Dremio Data Reflections Overview and Best Practices

June 2020

Introduction

One of the limitations of traditional scale-out query engines is that they typically scan the entire dataset in order to answer a query. While Dremio's Apache Arrow-based execution engine and data source pushdowns provide exceptional performance, business analysts and data scientists often need to accelerate queries by orders of magnitude in order to achieve interactive performance on very large datasets. Data reflections is a patented feature in Dremio that delivers such query acceleration.

Data reflections are essentially data structures that are created precomputed and then utilized during query execution. These data structures maintain data in formats that are both logically and physically optimized. For example, the system may maintain a materialization of data (perhaps from multiple data sources) that is partially aggregated on specific dimensions, and then compressed and partitioned in a specific way.

If you are familiar with the concept of using Oracle's materialized views in conjunction with the optional query rewrite mode, Dremio's data reflections will seem somewhat familiar at a logical level. However, because Dremio is a data lake engine with the ability to elastically scale out engines, there are many unique concepts and best practices that are covered in this document.

When considering the cost of executing a query, there are several areas that can be very expensive to perform on large datasets:

- **Scans.** Depending on the source, the underlying system may be suboptimal for scan-intensive workloads. In addition, the format of the data may be inefficient for scans (e.g., JSON), and the source may be accessible only through a slow network connector.
- **Filters.** Predicates that filter the data to a subset may be expensive to perform. In addition, the resulting subset may be significantly smaller than the total dataset, meaning that more data was scanned than necessary.
- **Joins.** Joining data between datasets can be both CPU and memory intensive, especially when the datasets involved are larger than memory, or the datasets reside in different locations.
- **Aggregations.** Depending on the cardinality of the datasets, aggregations on one or more columns can be expensive to compute, especially when the datasets involved are larger than memory.
- **Projections.** Transformations applied to data within the query can be expensive to compute, especially for elaborate CASE statements and functions.
- **Sorts.** Sorting large, unsorted datasets can be memory intensive, especially when the datasets involved are larger than memory.

Data reflections improve query performance by precalculating many of these expensive operations, and stores the results in Dremio's highly optimized reflection store. Dremio's query optimizer automatically recognizes when an existing data reflection can and should be used to satisfy a request. It then automatically rewrites the query plan to use the data reflection in a way that is invisible to the end user.

Internally, queries go directly to the data reflection and not to the underlying physical dataset. In general, rewriting queries to use data reflections rather than physical datasets improves response time, sometimes by many orders of magnitude.

This paper explains the underlying concepts of data reflections, how they are created and managed in Dremio, and best practices for designing, optimizing their use and increasing the productivity or your data analysis tasks.

Please note, this document is a complement to the product documentation on *data reflections.*

Data Reflections Overview

WHAT ARE DATA REFLECTIONS

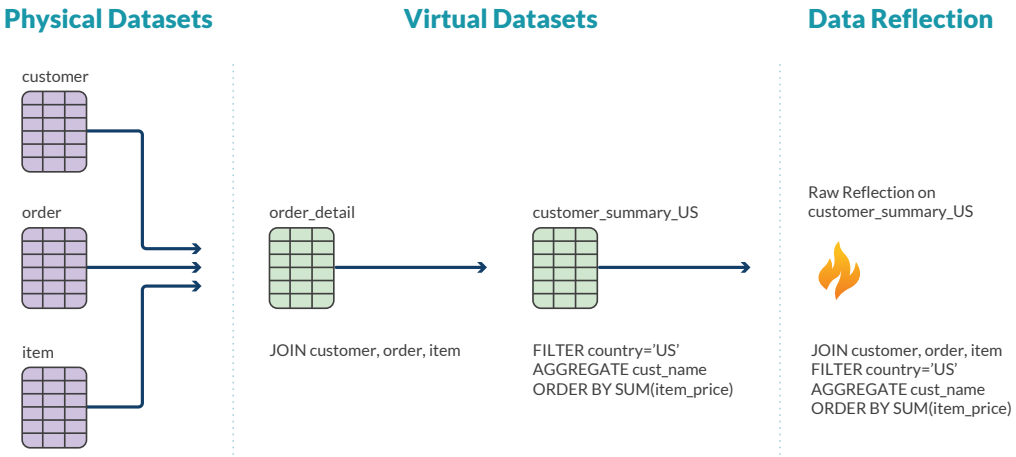
Dremio maintains physically optimized representations of source data known as data reflections. The query optimizer can accelerate a query by utilizing one or more data reflections to partially or entirely satisfy that query, rather than processing the raw data in the underlying data source.

A data reflection is associated with a single physical or virtual dataset for ease of configuration and management called the anchor dataset. Because virtual datasets can represent a combination of multiple datasets (via joins or subqueries), data reflections may include data from multiple data sources.

Dremio's query planner is capable of using a single data reflection to accelerate queries on different datasets that share one or more common ancestors. This flexibility increases the likelihood that queries will be accelerated, and allows administrators to maintain a relatively small number of data reflections that accelerate a diverse set of workloads.

At a high level, each data reflection is defined by the SQL expression that represents the expanded tree of all ancestor virtual datasets up to the underlying physical dataset.

Consider the following example:



The physical datasets (purple) are customer, order and item. These datasets could reside in a relational database, as collections of files on a file system (e.g., Amazon S3, Microsoft ADLS, and Hadoop HDFS), JSON records in a NoSQL database, Microsoft Excel spreadsheets, or any other supported data source in Dremio.

In this graphic, virtual datasets are represented in green. Like views in relational databases, virtual datasets represent logical expressions that ultimately refer to one or more physical datasets.

The process of evaluating a virtual dataset and its ancestor datasets is called dataset expansion. A virtual dataset that is based on a physical dataset is evaluated to include the data of the underlying physical datasets as well as any operators (e.g., projections, filters, aggregations, and joins) expressed in the virtual dataset. For virtual datasets based on one or more other virtual datasets, the expressions are expanded up the chain of dependency to the physical datasets at query time.

A virtual dataset called **order_detail** joins these three datasets together with the following SQL expression:

```
SELECT *
FROM ((order
INNER JOIN customer ON order.cust_id = customer.cust_id
INNER JOIN item on order.item_id = item.item_id))
```

Any time a user queries the virtual dataset order_detail, the query will be expanded to use the underlying physical datasets. The user could issue the query:

```
SELECT * FROM order_detail
WHERE price < 100
```

And this would be expanded to:

```
SELECT *
FROM ((order
INNER JOIN customer ON order.cust_id = customer.cust_id
INNER JOIN item on order.item_id = item.item_id))
WHERE item.price < 100
```

There is a second virtual dataset called **customer_summary_US** that is descendent from the order_detail virtual dataset. It is expressed with the following query:

```
SELECT cust_name, city, COUNT(item_id), SUM(item_price)
FROM order_detail
WHERE cust_country = 'US'
GROUP BY cust_name
ORDER BY SUM(item_price) DESC
```

This virtual dataset is a summary of all orders, grouped by customer and city, and ordered by the total amount they have spent across all orders. Issuing a `SELECT * FROM customer_summary_US` would expand into the following query:

```
SELECT cust_name, city, COUNT(item_id), SUM(item_price)
FROM ((order
INNER JOIN customer ON order.cust_id = customer.cust_id
INNER JOIN item on order.item_id = item.item_id))
WHERE cust_country = 'US'
GROUP BY cust_name, city
```

Similarly, getting the results for all customers in a particular city:

```
SELECT *
FROM customer_summary_US
WHERE city = 'Phoenix'
```

Would expand to:

```
SELECT cust_name, city, COUNT(item_id), SUM(item_price)
FROM ((order
INNER JOIN customer ON order.cust_id = customer.cust_id
INNER JOIN item on order.item_id = item.item_id))
WHERE cust_country = 'US' AND city='Phoenix'
GROUP BY cust_name, city
```

When a data reflection is created on the virtual dataset order_summary_US, it is defined by the fully expanded SQL expression:

```
SELECT cust_name, city, COUNT(item_id), SUM(item_price)
FROM ((order INNER JOIN customer ON order.cust_id =
customer.cust_id INNER JOIN item on order.item_id = item.
item_id)) WHERE cust_country = 'US' GROUP BY cust_name,
city
```

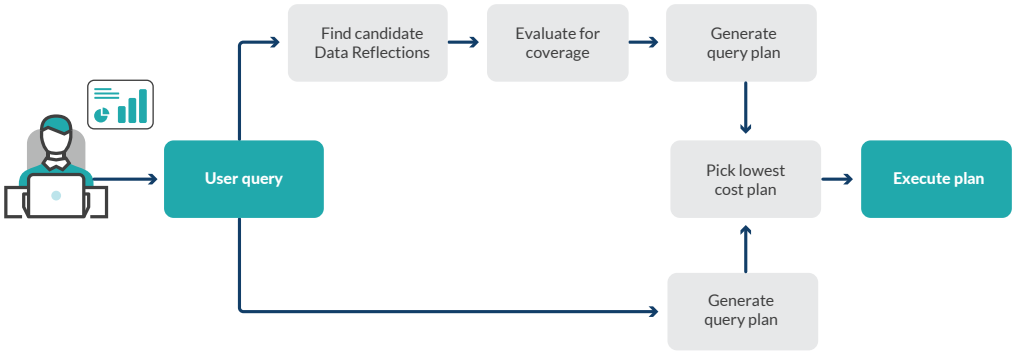
This data reflection then is the SQL expression (i.e., SELECT statement) that defines the fully expanded dataset, as well as the results of this expression materialized in the reflection store. Once the data reflection has been created, subsequent queries on the virtual dataset order_summary_US will cost out options for a pushdown on the underlying physical dataset as well as the data reflection, and will select the lowest cost in all cases.

