

Work with real-time data in an Eventhouse in Microsoft Fabric

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/query-data-kql-database-microsoft-fabric/1-introduction>

Introduction

Completed

An Eventhouse in Microsoft Fabric provides a data store for large volumes of data. It's a container that houses one or more KQL databases, each optimized for storing and analyzing real-time data that arrives continuously from various sources.

You can load data into a KQL database in an Eventhouse using an Eventstream or you can directly ingest data into a KQL database. Once you have ingested data, you can then:

- Query the data using Kusto Query Language (KQL) or T-SQL in a KQL queryset.
- Use Real-Time Dashboards to visualize the data.
- Use Fabric Activator to automate actions based on the data.

Understanding how KQL databases work helps you write effective queries to analyze real-time data. In this module, you'll learn about the characteristics that make KQL databases ideal for real-time data, then apply this knowledge by exploring KQL querying techniques and database objects like materialized views and stored functions.

How do KQL databases work with real-time data?

KQL databases automatically partition data by ingestion time, making recent data quickly accessible while storing historical data for trend analysis. *Partitioning* means the database organizes data into separate storage locations based on when it arrived, so when you query for recent data, the database knows exactly where to search rather than scanning all the data.

Think of it like a digital conveyor belt - events flow in continuously, get organized automatically by when they arrive, and are immediately available for analysis while the stream keeps flowing.

This automatic time-based organization works because real-time data has a unique characteristic: it represents immutable events that happened at specific moments in time. *Immutable* means these events can't be changed once they've occurred - a temperature reading at 3:15 PM will always be that reading because it represents what actually happened at that moment. Since each event is permanently tied to when it happened, this creates what we call **time-series data** - data where the timestamp is often as important as the event itself.

Time-series data follows an append-only pattern where new events are continuously added, and data is rarely updated or deleted because events represent what actually happened at specific moments. This is fundamentally different from traditional relational databases, where you typically update existing records and maintain relationships between different data tables.

