Author: *Tyler Chessman* | See *github.com/tylerchessman/RTI_Medallion_Eg* for most recent updates.

Medallion Architecture in Fabric Real-Time Intelligence

Summary

Building a multi-layer, medallion architecture using Fabric Real-Time Intelligence (RTI) requires a different approach compared to traditional data warehousing techniques. But even transactional source systems can be effectively processed in RTI. To demonstrate, we'll look at how sales orders (created in a relational database) can be continuously ingested and transformed through a bronze, silver, and gold layer.

Pre-Requisites

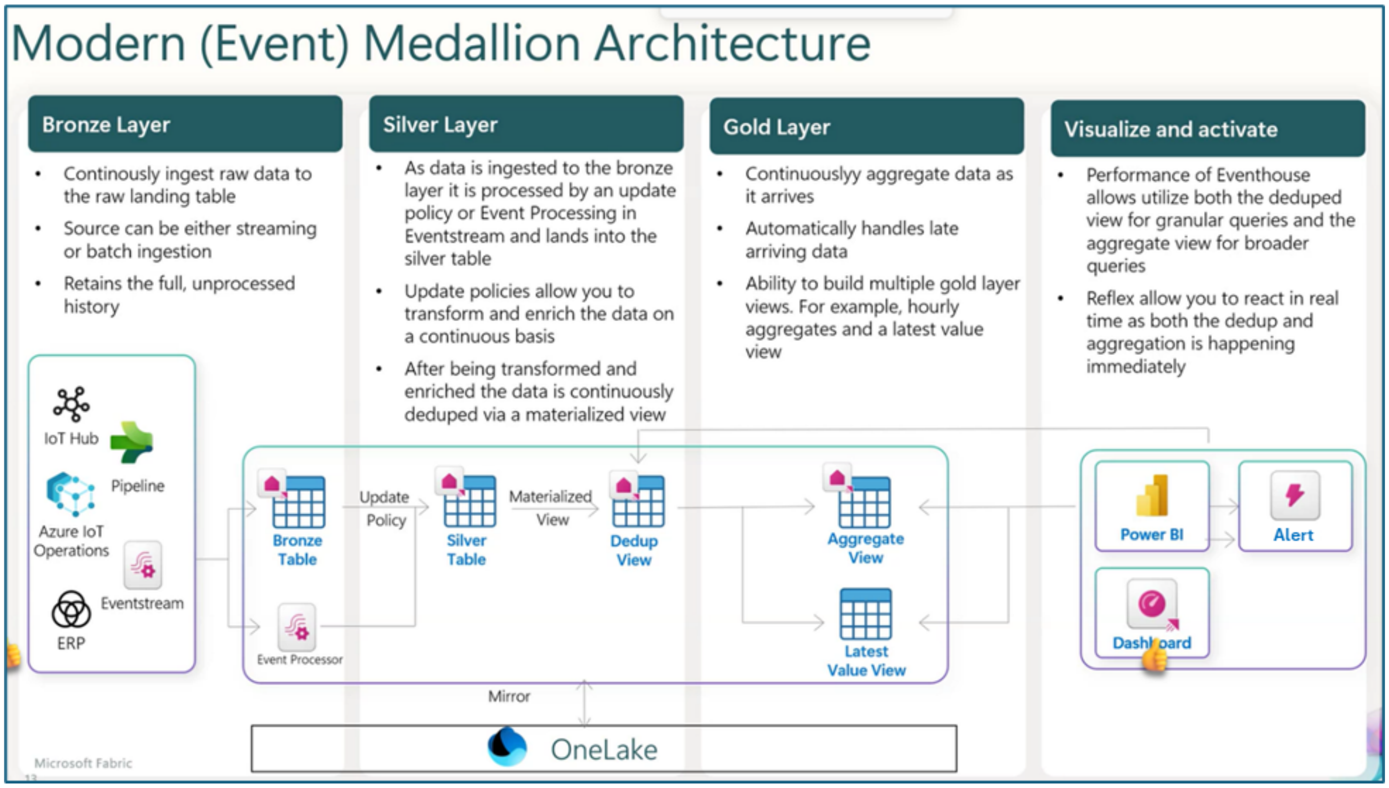
To implement this example in your own environment, get started by creating the AdventureWorksLT sample database in Azure. See here for information on how to install the sample database. Then, enable Change Data Capture (CDC). The **00_CDCSetup_AdWorksLT.sql** script (located in the CDCSetup folder) can be used to enable and configure CDC.

You'll also need access to a workspace associated to a Fabric-enabled capacity. Depending on your environment, you may be able to use a trial if an existing capacity is not available.

Medallion Architecture Primer

As big data and data lakes became popular, architectural patterns arose to help organize, process, and improve the quality of the data. The Medallion architecture defines three layers (bronze, silver, gold) where data can land and progress to serve different needs. The bronze (or raw) layer is typically used for initial ingestion of the data; data cleansing, validation, and transformation occurs in the silver layer – and additional modeling/aggregation takes place in a gold layer.