In many cases the query plan based on the data reflection will produce the lowest cost, and the user's query will be rewritten to use the data in the data reflection rather than executing the fully expanded query on the physical datasets. By using the logically equivalent data reflection, the work to join, filter, aggregate and transform the data can be skipped. Instead, a simple scan of the data reflection is performed, producing the same results much more efficiently.

Just as queries are fully expanded to be executed on the physical datasets, data reflections are defined by their fully expanded form. This allows the query planner's matching process to match on reflections to the dataset that is being queried as well as any data reflections associated with ancestor or sibling datasets, allowing for more frequent acceleration of queries. If a data reflection existed on the order_detail virtual dataset, it would be matched by the query planner, and so would data reflections on the physical datasets themselves.

QUERY REWRITING

When Dremio receives a user query, it first determines whether any data reflections have at least one physical dataset in common with the query after both have undergone dataset expansion. All data reflections that pass this step are then evaluated to determine if they cover the query. The following sections explore some of the logic used to determine if the data reflection covers the query.



For data reflections that cover the query, Dremio will determine the cost of using the data reflection to execute the query. These costs are then compared to the cost of executing the query against the physical datasets, and the lowest cost query plan is selected for physical plan generation. Typically, using one or more data reflections will be less expensive than executing the query against the raw physical data.

EXPRESSION MATCHING

An expression that appears in a query can be replaced with a simple column in a data reflection provided the data reflection column represents a precomputed expression that matches with the expression in the query. If a query can be rewritten to use a data reflection, it will be faster. This is because data reflections contain precomputed calculations and do not need to perform expression computation.

Expression matching is done by first converting the expressions into canonical forms and then comparing them for equality. Therefore, two different expressions will be matched as long as they are equivalent to each other. Further, if the entire expression in a query fails to match with an expression in a data reflection, then subexpressions of it are tried in order to find a match. The subexpressions are tried in a top-down order to get maximal expression matching.

DATE FOLDING

Date folding is a specific form of expression matching rewrite. In this type of rewrite, a date range in a query is folded into an equivalent date range representing higher date granules. The resulting expressions representing higher date granules in the folded date range are matched with equivalent expressions in a data reflection. The folding of a date range into higher date granules such as months, quarters or years is done when the underlying datatype of the column is a date or timestamp. The expression matching is done based on the use of canonical forms for the expressions.

Date and timestamp are built-in datatypes which represent ordered time units such as seconds, days and months, and incorporate a time hierarchy (second -> minute -> hour -> day -> month -> quarter -> year). This hard-coded knowledge about date and timestamp is used in folding date ranges from lower-date granules to higher date granules. Specifically, folding a date value to the beginning of a month, quarter, year or to the end of a month, quarter, year is supported. For example, the date value 1-jan-1999 can be folded into the beginning of either year 1999 or quarter 1999-1 or month 1999-01. And, the date value 30-sep-1999 can be folded into the end of either quarter 1999-03 or month 1999-09.

SELECTION COMPATIBILITY CHECK

Dremio supports rewriting queries so that they will use data reflections in which the HAVING or WHERE clause of the data reflection contains a selection of a subset of the data in one or more datasets. A data reflection's WHERE or HAVING clause can contain a join, a selection, or both, and still be used by a rewritten query. Predicate clauses containing expressions, or selecting rows based on the values of particular columns, are examples of non-join predicates.

To perform this type of query rewrite, Dremio must determine if the data requested in the query is contained in, or is a subset of, the data stored in the data reflection. Selection compatibility is performed when both the query and the data reflection contain selections (non-joins). A selection compatibility check is done on the WHERE as well as the HAVING clause.

If the data reflection contains selections and the query does not, then the selection compatibility check fails because the data reflection is more restrictive than the query. This will register as "Did Not Cover Query" in the query profile on the Acceleration tab. If the query has selections and the data reflection does not, then selection compatibility check is not needed. Regardless, selections and any columns mentioned in them must pass the data sufficiency check.

JOIN COMPATIBILITY CHECK

In this check, the joins in a query are compared against the joins in a data reflection. In general, this comparison results in the classification of joins into three categories:

- **Common joins** that occur in both the query and the data reflection. These joins form the common subgraph. The common join pairs between the two must be of the same type.
- A **query delta join** is a join that appears in the query but not in the data reflection. Any number and type of delta joins in a query are allowed and they are simply retained when the query is rewritten with a data reflection. Upon rewrite, the data reflection is joined to the appropriate datasets in the query delta.
- A **record-preserving join** is a join that appears in the data reflection but not the query. All joins in a data reflection are required to be record preserving with respect to the result of common joins. A record-preserving join is one where, if two datasets called A and B are joined together, rows in dataset A will always match with rows in dataset B and no data will be lost, hence the term record preserving.

For more on record-preserving joins, see the section on starflake reflections.

GROUPING COMPATIBILITY CHECK

This check is required only if both the data reflection and the query contain a GROUP BY clause. Dremio first determines if the grouping of data requested by a query is exactly the same as the grouping of data stored in a data reflection. In other words, the level of grouping is the same in both the query and the data reflection.

If the grouping of data requested by a query is at a coarser level compared to the grouping of data stored in a data reflection, the optimizer can still use the data reflection to rewrite the query. If the data requested in the query is more fine-grained than that of the data reflection, then the data reflection will not cover the query.

AGGREGATION COMPUTABILITY CHECK

This check is required only if both the query and the data reflection contain aggregates. Dremio determines if the aggregates requested by a query can be derived or computed from one or more aggregates stored in one or more data reflections. For example, if a query requests AVG(X) and a data reflection contains SUM(X) and COUNT(X), then AVG(X) can be computed as SUM(X)/COUNT(X).

If the grouping compatibility check determines that the rollup of aggregates stored in a data reflection is required, then the aggregate computability check determines if it is possible to roll up each aggregate requested by the query using aggregates in the data reflection.

For example, SUM(sales) at the city level can be rolled up to SUM(sales) at the state level by summing all SUM(sales) aggregates in a group with the same state value. However, AVG(sales) cannot be rolled up to a coarser level unless COUNT(sales) is also available in the data reflection.

The argument of an aggregate such as SUM can be an arithmetic expression like A+B. Dremio will try to match an aggregate SUM(A+B) in a query with an aggregate SUM(A+B) or SUM(A) + SUM(B) stored in a data reflection. In other words, expression equivalence is used when matching the argument of an aggregate in a query with the argument of a similar aggregate in a data reflection.

APPLYING FILTERS TO DIMENSION COLUMNS

For aggregation reflections, Dremio can use any column configured as a dimension for the purposes of applying a filter. For example, if the dataset includes columns A, B and C as dimensions and the query is an aggregation query that includes a filter on A, B or C, Dremio can accelerate the query including the filter expression.

