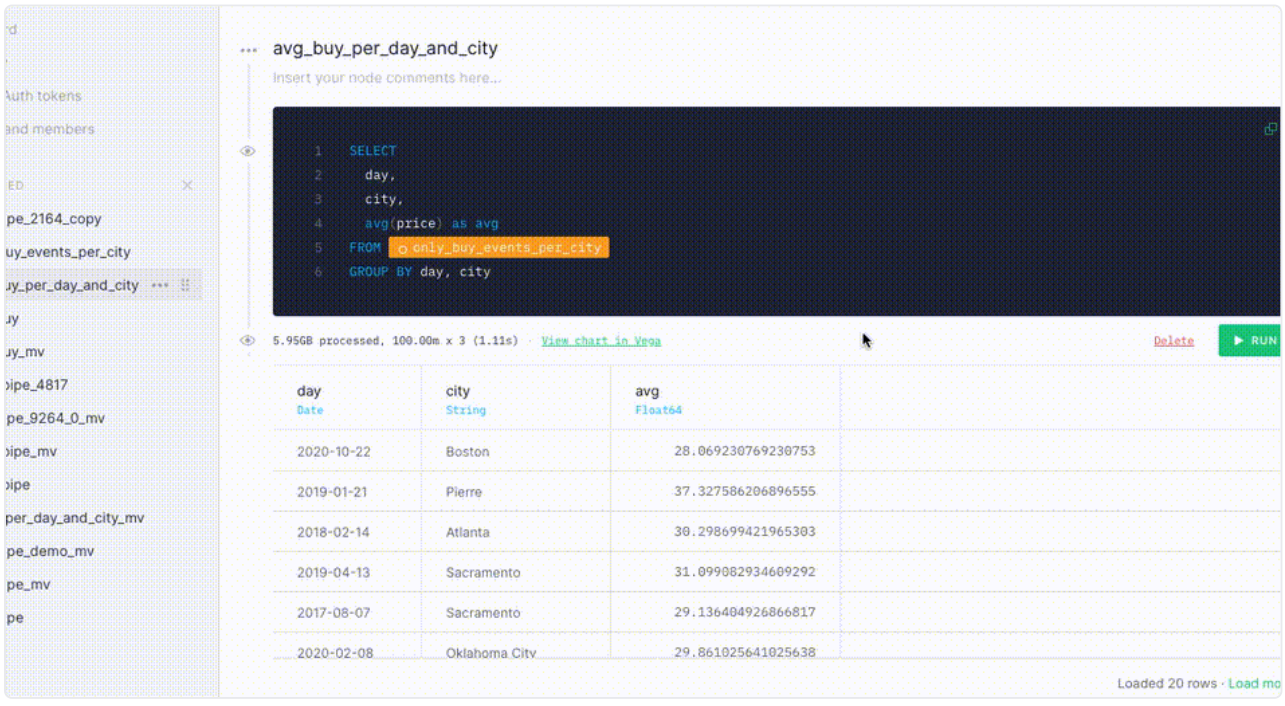
1. Duplicate the `events_pipe`, giving us another pipe with the same nodes as the baseline to use as a starting point.
2. Rename the pipe to `events_pipe_mv`, giving it a tidy, proper name (your future self will thank you for this).



Now, in this case we are going to materialize the second node in the Pipe, because it's the one performing the aggregation. The third node simply provides us a filter to create a parameterized Endpoint. We'll reuse that in a bit.

To create the Materialized View, simply:

1. Select the node options
2. Click "Create a Materialized View from this node"
3. Update the View settings as required (read more about this below).
4. Click "Create Materialized View"



That's it! Your Materialized View has been created as a new Data Source. By default, the name that Tinybird gives the Data Source is the name of the materialized Pipe node appended with `_mv`, in this case `avg_buy_per_day_and_city_mv`.

> ID your materalized views:
> We recommend that you append the names of all your Transformation Pipes and Materialized View Data Sources with _mv, or some other common identifier.

Populating Existing Data

When Tinybird creates a Materialized View, it initially only populates a partial set of data from the original Data Source. This allows you to quickly validate the results of the Materialized View.