In Fabric RTI, a medallion architecture also contains these three layers. Data, whether it *arrives* in near real-time or in batches, is *continuously* processed and transformed along the bronze, silver, and gold layers – as shown in the following figure.



Modern (Event) Medallion Architecture

Bronze Layer
- Continously ingest raw data to the raw landing table
- Source can be either streaming or batch ingestion
- Retains the full, unprocessed history

Silver Layer
- As data is ingested to the bronze layer it is processed by an update policy or Event Processing in Eventstream and lands into the silver table
- Update policies allow you to transform and enrich the data on a continuous basis
- After being transformed and enriched the data is continuously deduped via a materialized view

Gold Layer
- Continuouslyy aggregate data as it arrives
- Automatically handles late arriving data
- Ability to build multiple gold layer views. For example, hourly aggregates and a latest value view

Visualize and activate
- Performance of Eventhouse allows utilize both the deduped view for granular queries and the aggregate view for broader queries
- Reflex allow you to react in real time as both the dedup and aggregation is happening immediately

Last Saved: *4/16/2025 6:20:00 PM*

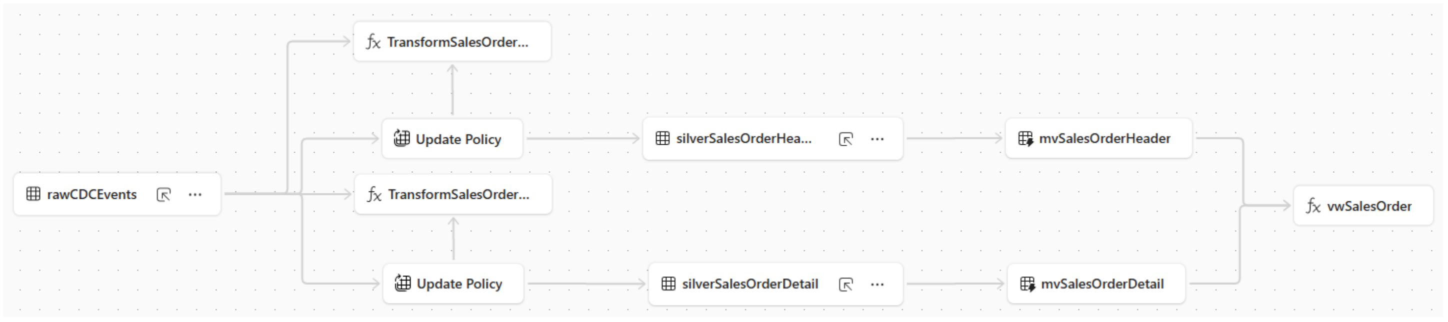
With RTI, data isn't *updated* or *deleted* (as often occurs in a traditional data warehouse). Instead, all events are *inserted*. Different processes are then used to reflect updates and/or filter out deleted records – but without overwriting the data.

To demonstrate, we start with a data source – the AdventureWorksLT database, with Change Data Capture (CDC) enabled for two tables – SalesOrderHeader and SalesOrderDetail. In Fabric, an Eventstream ingests and pushes these CDC events into a bronze eventhouse table. Update policies populate (and maintain) silver tables – and materialized views remove any duplicates. Finally, a "Latest Value" view joins header and details into a single, denormalized gold layer.

Fabric Eventhouse

Prior to connecting to the source database, we first create an eventhouse in our Fabric workspace. An eventhouse contains one or more KQL databases; similar to SQL Server, a KQL database can contain multiple tables, views, and functions. It supports its own query language, also referred to as KQL. For those unfamiliar with KQL, SQL queries are also supported – and can be converted to KQL as needed.

Bonus tip: A KQL Database in Fabric has a new entity diagram view, which shows the relationships among tables, functions, policies, views, etc. Peeking ahead, the entity diagram in our example will eventually look like the following:

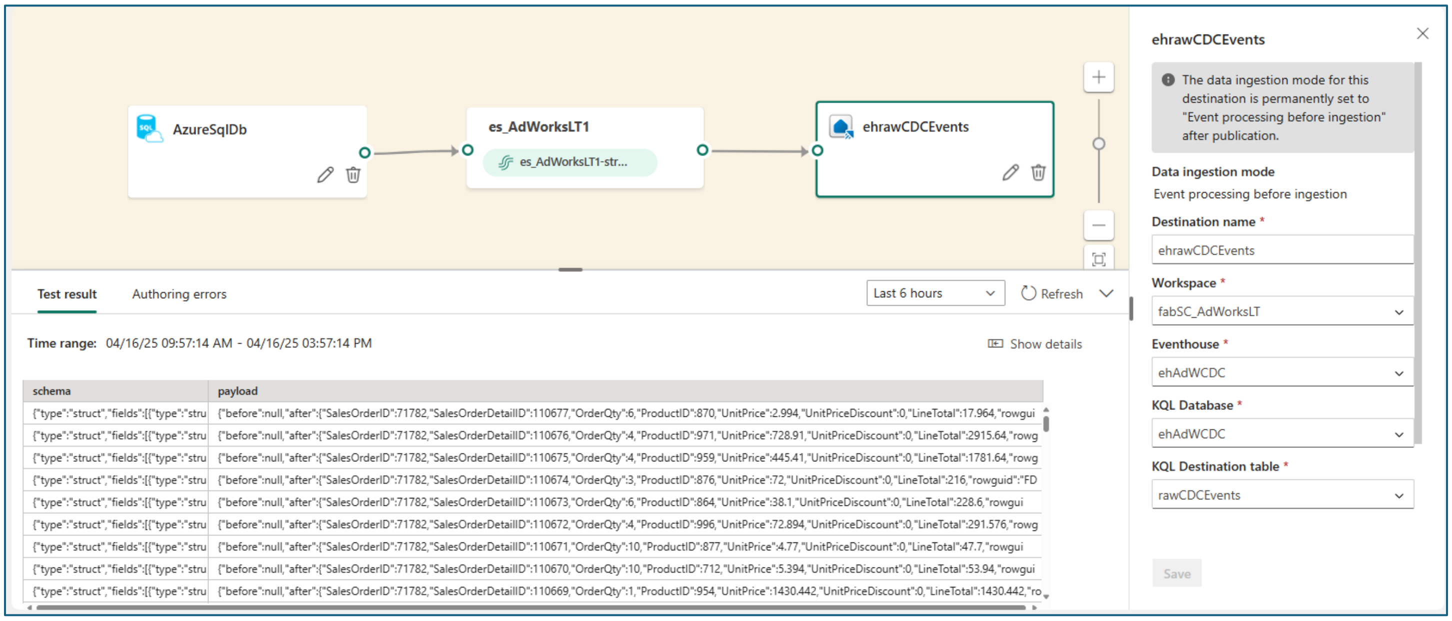


The Bronze Layer: Data Ingestion with an Eventstream

An Eventstream can connect to and process a variety of data sources (it can also expose an EventHub compatible endpoint so applications can push events directly to the stream). In this example, we connect to a CDC-enabled Azure SQL DB.

While an eventstream can also filter, join, split, or aggregate incoming events, our eventstream simply pushes the event data into a single landing table in the eventhouse – **rawCDCEvents.**

Author: *Tyler Chessman* | See github.com/tylerchessman/RTI_Medallion_Eg for most recent updates.



ehrawCDCEvents

The data ingestion mode for this destination is permanently set to "Event processing before ingestion" after publication.