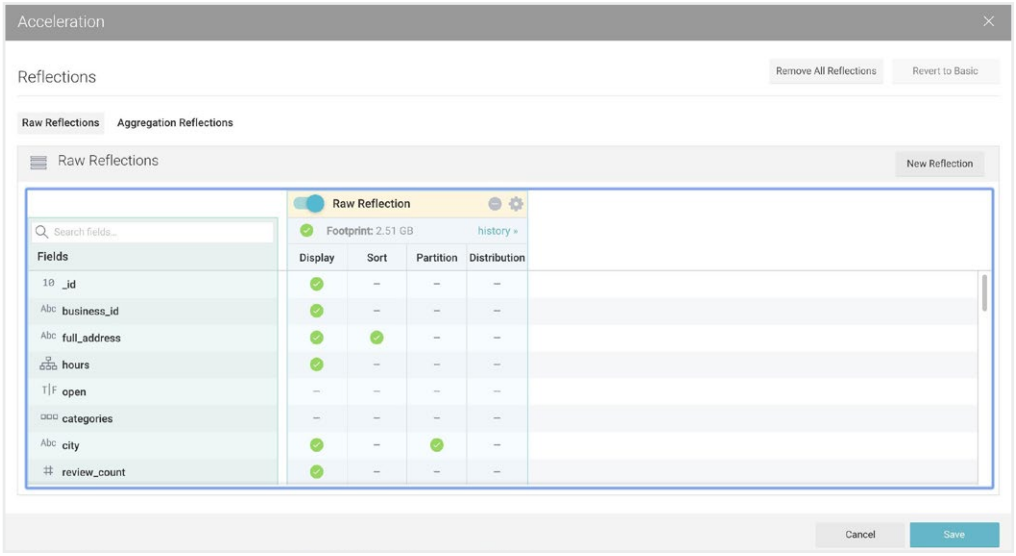
Raw Reflections

Dremio supports two fundamental types of data reflections: raw reflections and aggregation reflections. Many of the options for configuring and managing both types of data reflections are the same, but they each optimize different types of query patterns.

Raw reflections preserve row-level fidelity of the anchor dataset. A raw reflection includes one or more fields from the anchor dataset, and is sorted and partitioned by specific fields. You can use raw reflections to perform a number of optimizations:

USE CASE	HOW RAW REFLECTIONS HELP
"Needle in a haystack" -style queries that return relatively small numbers of records.	Raw reflections preserve row-level data in a form that is optimized for scans. Sorting and partitioning the data on specific columns allows Dremio's query planner to make use of how the data is physically organized to improve query performance.
Accelerate data from sources that are not optimized for scan-intensive workloads such as OLTP-optimized databases, document databases and row-oriented data formats.	Dremio maintains the data in a highly compressed, columnar form based on Apache Parquet. Dremio stores this data near Dremio's query engine which can be scaled out with additional nodes to support larger data volumes, greater concurrency and lower latency. If the physical data is JSON or row-oriented (e.g., CSV) then a raw reflection effectively columnarizes the data.
Offload analytical workloads from operational sources.	Dremio will automatically rewrite incoming queries to use data reflections instead of pushing the query into the physical source, thereby offloading analytical queries from source systems.
Vertically partition datasets with many columns.	For datasets with hundreds of columns, it is usually not the case that every query requests every column. Data reflections can be created on subsets of columns (e.g., groups of 20 columns), and Dremio's query planner will automatically use the minimum number of data reflections, thereby scanning far less data to make processing more efficient.
Horizontally partition large datasets.	When a column is selected for partitioning of the data reflection, Dremio will maintain physical partitions of the data within the reflection store. Dremio's query planner will perform partition pruning when appropriate to optimize query execution.

Dremio makes it easy to design raw reflections. For a given physical or virtual dataset, enter the data reflection configuration screen to select the appropriate options and to view the size and status at any time. You can create multiple data reflections on the same anchor dataset, each optimized for different workloads.



Aggregation Reflections

Aggregation reflections maintain summary data about the anchor dataset. An aggregation reflection includes one or more dimension and measure fields from the anchor dataset, that are sorted, partitioned and distributed by specified columns. Some of these columns are configured as dimensions that will be used in GROUP BY (or DISTINCT) statements, and other columns are configured as measures that will be used in calculations such as MAX, MIN, AVG, SUM and COUNT.

Dremio automatically maintains all measures for every permutation of combined dimensions in an aggregation reflection. With each additional dimension, additional measures are calculated and stored in the aggregation reflection. To understand the size impact of a new dimension, consider the number of unique values and multiply this times the number of unique values in other dimensions. Consider the following simple example:

COLUMN	NUMBER OF UNIQUE VALUES
State	50
Color	25
Size	10
Total rows in aggregation reflection	**12,500**

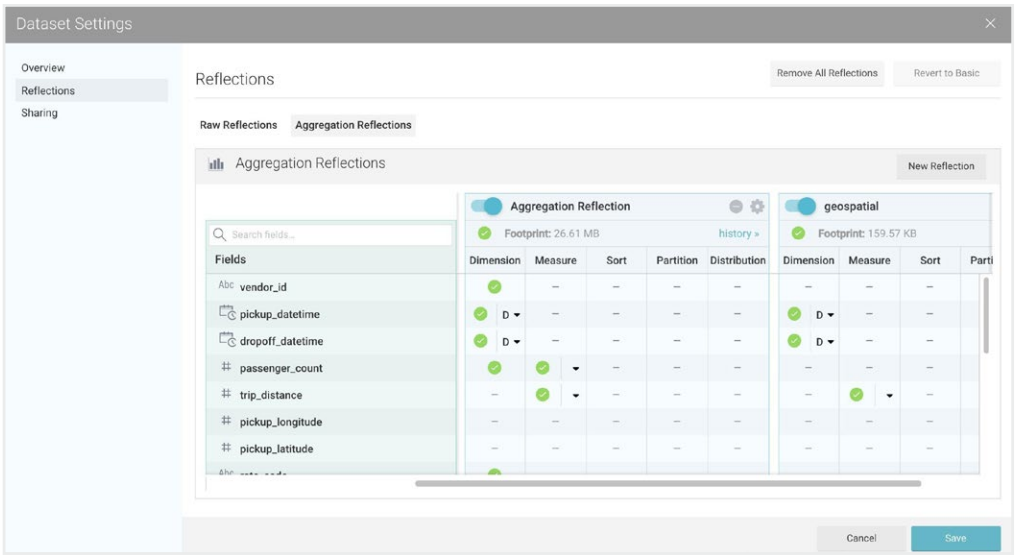
In this example, if the dataset comprises 100 million records, this aggregation reflection will be very small compared to the size of the raw data. It will include at most 12,500 rows of data, with all the measures calculated for each row (i.e., all intersections of possible dimensions). However, if the aggregation reflection includes columns that have a high degree of cardinality, then the size could be much larger:

COLUMN	NUMBER OF UNIQUE VALUES
State	42,000
Color	25
Size	10
Total rows in aggregation reflection	**10,500,000**

Note that if the columns are correlated, there will be fewer records than the multiple of the cardinalities. For example, if we assume that ZIP codes never span more than one city (which, in reality, is not entirely true), then adding City to the example above would slightly increase the number of records in the aggregation reflection but not as much as if the two fields were not correlated.

The core set of measures can be used to derive other measures at query time. For example, Dremio maintains COUNT and SUM which allows AVG to be calculated dynamically without storing this value. The full list of measures Dremio maintains are COUNT, MAX, MIN, SUM and approximate distinct count. Administrators can select one or all of these options to be maintained for a data reflection; only COUNT and SUM are enabled by default.

USE CASE	HOW AGGREGATION REFLECTIONS HELP
OLAP workloads from BI tools	Most BI tools issue queries with one or more aggregations and one or more GROUP BY expressions. Aggregation reflections precompute measures for all permutations of dimension combinations, greatly improving the efficiency of these types of queries.
Distinct values	Dremio can optimize distinct value expressions using aggregation reflections, provided the column is specified as a dimension in the data reflection.
Fast count distinct	Dremio can optionally store an approximate distinct count measure on a column. This count is determined using the HyperLogLog algorithm.



HOW DREMIO MANAGES DATE-RELATED DIMENSIONS IN AGGREGATION REFLECTIONS

There are two date-related data types in Dremio: date and timestamp. When timestamp columns are configured as dimensions in an aggregation reflection, Dremio will automatically extract the day-level date value and use this as the grouping value in the data reflection. Dremio can then automatically and efficiently roll up these day-level values to different levels of granularity in the time hierarchy such as month, year, quarter and week number at query time. This applies to date data types as well.