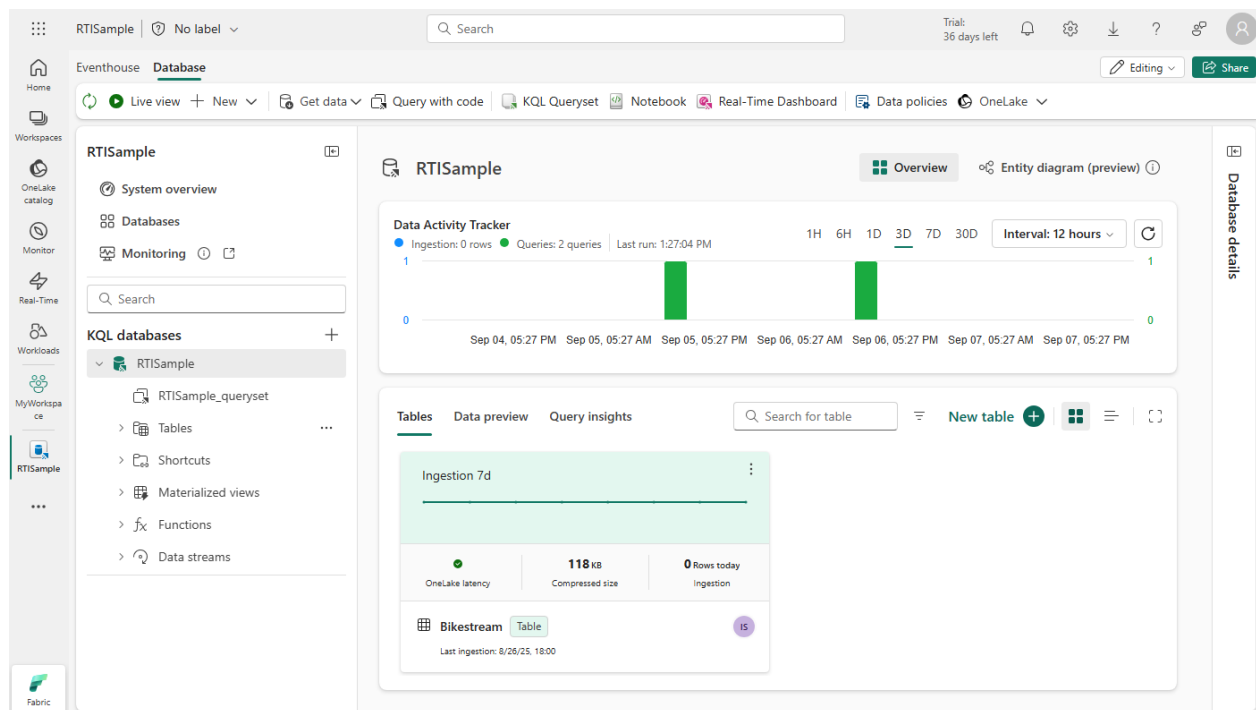
2. Get started with an Eventhouse

<https://learn.microsoft.com/en-us/training/modules/query-data-kql-database-microsoft-fabric/2-get-started-with-kql-queries>

Get started with an Eventhouse

Completed

When you create an Eventhouse, a default KQL database is automatically created with the same name. An Eventhouse contains one or more KQL databases, where you can create tables, stored procedures, materialized views, functions, data streams, and shortcuts to manage your data. You can use the default KQL database or create other KQL databases as needed.



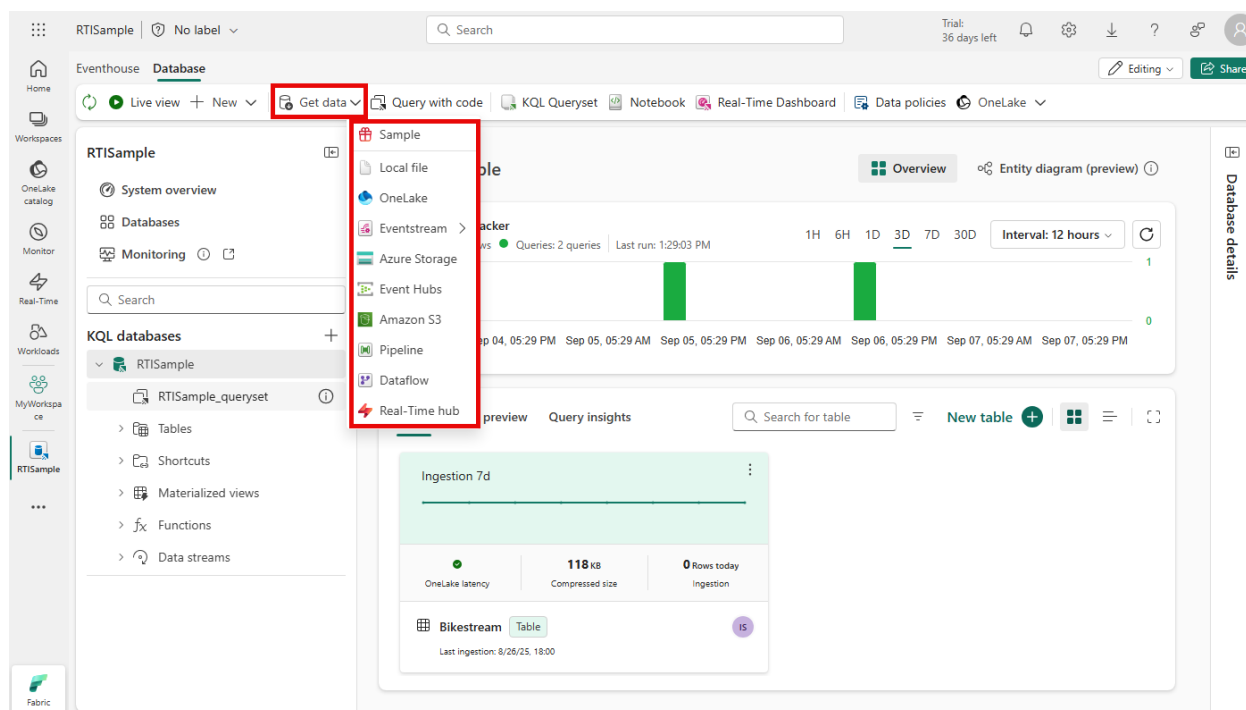
Work with data in your Eventhouse

There are several ways to access and work with data in a KQL database within an Eventhouse:

Data ingestion

You can ingest data directly into your KQL database from various sources:

- Local files, Azure storage, Amazon S3
- Azure Event Hubs, Fabric Eventstream, Real-Time hub
- OneLake, Data Factory copy, Dataflows
- Connectors to sources such as Apache Kafka, Confluent Cloud Kafka, Apache Flink, MQTT (Message Queuing Telemetry Transport), Amazon Kinesis, Google Cloud Pub/Sub



Database shortcuts

You can create **database shortcuts** to existing KQL databases in other eventhouses or Azure Data Explorer databases. These shortcuts let you query data from external KQL databases as if the data were stored locally in your eventhouse, without actually copying the data.

OneLake availability

You can enable **OneLake availability** for individual KQL databases or tables, making your data accessible throughout the Fabric ecosystem for cross-workload integration with Power BI, Warehouse, Lakehouse, and other Fabric services.

Query data in a KQL database

To query data in a KQL database, you can use **KQL** or **T-SQL** in *KQL querysets*. When you create a KQL database, an attached KQL queryset is automatically created for running and saving queries.

Basic KQL syntax

KQL uses a pipeline approach where data flows from one operation to the next using the pipe (|) character. Think of it like a funnel - you start with an entire data table, and each operator filters, rearranges, or summarizes the data before passing it to the next step. The order of operators matters because each step works on the results from the previous step.

Important

KQL is case-sensitive for everything including table names, column names, function names, operators, keywords, and string values. All identifiers must match exactly. For example, `TaxiTrips` is different from `taxitrips` or `TAXITRIPS`.

Here's an example that shows the funnel concept:

```
TaxiTrips
| where fare_amount > 20
| project trip_id, pickup_datetime, fare_amount
| take 10
```

This query starts with all data in the `TaxiTrips` table, filters it to show only trips with fares over \$20, selects specific columns using the **project** operator, and uses the **take** operator to return the first 10 rows that match the criteria in the **where** clause.

The simplest KQL query consists of a table name:

```
TaxiTrips
```

This returns all columns from the `TaxiTrips` table, but the number of rows displayed is limited by your query tool's default settings.

To retrieve a sample of data from potentially large tables, use the **take** operator:

```
TaxiTrips
| take 100
```

This returns the first 100 rows from the `TaxiTrips` table, which is useful for exploring data structure without processing the entire table.

You can also aggregate data:

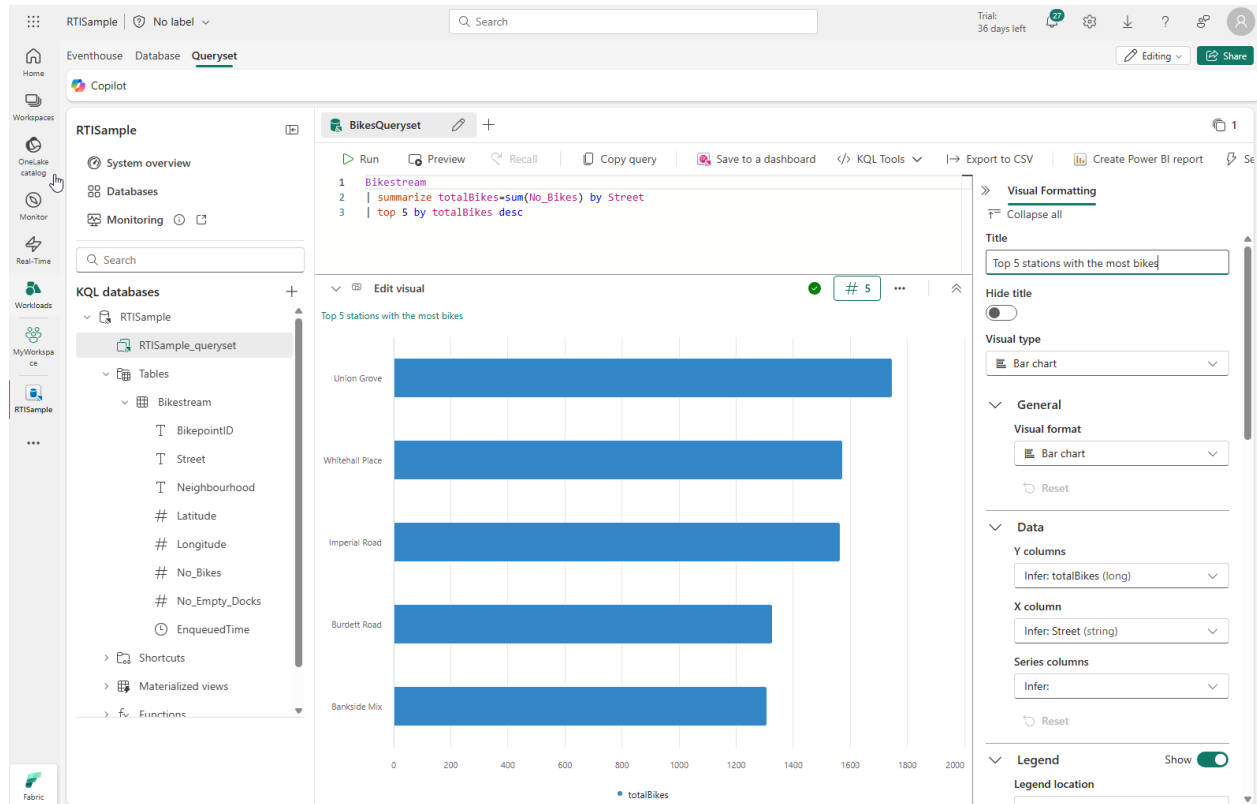
```
TaxiTrips
| summarize trip_count = count() by taxi_id
```