Data ingestion mode: Event processing before ingestion

Destination name *: ehrawCDCEvents

Workspace *: fabSC_AdWorksLT

Eventhouse *: ehAdWCDC

KQL Database *: ehAdWCDC

KQL Destination table *: rawCDCEvents

Save

Time range: 04/16/25 09:57:14 AM - 04/16/25 03:57:14 PM

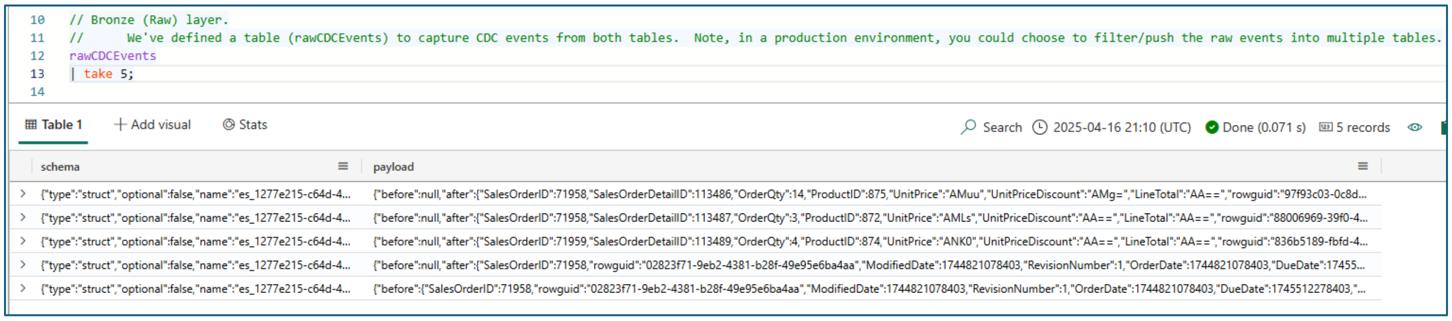
schema	payload
{"type":"struct","fields":[{"type":"stru	{"before":null,"after":{"SalesOrderID":71782,"SalesOrderDetailID":110677,"OrderQty":6,"ProductID":870,"UnitPrice":2.994,"UnitPriceDiscount":0,"LineTotal":17.964,"rowgui
{"type":"struct","fields":[{"type":"stru	{"before":null,"after":{"SalesOrderID":71782,"SalesOrderDetailID":110676,"OrderQty":4,"ProductID":971,"UnitPrice":728.91,"UnitPriceDiscount":0,"LineTotal":2915.64,"rowg
{"type":"struct","fields":[{"type":"stru	{"before":null,"after":{"SalesOrderID":71782,"SalesOrderDetailID":110675,"OrderQty":4,"ProductID":959,"UnitPrice":445.41,"UnitPriceDiscount":0,"LineTotal":1781.64,"rowg
{"type":"struct","fields":[{"type":"stru	{"before":null,"after":{"SalesOrderID":71782,"SalesOrderDetailID":110674,"OrderQty":3,"ProductID":876,"UnitPrice":72,"UnitPriceDiscount":0,"LineTotal":216,"rowguid":"FD
{"type":"struct","fields":[{"type":"stru	{"before":null,"after":{"SalesOrderID":71782,"SalesOrderDetailID":110673,"OrderQty":6,"ProductID":864,"UnitPrice":38.1,"UnitPriceDiscount":0,"LineTotal":228.6,"rowgui
{"type":"struct","fields":[{"type":"stru	{"before":null,"after":{"SalesOrderID":71782,"SalesOrderDetailID":110672,"OrderQty":4,"ProductID":996,"UnitPrice":72.894,"UnitPriceDiscount":0,"LineTotal":291.576,"rowg
{"type":"struct","fields":[{"type":"stru	{"before":null,"after":{"SalesOrderID":71782,"SalesOrderDetailID":110671,"OrderQty":10,"ProductID":877,"UnitPrice":4.77,"UnitPriceDiscount":0,"LineTotal":47.7,"rowgui
{"type":"struct","fields":[{"type":"stru	{"before":null,"after":{"SalesOrderID":71782,"SalesOrderDetailID":110670,"OrderQty":10,"ProductID":712,"UnitPrice":5.394,"UnitPriceDiscount":0,"LineTotal":53.94,"rowgui
{"type":"struct","fields":[{"type":"stru	{"before":null,"after":{"SalesOrderID":71782,"SalesOrderDetailID":110669,"OrderQty":1,"ProductID":954,"UnitPrice":1430.442,"UnitPriceDiscount":0,"LineTotal":1430.442,"ro

A couple of notes/tips when configuring the eventstream –

- When entering the initial list of Azure Tables, do not include spaces between tables names (as of April 2025, there is a UI glitch that won't let you save).
- The designer can create the rawCDCEvents table on your behalf. Alternatively, you can create it ahead of time with a simple KQL Statement:
  .create table rawCDCEvents (schema:dynamic,payload:dynamic);

Silver Layer: Table Update Policies and Materialized Views in the Eventhouse

The following query shows some of the records in **rawCDCEvents** (visit the KQL folder in the repo for a copy of all KQL queries and commands).



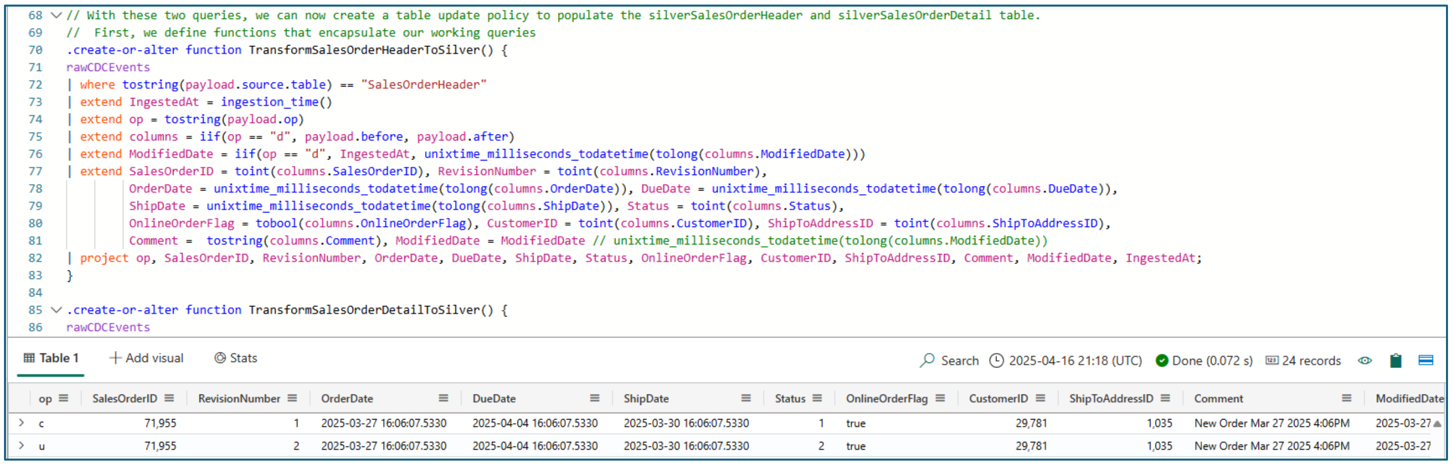
```
// Bronze (Raw) layer.
//      We've defined a table (rawCDCEvents) to capture CDC events from both tables.  Note, in a production environment, you could choose to filter/push the raw events into multiple tables.
rawCDCEvents
| take 5;
```