For example, consider a dataset that includes a timestamp column for sales records:

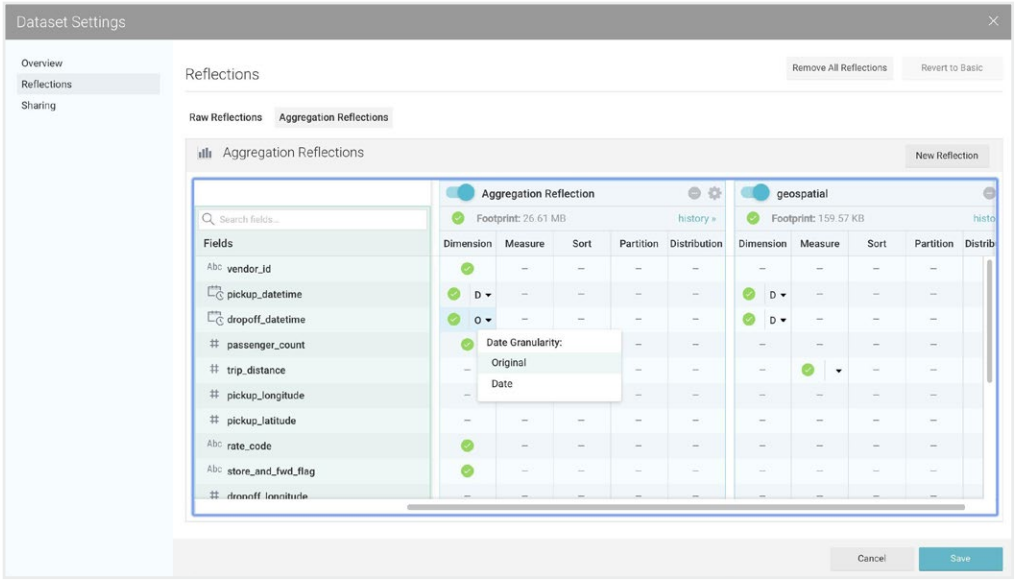
id	sale_date	items
1	2014-12-11 11:51:00.000	1
2	2014-12-12 11:59:00.000	3
3	2014-12-12 11:00:00.000	2
4	2014-12-13 10:35:00.000	6
5	2014-12-13 10:48:00.000	1

With an aggregation reflection on this dataset that includes the sale_date and item_count columns with all measure types enabled, the data reflection would include the following values:

sale_date	items_count	items_sum	items_max	items_min	items_acd
2014-12-11	1	1	1	1	1
2014-12-12	2	5	3	2	2
2014-12-13	2	7	6	1	2

Note that COUNT and SUM measure types are enabled by default, and here "acd" corresponds to the approximate count distinct measure. This table is for illustrative purposes only, the actual data in a data reflection would be organized differently and would include other pieces of metadata.

Day-level granularity is enabled by default. If you want to preserve the original granularity of the date in the data reflection, this option can be selected on the dimension column: "O" represents original and "D" represents day:

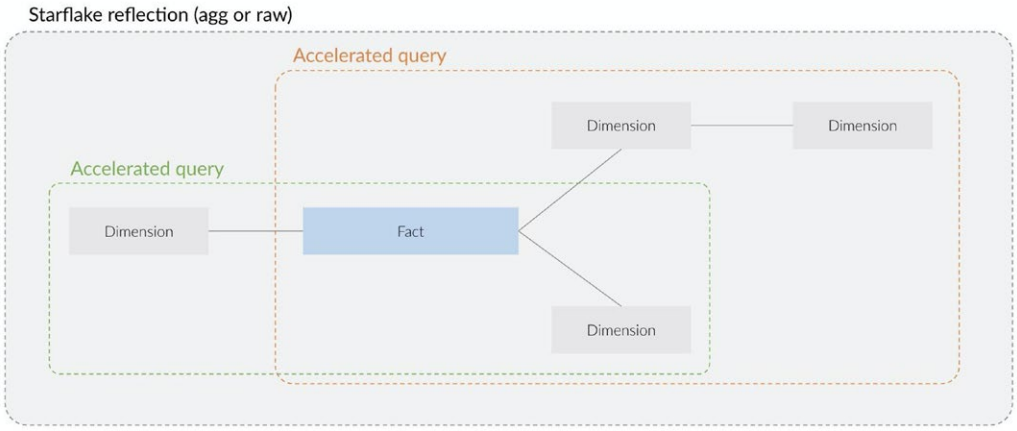


Typically, day-level granularity is the appropriate setting as this yields a reasonable cardinality for most datasets. However, there are use cases where preserving the original value is the appropriate setting, such as events distributed throughout the day where time of day patterns are being analyzed. Keep in mind that you can always adjust the precision of the time value (e.g., roll up to the nearest minute) to arrive at the ideal cardinality.

Starflake Reflections

A starflake reflection is a reflection that joins multiple physical datasets in which some or all of the joins are record-preserving joins. Any time Dremio executes a join, it records whether that join was a record-preserving join. An outer join is a record-preserving join if the number of records in the resulting dataset is equal to the number of records in one of the original datasets. An inner join, on the other hand, is a record-preserving join if every record in one of the original datasets was matched in the join, and the number of records in the resulting dataset is equal to the number of records in that same original dataset. Note that a join can be record preserving in one direction or both (or not at all).

If a reflection creation job consists of one or more record-preserving joins, the system considers the reflection to be a starflake reflection. There is no need for a user to explicitly designate fact and dimension datasets, or indicate that a join is record preserving.



Starflake reflections are used by Dremio to accelerate queries that include the dataset in the starflake reflection whose records were "preserved" by the joins (e.g., the fact dataset in a star/snowflake/starflake schema). Dremio's optimizer can determine if the starflake reflection can be substituted in such a query plan, and can use this reflection to accelerate the query. This is a unique and powerful capability as there are potentially very large numbers of variations of queries that can be accelerated from a single starflake reflection.

For an in-depth discussion, see Introduction to **Starflake Data Reflections.**

Data Reflection Store

Dremio maintains data reflections in a file system or object store. Users can configure their deployments to make use of network storage or distributed stores such as HDFS. Users can further improve performance by using low-latency storage such as SSDs. In addition, users can take advantage of cost-effective object stores like AWS S3 and Azure Data Lake Store (ADLS) as their reflection store, providing separation of compute and storage, as well as unlimited scalability.

When using object stores such as S3 and ADLS for storing data reflections, consider that the read latency provided by these stores is significantly greater than that of file systems, and Dremio's query acceleration performance may be impacted. Additionally, users don't have to worry about using network storage that is not attached because Dremio will use local storage in conjunction with *columnar cloud cache (C3).*

Data reflections are maintained in a high-performance columnar representation based on Apache Parquet and Apache Arrow, utilizing advanced compression techniques such as dictionary encoding, run-length encoding and delta encoding.

One or more executor nodes in a Dremio cluster will read from the reflection store in parallel. Dremio uses a proprietary, vectorized Parquet reader to read data from the data reflections directly into Apache Arrow memory buffers. Dremio carefully reads subsets of the Parquet file into memory as appropriate, rather than full files, to make optimal use of scarce memory resources.

HOW MUCH SPACE DO DATA REFLECTIONS CONSUME

In most Dremio environments, data reflections are very small compared to the raw data, typically less than 10%, and in many cases less than 1% the size of the original.

All data reflections are stored in a highly compressed columnar format, which is often much more space-efficient than the raw data. For example, JSON data is frequently less than 5% of the original size when stored in a data reflection due to the efficiency of the compression schemes used in the columnar format.

Raw reflections have the same number of records as their anchor datasets, but they are normally a small fraction of the size of the anchor dataset. In many cases only a small subset of the dataset columns are queried by users, and therefore, it makes sense to include only those columns in the raw reflection.

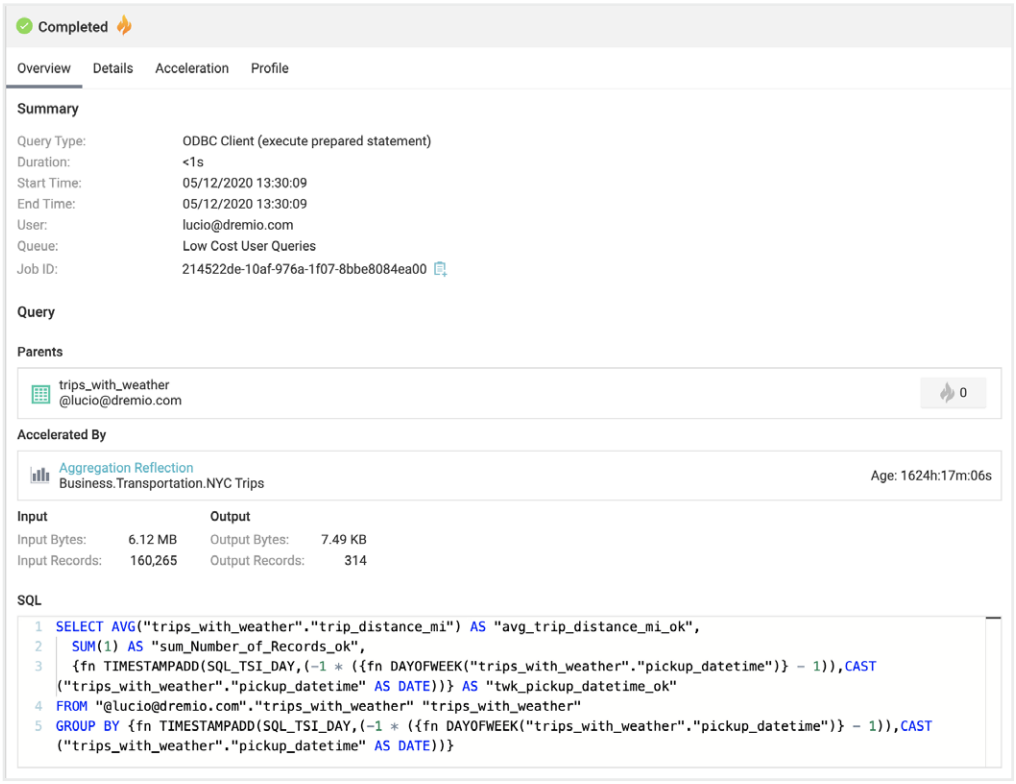
Aggregation reflections are summarizations of anchor datasets, and therefore have fewer records. The total number of records in the aggregation reflection can be calculated as the product of the number of unique values in each of the dimension columns. When the number of unique values in the dimension columns is low, the aggregation will be relatively small, and when there are many unique values it will be larger. For an example, see the section on *aggregation reflections* above.