This returns a summary table showing the total number of trips (`trip_count`) for each unique `taxi_id`, effectively counting how many trips each taxi has made.

Analyze data with KQL queryset

KQL queryset provides a workspace for running and managing queries against KQL databases. The KQL queryset allows you to save queries for future use, organize multiple query tabs, and share queries with others for collaboration. The KQL queryset also supports T-SQL queries, allowing you to use T-SQL syntax alongside KQL for data analysis.

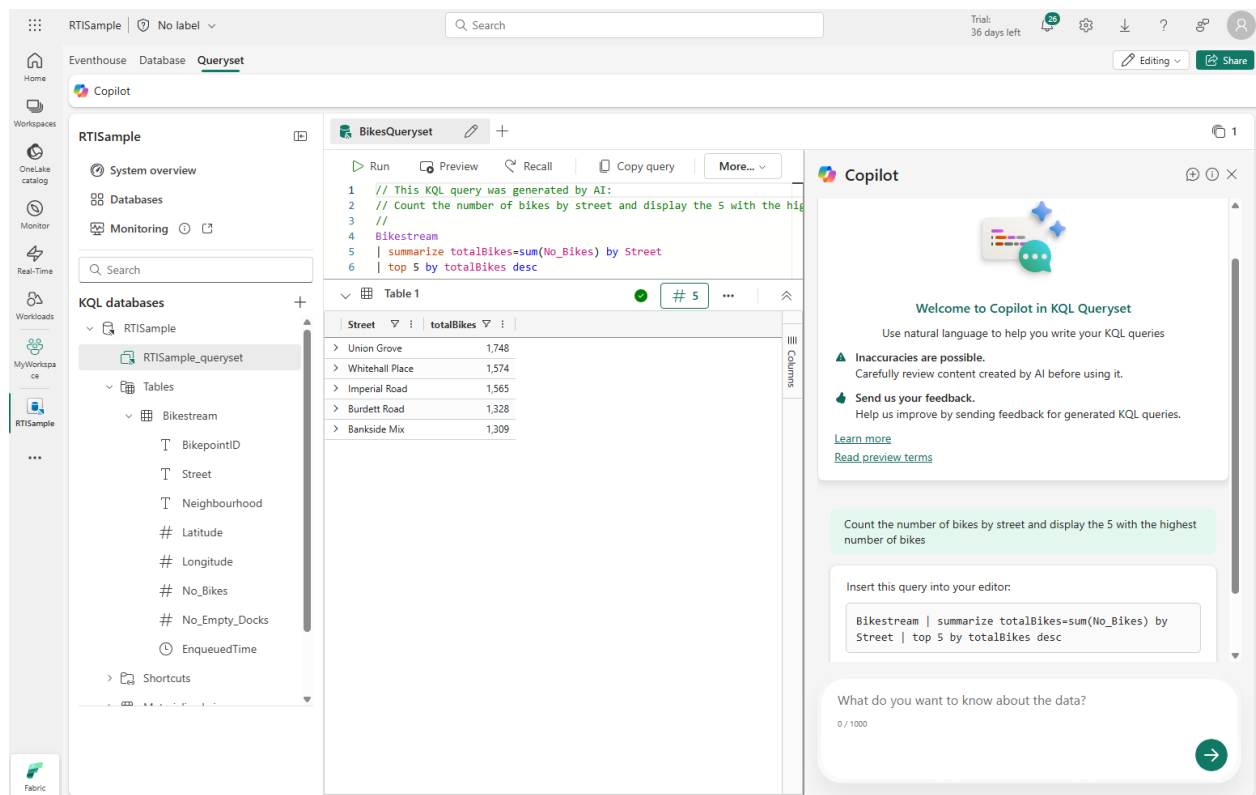
You can also create data visualizations while exploring your data, rendering query results as charts, tables, and other visual formats.