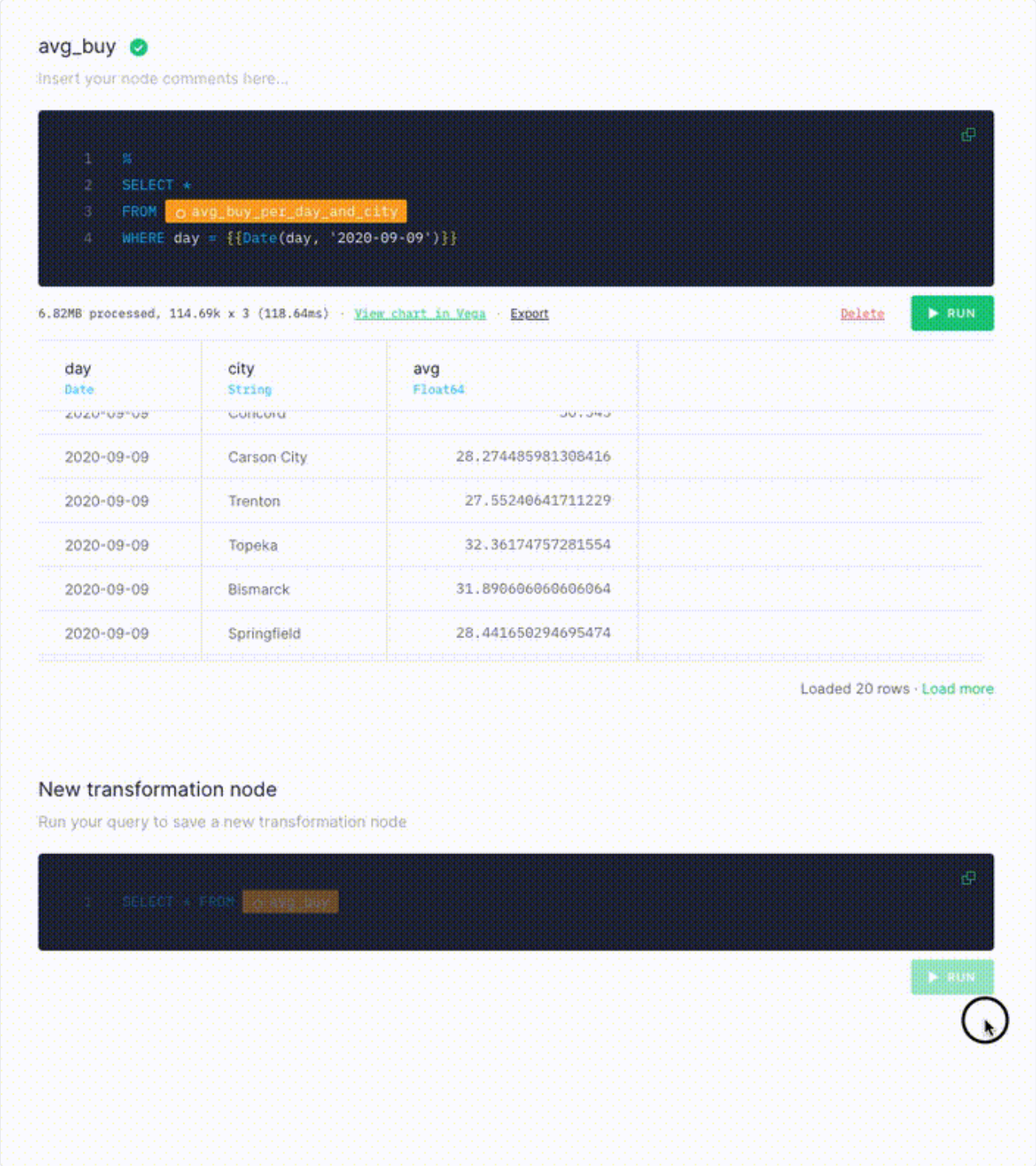
Once you have validated with the partial dataset, you can populate the Materialized View with all of the existing data in the Original Data source. To do so, click "Populate With All Data."

You now have a Materialized View Data Source that you can use to query against in your Pipes.