While it is possible to define an aggregation reflection that has the same number of records as its anchor dataset (by selecting a dimension with the same cardinality as the dataset), this would defeat the purpose of the aggregation reflection and would be equivalent to using a raw reflection on that dataset.

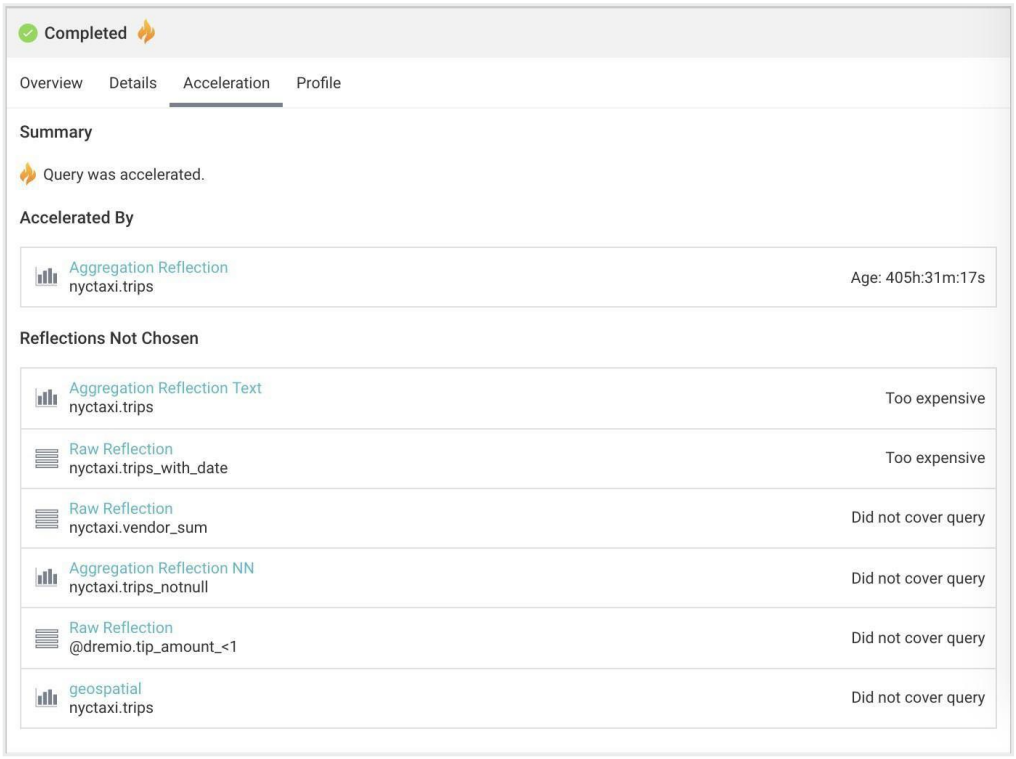
Dremio displays the size of each data reflection in several of the administrative screens, and are listed as "footprint."

How to Know if a Query Was Accelerated

Every query in Dremio is processed as a job, and the details for how the job was executed are maintained in Dremio's job history. If the query was accelerated, Dremio will display a small flame on the job detail page:



In addition, administrators can see details about the data reflections that were considered and used for executing the query on the Acceleration tab of the job:



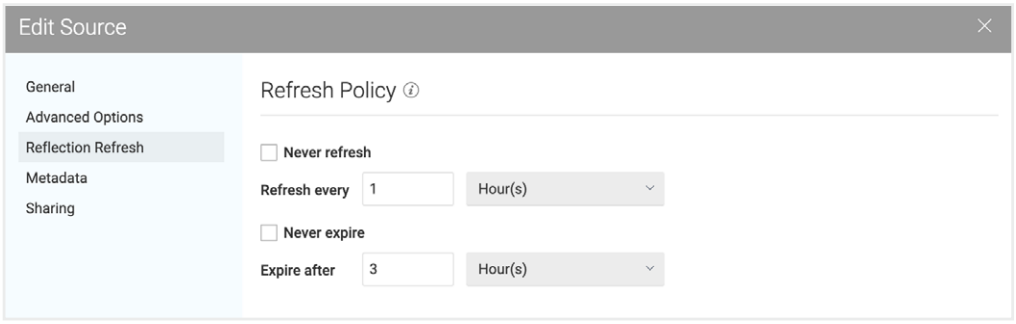
In this example, a number of data reflections were considered. Four were excluded because they did not cover the query, meaning they were missing columns or rows needed to satisfy the query (e.g., tip amount < 1). Two were not chosen because they would have required a query plan that was more expensive than the plan that was used for the data reflection that was chosen.

Data Reflection Maintenance

As physical datasets change, data reflections should be periodically refreshed. At a high level, there are two modes for managing the data reflection refresh process:

- Dremio manages the refresh following a schedule
- An external process notifies Dremio when it is time to refresh

In the first mode, an administrator specifies the desired refresh policy for an entire data source, or specifically for each dataset. These intervals are typically some number of weeks, days, hours or even minutes. Dremio effectively learns how long the refresh takes and will start the process in advance to maintain the SLA specified on the data source or dataset.



There are separate options in the refresh policy for specifying the refresh interval as well as the expiration of existing data reflections. All data reflections based on a physical dataset or source will be refreshed accordingly. Refresh policy options for a physical dataset will override the value for the data source.

Dremio will refresh data reflections at the provided refresh interval and serve them until the provided expiration.

Data reflections can be updated in one of two modes:

- **Full refresh.** The entire data reflection will be rebuilt.
- **Incremental refresh.** The data reflection will be updated based on new data since the last refresh job.

Incremental refreshes are only possible on certain data sources such as append-only file systems. For more information on incremental refresh, see the documentation on *Refreshing Data Reflections.* Note that for any downstream reflections that join an incrementally refreshed data reflection, the downstream data reflection will need to be fully refreshed, since the join may have removed rows that it now needs for the new data.

The work to build and maintain a data reflection can be significant. Dremio automatically associates these maintenance tasks with a workload queue that limits concurrency and has a lower priority than user queries. Administrators can adjust these settings as appropriate. For more information, see the documentation on *Managing Job Queues.*

For the second mode, an external process notifies Dremio that it is time to refresh the data reflection. This option is useful if you have a data pipeline or ETL process that prepares data, and as a final step, can notify Dremio via the REST API that it is time to refresh the data reflections. To use this approach, configure the dataset or the data source to "Never Refresh" and then call the REST API when appropriate. You can read more about *using the REST API to update data reflections* in the documentation.

Administrators can use a combination of both scheduled and triggered data reflection refresh strategies. For example, if the data is usually updated every day, but occasionally, based on an event, the data was updated off schedule. The script that writes the data to the destination can make a REST API call to Dremio to trigger a refresh on demand, and the scheduled refreshes will continue according to the refresh policy.

Optimizing Data Reflection Refreshes

Internally, Dremio maintains a dependency graph (DAG) that defines the order in which data reflections are refreshed. The dependencies are calculated based on relational algebra, and the actual refresh start time takes into account the expected amount of time required to complete the entire refresh cycle.

Note that this graph-based approach reduces the end-to-end cycle time, as well as the compute resources required to complete the cycle. In addition, by leveraging one data reflection to refresh another data reflection, the system can avoid resource-intensive reads on operational databases more than once.

External Reflections

Dremio supports the notion of external reflections. Users can create and maintain data reflections using an external process such as Apache Hive or Apache Spark, then register the external reflection in Dremio. Dremio will consider these data reflections in its cost-based analysis to accelerate queries. Any data source that Dremio supports can be used for external reflections, including Amazon S3, Microsoft ADLS, relational databases and Hadoop.

External reflections are useful in cases where it makes sense to manage the creation outside of the Dremio process. For example, for processing that runs for many hours or days, or when existing processes are already in place that create optimized representations of data for specific query patterns. The advantage of registering these resources in Dremio is that it simplifies the experience for data consumers, and provides additional capabilities for securing and governing access to the data, as well as tracking data lineage.