2025-04-16 21:10 (UTC) Done (0.071 s) 5 records

schema	payload
{"type":"struct","optional":false,"name":"es_1277e215-c64d-4...	{"before":null,"after":{"SalesOrderID":71958,"SalesOrderDetailID":113486,"OrderQty":14,"ProductID":875,"UnitPrice":"AMuu","UnitPriceDiscount":"AMg=","LineTotal":"AA==","rowguid":"97f93c03-0c8d...
{"type":"struct","optional":false,"name":"es_1277e215-c64d-4...	{"before":null,"after":{"SalesOrderID":71958,"SalesOrderDetailID":113487,"OrderQty":3,"ProductID":872,"UnitPrice":"AMLs","UnitPriceDiscount":"AA==","LineTotal":"AA==","rowguid":"88006969-39f0-4...
{"type":"struct","optional":false,"name":"es_1277e215-c64d-4...	{"before":null,"after":{"SalesOrderID":71959,"SalesOrderDetailID":113489,"OrderQty":4,"ProductID":874,"UnitPrice":"ANK0","UnitPriceDiscount":"AA==","LineTotal":"AA==","rowguid":"836b5189-fbfd-4...
{"type":"struct","optional":false,"name":"es_1277e215-c64d-4...	{"before":null,"after":{"SalesOrderID":71958,"rowguid":"02823f71-9eb2-4381-b28f-49e95e6ba4aa","ModifiedDate":1744821078403,"RevisionNumber":1,"OrderDate":1744821078403,"DueDate":17455...
{"type":"struct","optional":false,"name":"es_1277e215-c64d-4...	{"before":{"SalesOrderID":71958,"rowguid":"02823f71-9eb2-4381-b28f-49e95e6ba4aa","ModifiedDate":1744821078403,"RevisionNumber":1,"OrderDate":1744821078403,"DueDate":1745512278403,"...

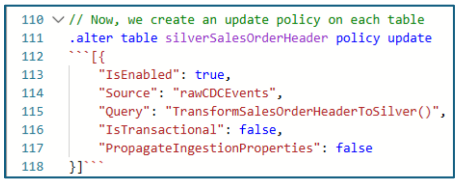
The payload column is what we're interested in – a JSON representation of a CDC event. KQL has good JSON support, so we can parse each event into something suitable for light transformation. To do this, we define a table update policy. Specifically, a query is encapsulated into a function; this function is then referenced in an update policy. The policy also specifies a target table.

Think of an update policy like a trigger in a database table; the policy runs every time data is ingested; there isn't anything needed in terms of scheduling the update. And, though it is called an *update* policy, we are only appending *new* data to the target table; there are no updates to *existing* rows. The following figures show the function and update policy used to populate the **silverSalesOrderHeader** table.



```
// With these two queries, we can now create a table update policy to populate the silverSalesOrderHeader and silverSalesOrderDetail table.
//  First, we define functions that encapsulate our working queries
.create-or-alter function TransformSalesOrderHeaderToSilver() {
rawCDCEvents
| where tostring(payload.source.table) == "SalesOrderHeader"
| extend IngestedAt = ingestion_time()
| extend op = tostring(payload.op)
| extend columns = iif(op == "d", payload.before, payload.after)
| extend ModifiedDate = iif(op == "d", IngestedAt, unixtime_milliseconds_todatetime(tolong(columns.ModifiedDate)))
| extend SalesOrderID = toint(columns.SalesOrderID), RevisionNumber = toint(columns.RevisionNumber),
        OrderDate = unixtime_milliseconds_todatetime(tolong(columns.OrderDate)), DueDate = unixtime_milliseconds_todatetime(tolong(columns.DueDate)),
        ShipDate = unixtime_milliseconds_todatetime(tolong(columns.ShipDate)), Status = toint(columns.Status),
        OnlineOrderFlag = tobool(columns.OnlineOrderFlag), CustomerID = toint(columns.CustomerID), ShipToAddressID = toint(columns.ShipToAddressID),
        Comment =  tostring(columns.Comment), ModifiedDate = ModifiedDate // unixtime_milliseconds_todatetime(tolong(columns.ModifiedDate))
| project op, SalesOrderID, RevisionNumber, OrderDate, DueDate, ShipDate, Status, OnlineOrderFlag, CustomerID, ShipToAddressID, Comment, ModifiedDate, IngestedAt;
}

.create-or-alter function TransformSalesOrderDetailToSilver() {
rawCDCEvents
```