Use Copilot to assist with queries

For AI-based assistance with KQL querying, you can use Copilot for Real-Time Intelligence

When your administrator enables Copilot, you see the option in the queryset menu bar. Copilot opens as a pane to the side of the main query interface. When you ask a question about your data, Copilot generates the KQL code to answer your question.



3. Use KQL effectively

<https://learn.microsoft.com/en-us/training/modules/query-data-kql-database-microsoft-fabric/3-kql-best-practices>

Use KQL effectively

Completed

Understanding how to write efficient KQL queries is essential for getting good performance when working with Eventhouses. This unit covers key optimization techniques and explains why they matter for your queries.

Why query optimization matters

Query performance in KQL databases depends on the amount of data processed. When you understand how KQL processes data, you can write queries that:

- **Run faster** by reducing the data scanned - For example, instead of scanning millions of rows, filter early to process only thousands

- **Use fewer resources** - For example, selecting only 3 columns instead of all 50 columns reduces processing overhead
- **Work reliably with growing data** - For example, a query that works on 1 million rows today will still perform well when your data grows to 10 million rows

The key principle is: the less data your query needs to process, the faster it runs.

Understand key optimization techniques

Filter data early and effectively

Filtering reduces the amount of data that subsequent operations need to process, and KQL databases use indexes and data organization techniques that make early filtering especially efficient.

Time-based filtering is effective because Eventhouses typically contain time-series data:

```
TaxiTrips
| where pickup_datetime > ago(30min) // Filter first - uses time index
| project trip_id, vendor_id, pickup_datetime, fare_amount
| summarize avg_fare = avg(fare_amount) by vendor_id
```

Order your filters by how much data they eliminate - put filters that eliminate the most data first. Think of it like a funnel: start with the filter that removes the most rows, then apply more specific filters to the remaining data:

```
TaxiTrips
| where pickup_datetime > ago(1d) // Time filter first - eliminates most data
| where vendor_id == "VTS" // Specific vendor - eliminates some data
| where fare_amount > 0 // Value filter - eliminates least data
| summarize trip_count = count()
```

Reduce columns early

Projecting or selecting only the columns you need reduces resource usage. This is especially important when working with wide tables that have many columns.

```
TaxiTrips
| project trip_id, pickup_datetime, fare_amount // Select columns early
| where pickup_datetime > ago(1d) // Then filter
| summarize avg_fare = avg(fare_amount)
```

Optimize aggregations and joins

Aggregations and joins are resource-intensive operations because they need to process and combine large amounts of data. How you structure them can significantly affect query performance.

For aggregations, limit results when exploring data:

```
TaxiTrips
| where pickup_datetime > ago(1d)
| summarize trip_count = count() by trip_id, vendor_id
| limit 1000 // Limit results for exploration
```

For joins, put the smaller table first. When joining tables, KQL processes the first table to match with the second table. Starting with a smaller table means fewer rows to process, making the join more efficient.

```
// Good: Small vendor table first
VendorInfo
| join kind=inner TaxiTrips on vendor_id

// Avoid: Large taxi table first
TaxiTrips
| join kind=inner VendorInfo on vendor_id
```

4. Materialized views and stored functions

<https://learn.microsoft.com/en-us/training/modules/query-data-kql-database-microsoft-fabric/4-advanced-features>

Materialized views and stored functions

Completed

Now that you understand basic KQL querying and optimization techniques, let's explore materialized views and stored functions in eventhouses.