Best Practices

The following sections represent best practices for administering and operating a Dremio cluster based on experiences with the Dremio community and customer deployments. We update this document periodically to incorporate lessons learned based on the newest updates to Dremio, so be sure to read the latest version.

THINK IN TERMS OF SEVERAL DISCRETE DATA REFLECTIONS

Data reflections allow administrators to be iterative in their approach to performance optimization. Because data reflections do not require any change in the behavior of the data consumer, administrators can add and refine data reflections on an ongoing basis with little to no impact to ongoing workloads. Generally speaking, administrators should think about their data reflection strategy as employing on the order of tens of data reflections, each configured for different query patterns, rather than a single uber-data reflection that works for all queries.

HOW TO DESIGN DATA REFLECTIONS

Administrators should perform a clustering exercise on known workloads to determine the optimal set of data reflections for a given deployment. Administrators should isolate known query patterns into groups that do not interact with one another. The more discrete the query groups, the smaller the reflections will be on disk, the more efficient data reflection maintenance will be, and the faster queries will be executed.

Keep in mind that a single query can use multiple data reflections and a single data reflection can serve many queries. Administrators should optimize for the overall workloads based on requirements for cost, storage, response time and data maturity.

HOW TO ACCELERATE A QUERY PATTERN

Dremio supports two types of data reflections: raw reflections and aggregation reflections. If a known query pattern returns row-level information, raw reflections are appropriate. If the query returns summarized data based on GROUP BY expressions or aggregations (e.g., SUM, AVG, COUNT, MIN, MAX), then an aggregation reflection is appropriate.

When determining how to accelerate workloads, administrators should consider information provided in the query profile to understand which stages of the query plans are most expensive to perform. For example, it may be the case that joins are the dominant factor in a query plan, and that while aggregations consume some of the resources, they are relatively minor compared to the joins. In such a case, a raw reflection may be more appropriate than an aggregation reflection, as it can be used to accelerate a broader range of queries.

Certain types of patterns can be used to group queries together. The following sections describe some examples.

JOINS

Joins between datasets tend to be expensive operations for known query patterns. By using data reflections, the costs of performing a join can be amortized across many queries. Administrators can identify a group of queries that make use of a common join pattern. Then, administrators can generalize the join to be beneficial for all queries in the query pattern by removing any additional predicates from the queries to express a common join. The resulting query can serve as the basis of the data reflection, for both raw and aggregation reflections.

For example, consider the following three queries which use a common join pattern on datasets A, B and C:

Query 1

```
SELECT a1, b1, c1 FROM a,b,c WHERE a.3 > ‘2001-01-01’ AND
b.3 IN
(‘red’,’blue’,green’)
```

Query 2

```
SELECT a1, a2, c1, COUNT(b.1) FROM a,b,c WHERE a.size =
‘M’ AND b.2 < 10 AND c.2 > 2
GROUP BY a1, a2, c1
```

Query 3

```
SELECT a1, b2 FROM a,b,c WHERE c.1 = 123
```

A Dremio administrator can create a raw reflection that accelerates all three queries using the following configuration:

```
SELECT a1, a2, a3, b1, b2, b3, c1, c2 FROM a,b,c WHERE a.3
> ‘2001-01-01’ AND b.3 IN
(‘red’,’blue’,green’) AND c.1 = 123
```

This data reflection would accelerate these three queries without changing any of the three original queries—Dremio would rewrite each to use the data reflection instead of scanning the raw data. This specific configuration is valid and may be optimal; however, it would be limited in its ability to accelerate more generic queries. The administrator could create a more generic data reflection that could accelerate a broader range of queries with some additional maintenance cost by using the following expression:

```
SELECT a1, a2, a3, b1, b2, b3, c1, c2 FROM a,b,c WHERE a.3
> ‘2001-01-01’
```

Optionally, an even more generic form could be created that omits the date constraint. Each version of the query trades the ability to support more queries with additional maintenance cost, and administrators can make decisions that are optimal based on their resources and the SLA of the deployment.

FILTERS

If filters are common in your query patterns, pre-filtering the data can provide a significant benefit to query performance. However, it may be the case that these filters do not apply to all query patterns, and so it does not make sense to build them into your joined virtual datasets. Instead, you can build supporting anchor datasets whose sole purpose is to act as anchor datasets for data reflections. For more information on this concept, see the section on *supporting anchor datasets* below.

For example, consider a schema with sales data that includes a lookup table to calculate tax based on the region of the sale. The administrator can create supporting anchor datasets that filter the data for specific regions, such as North America, EMEA or APAC. Each of these virtual datasets would be a simple SELECT * on the foundational virtual dataset used to model the sales data for a company, a join to the lookup table for tax calculation, with a filter on the specific region.

With these three supporting anchor datasets in place, administrators would then create raw or aggregation reflections on each. Data consumers would continue to query the foundational data model, but now their queries would automatically be accelerated to use the appropriate data reflection for their respective regions, showing them sales with the appropriate tax applied.

If there is a difference in filters, the data reflection should include the broadest (i.e., most coarse) filter or potentially no filter. For example, if query 1 filters A > 8 and query 2 filters A > 4, the data reflection should have A > 4. However, if there's a query 3 in the set which doesn't filter on A, the data reflection should not have a filter on A, since the data reflection would not cover query 3.

AGGREGATIONS

Dremio can pre-aggregate data at multiple levels of granularity. Then, at query time, Dremio can determine how to further aggregate the data as appropriate. Administrators can create aggregation reflections that include the lowest level granularity as well as the most coarse granularity, and Dremio will automatically aggregate at the appropriate level at query time.

For example, consider a history of sales data with data about each customer, including ZIP, city, state and country on each record. The administrator can create an aggregation reflection with several measures and specify ZIP, city,

state and country as dimensions. Dremio will maintain aggregations for the measures down to the ZIP level in the aggregation reflection. Dremio will also automatically summarize to ZIP, city, state or country using the same aggregation reflection at query time.

Administrators should always include all levels of granularity that data consumers may wish to group by, and they should avoid any level that is not used, as this will make the aggregation reflection unnecessarily large. See the section on *aggregation reflections* for an example of how the cardinality of dimensions affects the overall size.

In addition to creating an aggregation reflection directly on the dataset, administrators can also use supporting anchor datasets to accomplish the same goal. For example, the administrator could create a supporting anchor dataset that includes the aggregations and GROUP BY statement for the dimensions, then create a raw reflection on this data. The end result would be the same.

CALCULATED FIELDS

For calculated fields that are frequently used by data consumers, administrators have a few different options for accelerating these calculations:

- **Add the calculated field to a virtual dataset.** The administrator can add a new column that provides the calculation. Depending on the expression, Dremio may be able to match the new column without making the data consumers explicitly use the new column. Otherwise, they will need to include the new column in their queries.
- **Use a supporting anchor dataset.** The administrator can create a supporting anchor dataset that includes the calculated field along with other fields from the dataset, and Dremio will automatically use the associated data reflection to accelerate the query.

For more information on how to use *supporting anchor datasets*, see the section below.

WHEN TO USE RAW REFLECTIONS

Raw reflections allow administrators to address a number of performance concerns. Here are some common examples of when to consider raw reflections:

- **Non-columnar datasets.** If the source data is stored in a non-columnar format (e.g., JSON, CSV), using a raw reflection can dramatically improve the performance of queries.
- **Offloading operational sources.** If the data source is deployed for operational workloads, there's a good chance the system is not optimized for scan-intensive workloads. Using a raw reflection will allow Dremio to execute most analytical queries without touching the data source.

- **Needle-in-a-haystack query pattern.** If queries return a subset of the data based on predicates, in non-aggregated form, then raw reflections are the best way to optimize these queries.
- **Complex joins.** Joins are expensive. Using a raw reflection to pre-join data from one or more sources can significantly improve performance.
- **Expensive transformations.** Rather than calculating expensive transformations at query time, a raw reflection can store the results of the transformation so they are effectively free for subsequent queries.
- **Wide datasets.** If your datasets have many columns (e.g., more than 100), raw reflections can be used to vertically partition the dataset into subsets that are commonly accessed together.

WHEN TO USE AGGREGATION REFLECTIONS

Aggregation reflections manage summarized representations of the data. Most BI tools generate aggregation and GROUP BY queries. Aggregation reflections optimize these kinds of query patterns.

For aggregation reflections, keep in mind that:

- Dimensions should have relatively low cardinality, and high cardinality columns will yield less benefit.
- Create multiple aggregation reflections with a subset of dimensions, rather than one uber-aggregation reflection. Caution: If a query uses more dimensions than in an aggregation reflection, it will not cover the query and cannot be used for acceleration.
- Multiple small aggregation reflections (versus one large one) are good for isolated pockets of query patterns on the same dataset that do not overlap. If there is a lot of overlap, fewer larger aggregation reflections will work better.
- For computed measures (e.g., revenue=qty*item_price), first create a VDS with revenue column and create the aggregation reflection on the VDS.