Testing Performance Improvements

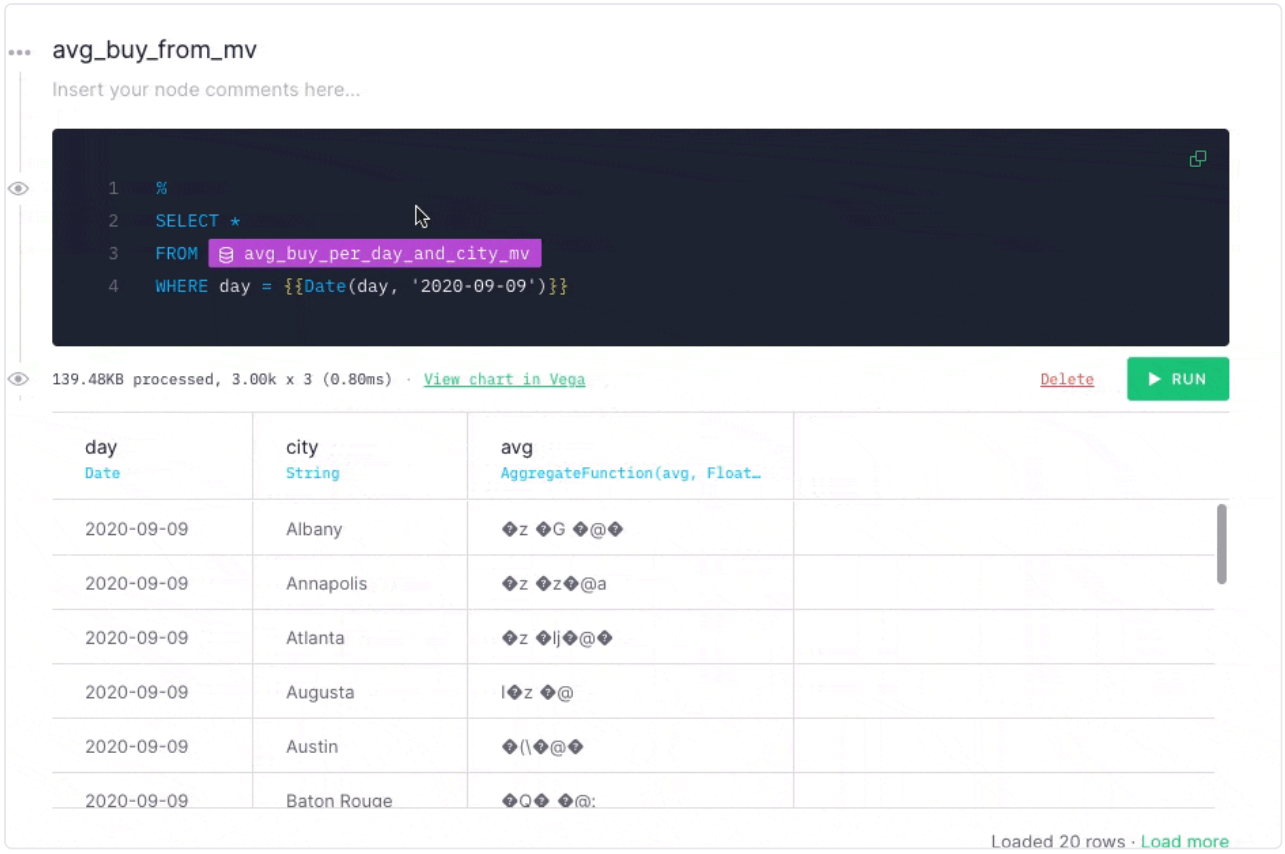
To test how the Materialized View has improved the performance of the endpoint, we'll go back to the original Pipe. In the original Pipe, do the following:

1. Copy the SQL from the original endpoint node, `avg_buy`.
2. Create a new transformation node, called `avg_buy_mv`.
3. Paste the query from the original endpoint node into your new node.
4. Update the query to select from your new Materialized View, `avg_buy_per_day_and_city_mv`.



Now, because this query is an aggregation, we'll need to rewrite the query slightly. Why? Data in Materialized Views in Tinybird exists in intermediate states. As new data is ingested, the data in the Materialized View gets appended in blocks of partial results. Every 10 minutes or so, a background process merges the appended partial results and saves them in the Materialized View.

Since we are processing data in real-time, we can't wait 10 minutes for the background process to complete. To account for this, we simply reaggregate in the endpoint query using the -merge combinator. In this example, we use an `avg` aggregation, so we need to use `avgMerge` to compact the results in the Materialized View.