Understand materialized views

Materialized views are precomputed aggregations that solve a common performance challenge in KQL databases. KQL databases in eventhouses often contain millions or billions of rows from streaming data sources like IoT sensors, application logs, and other events. Running aggregation queries across these large datasets can take significant time and computing resources.

Materialized views store precomputed aggregation results and automatically update them as new data arrives. Instead of recalculating metrics from all historical data every time you query, the materialized view maintains the results and only processes the new data to update the aggregations. This provides instant results for dashboards and reports, even when working with massive datasets.

How automatic updates work

A materialized view consists of two parts that work together to provide always-current results:

- **A materialized part:** Precomputed aggregation results from data that has already been processed
- **A delta:** New data that has arrived since the last background update

When you query a materialized view, the system automatically combines both parts at query time to give you fresh, up-to-date results. This means materialized views always return current data, regardless of when the background materialization process last ran. Meanwhile, a background process periodically moves data from the delta part into the materialized part, keeping the precomputed results current. This approach provides the speed of precomputed results with the freshness of real-time data.

Create materialized views

A materialized view encapsulates a KQL **summarize** statement that automatically updates as new data arrives. Here's an example that tracks trip metrics by vendor and day:

```
.create materialized-view TripsByVendor on table TaxiTrips
{
    TaxiTrips
    | summarize trips = count(), avg_fare = avg(fare_amount), total_revenue = sum(fare_amount)
    by vendor_id, pickup_date = format_datetime(pickup_datetime, "yyyy-MM-dd")
}
```

Query materialized views

Once created, materialized views can be queried like regular tables:

```
TripsByVendor
| where pickup_date >= ago(7d)
| project pickup_date, vendor_id, trips, avg_fare, total_revenue
| sort by pickup_date desc, total_revenue desc
```

Understand stored functions

KQL includes the ability to encapsulate a query as a function, making it easier to repeat common queries. You can also specify parameters for a function, so you can repeat the same query with variable values.

Stored functions are useful in eventhouses where you have streaming data and multiple people writing queries. Instead of writing the same filtering or transformation logic repeatedly, you can define it once as a function and reuse it across different queries. Functions also help ensure that calculations are performed consistently when different team members need to apply the same logic to the data.

Create a function

```
.create-or-alter function trips_by_min_passenger_count(num_passengers:long)
{
    TaxiTrips
    | where passenger_count >= num_passengers
    | project trip_id, pickup_datetime
}
```

To call the function, use it like a table. In this example, the **trips_by_min_passenger_count** function is used to find 10 trips with at least three passengers:

```
trips_by_min_passenger_count(3)
| take 10
```

5. Exercise - Work with data in an Eventhouse

<https://learn.microsoft.com/en-us/training/modules/query-data-kql-database-microsoft-fabric/5-exercise>

Exercise - Work with data in an Eventhouse

Completed

Now it's your chance to work with real-time data.

In this exercise, you use KQL and T-SQL to query bike-sharing data in a KQL database.

This lab takes approximately **30** minutes to complete.

Important

You need a **Microsoft Fabric license** to complete this exercise. See [Getting started with Fabric](#) for details of how to enable a free Fabric trial license.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

6. Module assessment

<https://learn.microsoft.com/en-us/training/modules/query-data-kql-database-microsoft-fabric/6-knowledge-check>

Module assessment

Completed

7. Summary

<https://learn.microsoft.com/en-us/training/modules/query-data-kql-database-microsoft-fabric/7-summary>

Summary

Completed

In this module, you explored how to work with real-time data in eventhouses using KQL. You learned how to write effective KQL queries and use advanced features like materialized views and stored functions.

To learn more about KQL, explore these resources:

- [Kusto Query Language \(KQL\) overview](#)
- [Kusto Query Language best practices](#)
- [Copilot for Real-Time Intelligence](#)