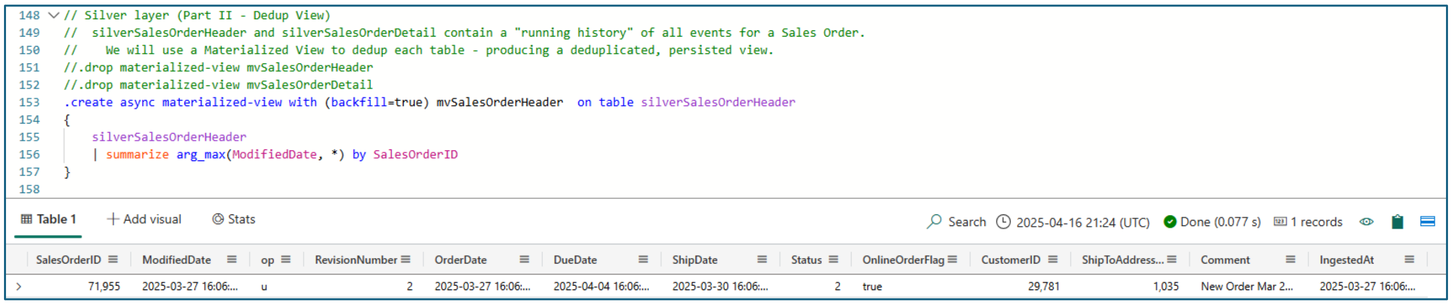
op	SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	CustomerID	ShipToAddressID	Comment	ModifiedDate
c	71,955	1	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	1	true	29,781	1,035	New Order Mar 27 2025 4:06PM	2025-03-27
u	71,955	2	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	2	true	29,781	1,035	New Order Mar 27 2025 4:06PM	2025-03-27



~~~
// Now, we create an update policy on each table
.alter table silverSalesOrderHeader policy update
```[{
    "IsEnabled": true,
    "Source": "rawCDCEvents",
    "Query": "TransformSalesOrderHeaderToSilver()",
    "IsTransactional": false,
    "PropagateIngestionProperties": false
}]```
~~~

Note that the silver tables can contain multiple rows for a given Order Header or Order Detail. For example, when a new Order Header is created, a row with an operation type **c** is ingested. Later, if the Order Header is updated, another row with an operation type **u** is ingested.

A Materialized View, using the summarize operation and arg_max() aggregation function, provides a persisted, deduplicated query that makes it easier for downstream users to analyze orders. Similar to a materialized view in a data warehouse, the view is maintained automatically (as new data arrives).

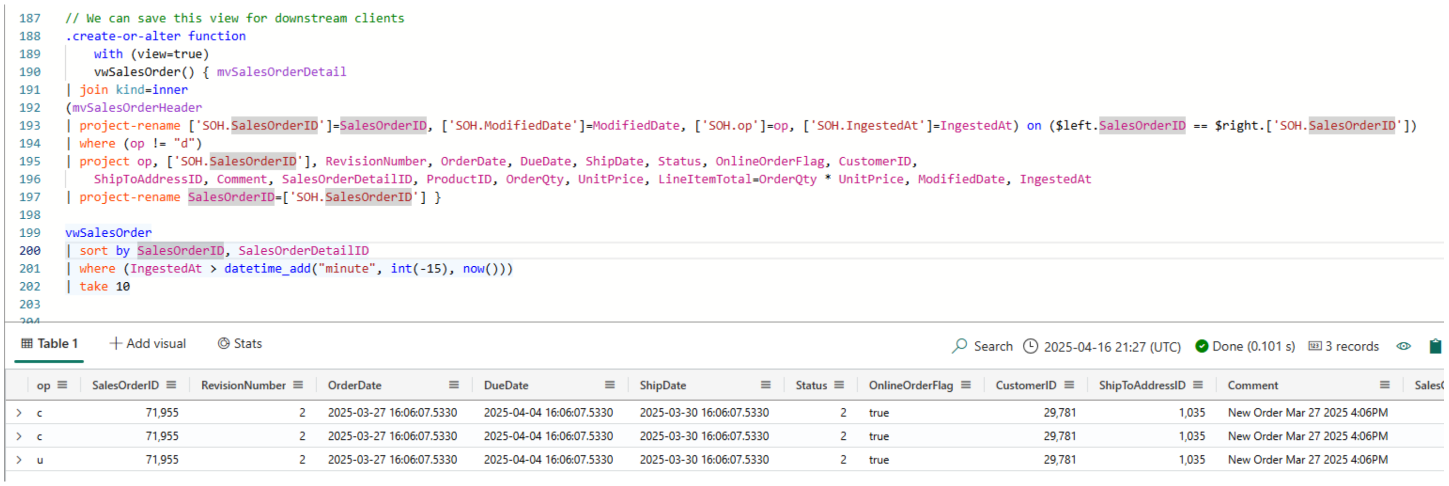