WHEN TO USE EXTERNAL DATA REFLECTIONS

External reflections provide benefits that are similar to data reflections managed by Dremio. In contrast, external reflections are managed by a process external to Dremio, such as Apache Hive or Apache Spark, and work with any data source supported by Dremio. The same kinds of cost optimization and query rewriting will be performed, whether the data reflection is managed by Dremio or is actually an external reflection.

Dremio's SQL engine is optimistic in its execution and expects most queries can be completed in a few seconds or less. To optimize performance Dremio does

not perform checkpointing or other techniques that are appropriate trade-offs for long-running processes.

When Dremio is creating a data reflection, it could fail 99% of the way to completion, then need to restart the process from the beginning, whereas with Hive the process of creating the transformed dataset is less efficient but more resilient in the event of failures. If the process of creating and updating data reflections runs for many hours, it may make sense to use an external reflection.

The advantage of using Dremio with datasets that are being managed outside of Dremio is that it simplifies the experience of data consumers who work with one logical data model, independent of the number of physical datasets that are being created to optimize different workloads.

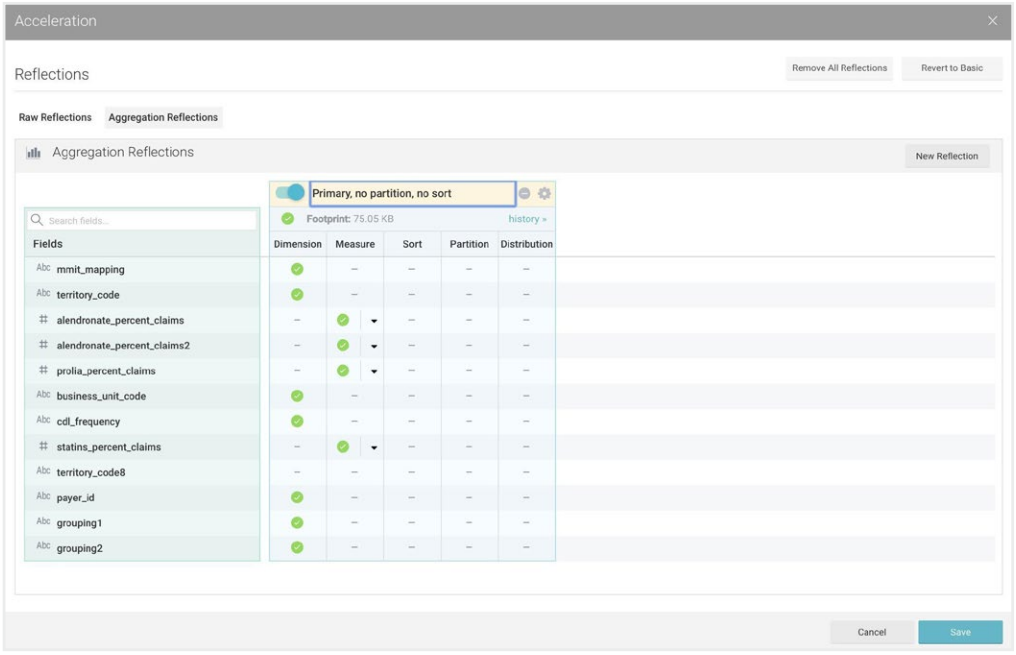
Keep in mind the following:

- When using external reflections, Dremio assumes the data is sufficiently fresh, and makes no checks to determine otherwise.
- Finally, external reflections will not appear in Dremio's management screens. However, they do appear in the system tables, for example, when querying SELECT * FROM sys.reflections.

NAMING DATA REFLECTIONS

Dremio uses the names raw reflection and aggregation reflection by default when creating a new data reflection. If there is more than one of each type of data reflection, Dremio will append an incrementing number (e.g., raw reflection (1)) to maintain a unique naming scheme.

Administrators can specify a name through the advanced modal:



The purpose of this name is to help the administrator understand which data reflections were used or considered in query planning, so it can make sense to use a naming scheme that helps in this context. Note that if you rename a data reflection, Dremio will rebuild the data reflection.

WHICH FIELDS TO USE FOR THE PARTITION OPTION

When administrators select a field for partitioning in a data reflection, Dremio will physically group records together into a common directory on the file system. For example, when partitioning by COUNTRY where the values of this column are US, UK, DE, CA, etc., Dremio will store data together into directories called US, UK, DE, CA, etc. This optimization allows Dremio to scan a subset of the directories based on the query in an optimization called partition pruning. If a user queries on records that have COUNTRY IN ("US," "UK") then Dremio can apply partition pruning to scan only the US and UK directories, significantly reducing the total data that is scanned for the query.

When using a partitioning column for data reflections, administrators should consider:

1. Is the field used in many queries?
2. Are there relatively few unique values in the field (low cardinality)?

To partition the data, Dremio must first sort all records, which will consume resources. Accordingly, data should only be partitioned on fields that can be used to optimize queries. In addition, the number of unique values for a field should be relatively small (e.g., COUNTRY) to provide a relatively small number of partitions. If all values in a field are unique, in contrast, the work to partition will be high cost for relatively small benefit. In general, we recommend the total number of partitions for a dataset be less than 10,000 unique values.

WHICH FIELDS TO USE FOR THE SORT OPTION

The sort option is useful for optimizing filters and range queries, especially on columns with high cardinality. Dremio can take advantage of the sorted data to more efficiently apply filters and range predicates to queries. If sorting is enabled, during query execution, Dremio skips over large blocks of records based on filters on sorted columns.

Typically, it is not beneficial to sort on more than one column in a single data reflection, as this does not improve read performance significantly and will increase the costs of maintenance tasks.

For workloads that need to support multiple sort options, consider multiple data reflections where each is sorted on a single column.

At this time Dremio does not use the sort option to optimize sort operations. While the data is sorted locally within a node (and partition, if applicable), most data reflections span multiple nodes and partitions, and there is no global sort applied across all files. Ultimately, the data must still be sorted at query time.

WHICH FIELDS TO USE FOR THE DISTRIBUTION OPTION

The distribution option allows data from multiple datasets to be co-located and co-partitioned across nodes in a cluster in order to minimize data movement during join operations. Currently, Dremio does not optimize queries using the distribution option.

MANAGING DATA REFLECTIONS WITH SUPPORTING ANCHOR DATASETS

Administrators will usually begin by creating data reflections on the virtual datasets accessed by data consumers. As administrators develop a better understanding of the query patterns that need to be accelerated, they may want to manage their data reflections with generalized virtual datasets that are not intended to be accessed by data consumers directly. We call these *anchor datasets.*

For example, consider a schema that includes three popular tables used in many queries:

Customer	Order	Lineitem

The administrator has determined that there are a few other common patterns in user queries:

- These three tables are frequently joined together
- Queries always filter by commit_date < ship_date
- There is a calculated column in most queries: extended_price * (1-discount) AS revenue

The administrator can create a virtual dataset that applies these common patterns on these three tables, and then create a raw reflection to accelerate queries. By using a supporting anchor dataset, the administrator can accelerate these user queries without having them change their existing queries.

To proceed, the administrator follows these guidelines:

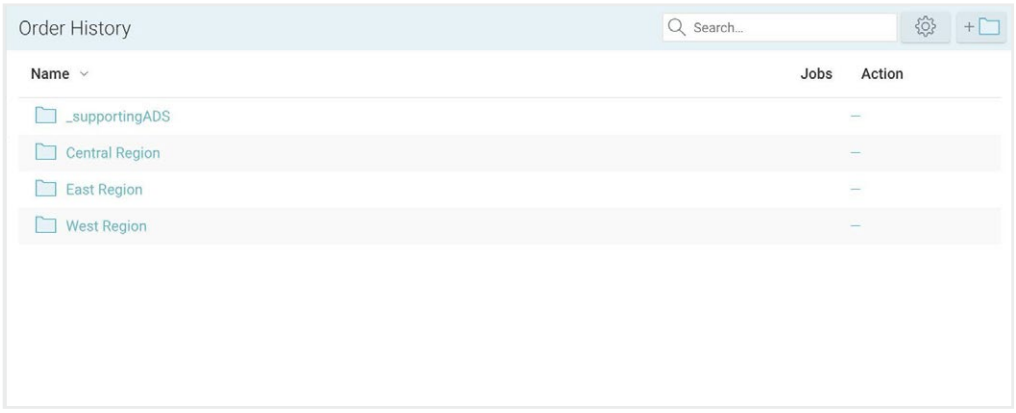
- Use SELECT * to include all columns from the three tables to accelerate the broadest set of queries. Alternatively, if the administrator knows exactly which subset of columns will be used, only the subset can be included to increase efficiency.
- Add any calculated columns, in this case the revenue column.
- Apply the appropriate joins on the three tables.
- Apply any filters that are used by all queries, in this case commit_date < ship_date. Always use the most generic predicate possible to maximize the number of queries that will match.