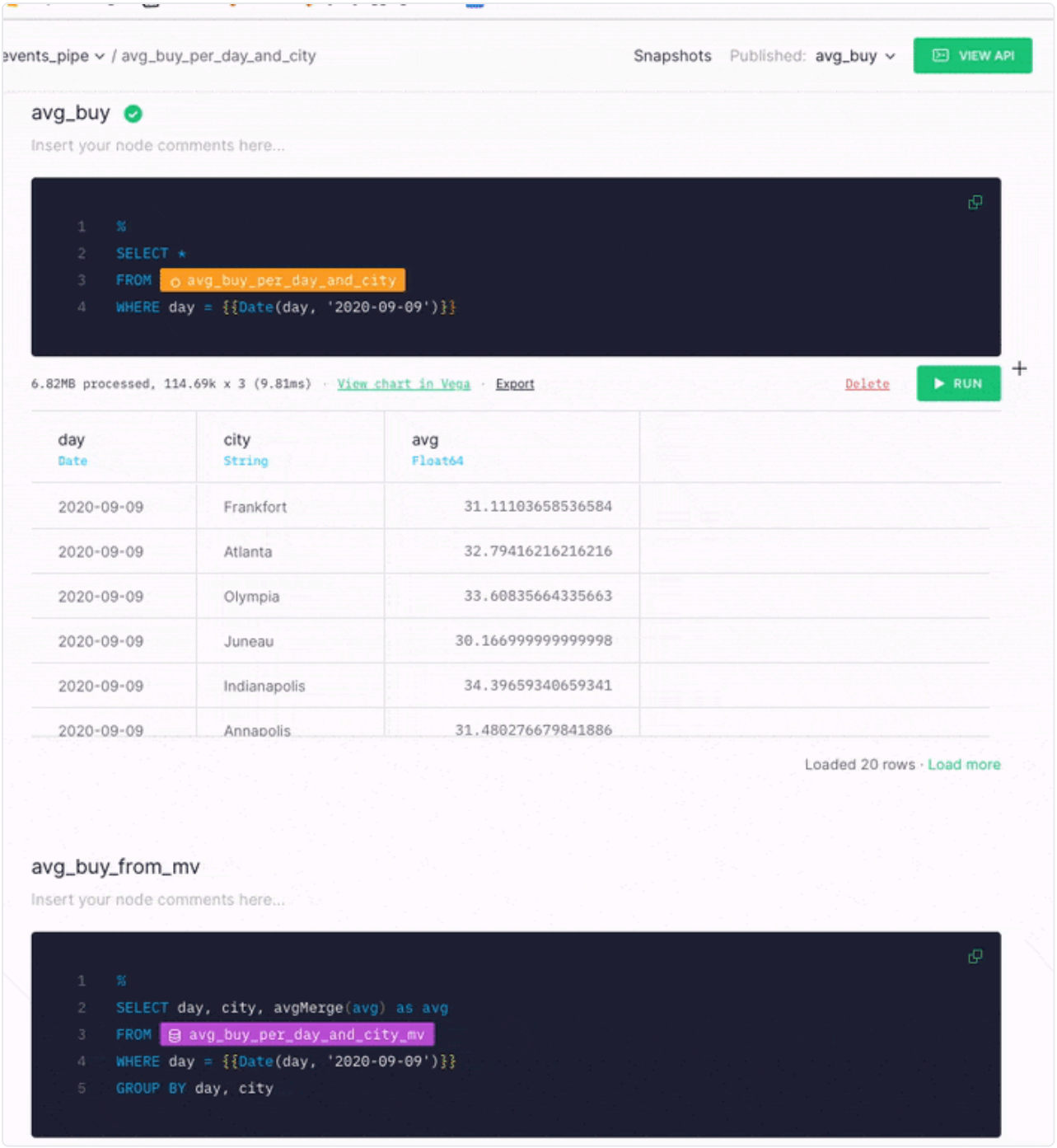
When you run your modified query, you should get the same results as you got when you ran the final node against the original Data Source.

This time, however, the performance has improved dramatically. The original endpoint query processed 5.4 MB of data and took, on average, 40ms to run (which is already pretty fast... this is a simple use case).

With the Materialized View, we get the exact same results, but process only 140kB of data (40x smaller) in an average of 20 ms. In other words, twice as fast at a fraction of the cost. What's not to love?

Pointing the endpoint at the new node

Now that we've seen how much the performance of the endpoint query has improved by using a Materialized View, we can easily change which node the endpoint uses. Just select the node dropdown, and choose the new node we created querying the Materialized View.