```
// Silver layer (Part II - Dedup View)
//  silverSalesOrderHeader and silverSalesOrderDetail contain a "running history" of all events for a Sales Order.
//    We will use a Materialized View to dedup each table - producing a deduplicated, persisted view.
//.drop materialized-view mvSalesOrderHeader
//.drop materialized-view mvSalesOrderDetail
.create async materialized-view with (backfill=true) mvSalesOrderHeader  on table silverSalesOrderHeader
{
    silverSalesOrderHeader
    | summarize arg_max(ModifiedDate, *) by SalesOrderID
}

```

SalesOrderID	ModifiedDate	op	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	CustomerID	ShipToAddress...	Comment	IngestedAt
71,955	2025-03-27 16:06:...	u	2	2025-03-27 16:06:...	2025-04-04 16:06:...	2025-03-30 16:06:...	2	true	29,781	1,035	New Order Mar 2...	2025-03-27 16:06:...

Gold Layer: The Latest (and Denormalized) View of Orders

Depending on the use case, creation of a gold layer may be done outside the eventhouse; for example, additional transformations may be done in a Power BI semantic model. For this scenario, we make use of a view to do a bit of final clean-up. Specifically, we join the two materialized views together – and filter our any deleted records.



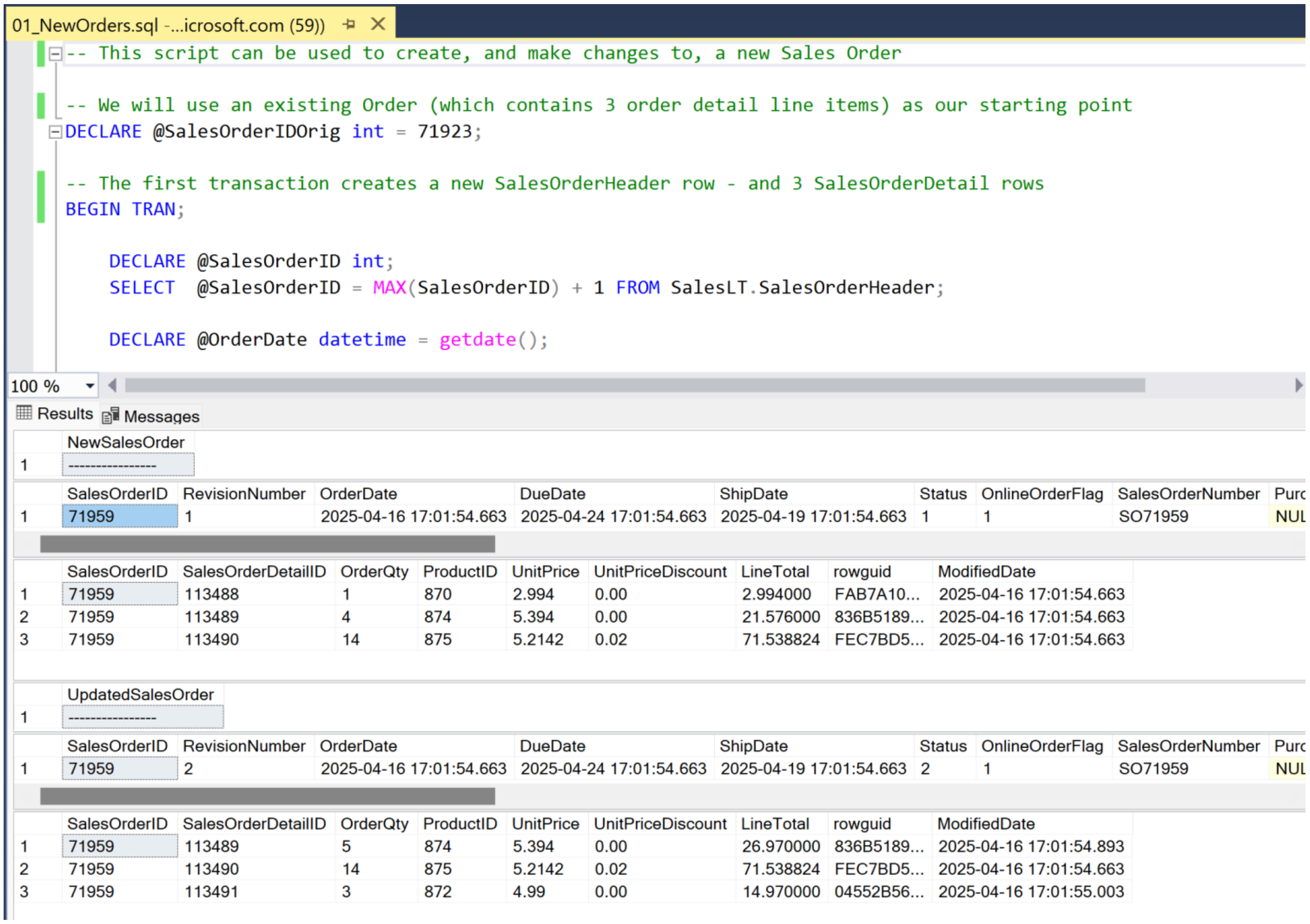
```
// We can save this view for downstream clients
.create-or-alter function
    with (view=true)
    vwSalesOrder() { mvSalesOrderDetail
| join kind=inner
(mvSalesOrderHeader
| project-rename ['SOH.SalesOrderID']=SalesOrderID, ['SOH.ModifiedDate']=ModifiedDate, ['SOH.op']=op, ['SOH.IngestedAt']=IngestedAt) on ($left.SalesOrderID == $right.['SOH.SalesOrderID'])
| where (op != "d")
| project op, ['SOH.SalesOrderID'], RevisionNumber, OrderDate, DueDate, ShipDate, Status, OnlineOrderFlag, CustomerID,
    ShipToAddressID, Comment, SalesOrderDetailID, ProductID, OrderQty, UnitPrice, LineItemTotal=OrderQty * UnitPrice, ModifiedDate, IngestedAt
| project-rename SalesOrderID=['SOH.SalesOrderID'] }

vwSalesOrder
| sort by SalesOrderID, SalesOrderDetailID
| where (IngestedAt > datetime_add("minute", int(-15), now()))
| take 10
```

Table 1 | Add visual | Stats | Search | 2025-04-16 21:27 (UTC) | Done (0.101 s) | 3 records

op	SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	CustomerID	ShipToAddressID	Comment
c	71,955	2	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	2	true	29,781	1,035	New Order Mar 27 2025 4:06PM
c	71,955	2	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	2	true	29,781	1,035	New Order Mar 27 2025 4:06PM
u	71,955	2	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	2	true	29,781	1,035	New Order Mar 27 2025 4:06PM

Try It Yourself

In addition to the scripts needed to configure CDC, the SQL script **01_NewOrders.sql** (located in the CDCSetup folder) can be used to create a new Sales Order. Run this script (from a tool like SSMS or Azure Data Studio) to create new rows in the database – and then inspect the events being ingested into the eventhouse by running KQL queries. The SQL script itself is pretty simple, but it covers all the basic operations – inserts, updates, and deletes.



01_NewOrders.sql -...icrosoft.com (59))

```
-- This script can be used to create, and make changes to, a new Sales Order

-- We will use an existing Order (which contains 3 order detail line items) as our starting point
DECLARE @SalesOrderIDOrig int = 71923;

-- The first transaction creates a new SalesOrderHeader row - and 3 SalesOrderDetail rows
BEGIN TRAN;

    DECLARE @SalesOrderID int;
    SELECT  @SalesOrderID = MAX(SalesOrderID) + 1 FROM SalesLT.SalesOrderHeader;

    DECLARE @OrderDate datetime = getdate();
```

100 %

Results | Messages

	NewSalesOrder
1	----------------

	SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	SalesOrderNumber
1	71959	1	2025-04-16 17:01:54.663	2025-04-24 17:01:54.663	2025-04-19 17:01:54.663	1	1	SO71959

	SalesOrderID	SalesOrderDetailID	OrderQty	ProductID	UnitPrice	UnitPriceDiscount	LineTotal	rowguid	ModifiedDate
1	71959	113488	1	870	2.994	0.00	2.994000	FAB7A10...	2025-04-16 17:01:54.663
2	71959	113489	4	874	5.394	0.00	21.576000	836B5189...	2025-04-16 17:01:54.663
3	71959	113490	14	875	5.2142	0.02	71.538824	FEC7BD5...	2025-04-16 17:01:54.663

	UpdatedSalesOrder
1	----------------

	SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	SalesOrderNumber
1	71959	2	2025-04-16 17:01:54.663	2025-04-24 17:01:54.663	2025-04-19 17:01:54.663	2	1	SO71959

	SalesOrderID	SalesOrderDetailID	OrderQty	ProductID	UnitPrice	UnitPriceDiscount	LineTotal	rowguid	ModifiedDate
1	71959	113489	5	874	5.394	0.00	26.970000	836B5189...	2025-04-16 17:01:54.893
2	71959	113490	14	875	5.2142	0.02	71.538824	FEC7BD5...	2025-04-16 17:01:54.663
3	71959	113491	3	872	4.99	0.00	14.970000	04552B56...	2025-04-16 17:01:55.003

Wrap-Up

Using a few innovative tools in the RTI toolbelt, we can continuously process transactional events - including updated and/or deleted records. While the approach differs from traditional data warehousing techniques, RTI can handle large volumes of data – while providing a full audit trail of changes for analysis and monitoring.