This way, we improve the endpoint performance while retaining the original endpoint URL, so applications which call that endpoint will see an immediate performance boost.

A practical example using the CLI

We have an `events` Data Source which for each action performed in an ecommerce website, stores a timestamp, the user that performed the action, the product, which type of action (`buy`, `add to cart`, `view`, etc.) and a json column containing some metadata, such as the price.

In this guide we will learn how to use the CLI to create Pipes and Materialized Views.

The `events` data source is expected to store billions of rows per month. Its data schema is as follows:

DEFINITION OF THE EVENTS.DATASOURCE FILE

```
SCHEMA >
    `date` DateTime,
    `product_id` String,
    `user_id` Int64,
    `event` String,
    `extra_data` String

ENGINE "MergeTree"
ENGINE_PARTITION_KEY "toYear(date)"
ENGINE_SORTING_KEY "date, cityHash64(extra_data)"
ENGINE_SAMPLING_KEY "cityHash64(extra_data)"
```

We want to publish an API Endpoint calculating the top 10 products in terms of sales for a date range ranked by total amount sold. Here's where Materialized Views can help us.

As you can see, after doing the desired transformations, all you need to do is set the TYPE parameter to `materialized` and add the name of the Data Source, which will materialize the results on-the-fly.

DEFINITION OF THE TOP PRODUCT PER_DAY.PIPE

```
NODE only_buy_events
DESCRIPTION >
    filters all the buy events

SQL >

    SELECT
        toDate(date) AS date,
        product_id,
        JSONExtractFloat(extra_data, 'price') AS price
    FROM events
    WHERE event = 'buy'



NODE top_per_day
SQL >
```

And then, do the rest in the Data Source schema definition for the Materialized View, which we will name `top_products_view`:

DEFINITION OF THE TOP PRODUCTS VIEW.DATASOURCE FILE

```
SCHEMA >
    `date` Date,
    `top_10` AggregateFunction(topK(10), String),
    `total_sales` AggregateFunction(sum, Float64)

ENGINE "AggregatingMergeTree"
ENGINE_SORTING_KEY "date"
```

As you can see, the destination Data Source uses an AggregatingMergeTree engine, which for each `date` will store the corresponding `AggregateFunction` for the top 10 products and the total sales.

Having the data precalculated as it is ingested, will make the API Endpoint run in real time, no matter the number of rows in the `events` Data Source.

Regarding the Pipe we will use to build the API Endpoint, which we will call `top_products_agg`, it will be as simple as:

DEFINITION OF THE TOP PRODUCTS PER DAY PIPE

```
NODE top_products_day
SQL >

    SELECT
        date,
        topKMerge(10)(top_10) AS top_10,
        sumMerge(total_sales) AS total_sales
    FROM dev__top_products_view
    GROUP BY date
```

> Aggregate function modes:
> Note that when preaggregating the Aggregate Function uses the mode `State`, while when getting the calculation it makes use of `Merge`.

Once it's done, let's push everything to our Tinybird account:

PUSH YOUR PIPES AND DATASOURCES USING THE CLI

```
tb push datasources/top_products_view.datasource --prefix dev
tb push pipes/top_product_per_day.pipe --prefix dev --populate
tb push endpoints/top_products_endpoint.pipe --prefix dev
```

Note that when pushing the `top_product_per_day.pipe` we use the `--populate` flag. This causes the transformation to be run in a job, and the Materialized View `top_products_view` to be populated.

Of course, you can repopulate Materialized Views at any moment:

COMMAND TO FORCE POPULATE THE MATERIALIZED VIEW

```
tb push pipes/top_product_per_day.pipe --prefix dev --populate --force
```

A practical example using version control

When using version control, you can create a new Materialized View in a Branch and then test the performance improvements safely in a Release before promoting.

For this example, we will assume that there will be no new appends to the Data Source during the whole process. But if this is not your case, be sure to check the corresponding example from the use case examples repo.

> There are some extra things to check if you have continuous (streaming) ingestion in your Data Source, as explained below.

1. Create a new Branch

From the UI or CLI, create a new Branch.

2. Make the changes

From the UI or CLI, create the Pipe that materializes into a new Data Source, and adapt the API Endpoint so that it reads from the new Materialized View.

3. Create a Pull Request

Bump your VERSION, increasing the **major** version (e.g. 0.0.0 to 1.0.0).

Customize the post-deployment actions to run a populate, for example:

POPULATE THE MATERIALIZED VIEW

```
tb --semver <VERSION> pipe populate <PIPE_NAME> --node <NODE> --wait
```

From the UI or CLI, commit and create a Pull Request in your Git provider that includes all of these changes.

4. Merge the Pull Request and Deploy

The automated CI tests will execute on your PR.

Once all tests have passed, merge the PR. This will create a new Preview Release in your Workspace.

5. Promote Preview Release to Live

Use the Preview Release to validate your changes.

After checking that everything is OK, promote the Preview Release to Live via the UI, CLI or CD.

You can find a detailed example of implementing this process [in this GitHub repo](https://github.com/tinybirdco/use-case-examples/tree/main

create_a_materialized_view_batch_ingest#tinybird-versions---create-a-materialized-view-with-batch-ingest) and this Pull Request

Limitations

Materialized Views work as insert triggers, which means a delete or truncate operation on your original Data Source won't affect the related Materialized Views.

As transformation and ingestion in the Materialized View is done on each block of inserted data in the original Data Source, certain operations such as `GROUP BY`, `ORDER BY`, `DISTINCT` and `LIMIT` might need a specific `engine` (such as `AggregatingMergeTree` or `SummingMergeTree`), which can handle data aggregations in this way.

Materialized Views generated using JOIN clauses are tricky. The resulting Data Source will be only automatically updated if and when a new operation is performed over the Data Source in the FROM.

You cannot create Materialized Views that depend on the UNION of several Data Sources.

Operations such as Window functions, `ORDER BY`, `Neighbor`, `DISTINCT` should not be used in Materialized Views.

A note on Populates

As described in the Master Materialized Views guide, populates are the process to move the data that was already in the original Data Source through to the new Materialized View.

If you have a continuous (streaming) ingestion into the original Data Source, the populate may produce duplicated rows in the Materialized View. The populate is executed as a separate job that will go partition by partition moving the existing data into the Materialized View. At the same time, the Materialized View is already automatically recieving new rows. Thus, there may be an overlap where the populate job moves a partition that includes rows that were already ingested into the Materialized View.

There are two strategies to handle this scenario, here and here.

Troubleshooting

Use the same alias in SELECT and GROUP BY

If you use an alias in the `SELECT` clause, you must re-use the same alias in the `GROUP BY`.

Take the following query as an example:

DIFFERENT ALIAS IN SELECT AND GROUP BY

```
SELECT
    key,
    multiIf(value = 0, 'isZero', 'isNotZero') as zero,
    sum(amount) as amount
FROM ds
GROUP BY key, value
```

The query above will result in the following error:

`Column 'value' is present in the GROUP BY but not in the SELECT clause`

To fix this, use the same alias in the `GROUP BY`:

GOOD: SAME ALIAS IN SELECT AND GROUP BY

```
SELECT
    key,
    multiIf(value = 0, 'isZero', 'isNotZero') as zero,
    sum(amount) as amount
FROM ds
GROUP BY key, zero
```

Don't use nested GROUP BYs

Avoid using a nested `GROUP BY` in the Pipe that creates a Materialized View. For example:

NESTED GROUP BY IN PIPE

```
SELECT
    product,
    count() as c
FROM (
    SELECT
        key,
        product,
        count() as orders
    FROM ds
    GROUP BY key, product
)
GROUP BY product
```

The query above would result in the following error:

```
Columns 'key, product' are present in the GROUP BY but not in the SELECT clause
```

To fix this, make sure you don't nest `GROUP BY` clauses:

SINGLE GROUP BY

```
SELECT
    key,
    product,
    count() as orders
FROM ds
GROUP BY key, product
```

Next steps

- Read through how Materialized Views work in Tinybird

Integrations

Amazon S3

Kafka data streams

Google Cloud Storage

Google BigQuery

Snowflake

Confluent

Use cases

In-Product Analytics

Operational Intelligence

Realtime Personalization

Anomaly Detection & Alerts

Usage Based Pricing

Sports Betting/Gaming

Smart Inventory Management

Serverless ClickHouse

Spain
Calle del Dr. Fourquet, 27
28012 Madrid

USA
41 East 11th Street 11th floor
New York, NY 10003