The administrator uses the following query to create a new virtual dataset:

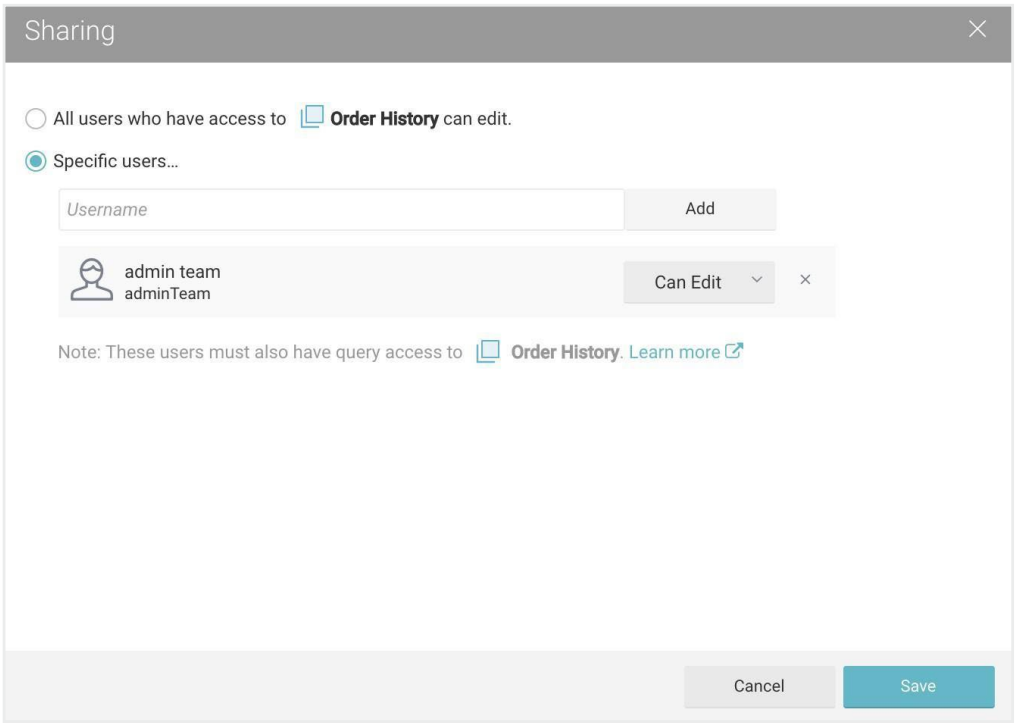
```
SELECT *,
extendedprice * (1 - discount) AS revenue FROM customer AS
c, orders AS o, lineitem AS l WHERE
c.c_custkey = o.o_custkey AND
l.l_orderkey = o.o_orderkey AND
o.commit_date < o.ship_date
```

The administrator then creates a raw reflection on this new virtual dataset. Options for sorting and partitioning apply in this case just as they do for any other raw reflection. Similarly, if most queries are aggregation queries then an aggregation reflection can be used instead.

To effectively hide supporting anchor datasets from users, the administrator can organize them in a folder in the space used by these data consumers called _supportingADS:



Then the administrator can configure this folder to be invisible and inaccessible to the data consumers:



Even though the data consumers do not have access to the datasets stored in _supportingADS, Dremio will be able to accelerate their queries as long as they have access to the physical datasets that their queries ultimately expand to include.

DATA REFLECTION MAINTENANCE DURING DESIGN

When developing a data reflection it is common to make changes to the configuration or the underlying virtual dataset. While iteratively working through these configurations, you may want to create the data reflection, then edit the virtual dataset definition, then re-create the reflection, since edits to the virtual dataset do not trigger a data reflection refresh. In this case, you'll want to disable the reflection via the toggle, click save, then enable the data reflection via the same toggle.

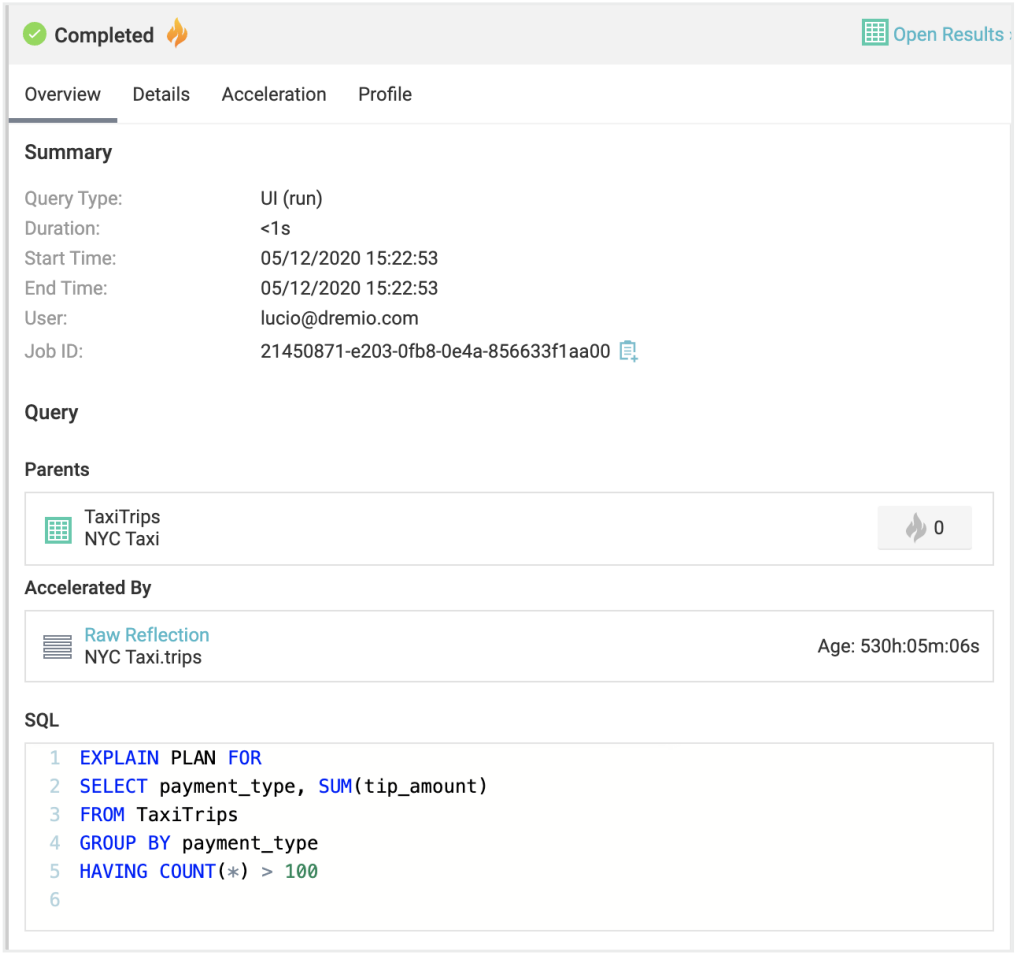
TESTING DATA REFLECTION MATCHING USING EXPLAIN PLAN

Administrators can test which data reflections will be used for a query without actually running the query. This can be very helpful when the physical datasets are very large and administrators want to avoid processing large queries unnecessarily.

To understand how Dremio will execute the query, administrators can use the EXPLAIN PLAN option by simply prepending this statement to the query they wish to test. For example:

```
EXPLAIN PLAN FOR
SELECT payment_type, SUM(tip_amount)
FROM TaxiTrips
GROUP BY payment_type
HAVING COUNT(*) > 100
```

Then navigate to the jobs tab to view the query profile:



What to Avoid

Data reflections are very useful when they are created following the best practices listed above. Here is a list of things to keep in mind when working with data reflections to ensure the best user experience:

- Avoid the "more is better" approach, creating multiple reflections without a proper strategy in place can be difficult to manage.
- Only incorporate reflections when users are experiencing slow query responses or reports are not meeting established SLAs.
- Frequently check for reflections that are no longer being used to accelerate queries and evaluate whether they should be kept or removed.

Accessing Data Reflections Without Querying Dremio

Data reflections are designed to be used by Dremio's query optimizer, and are not designed for direct access by external applications. However, the files generated by Dremio are standard Parquet files and assuming appropriate permissions are available to an application, there is no reason why they could not be accessed.

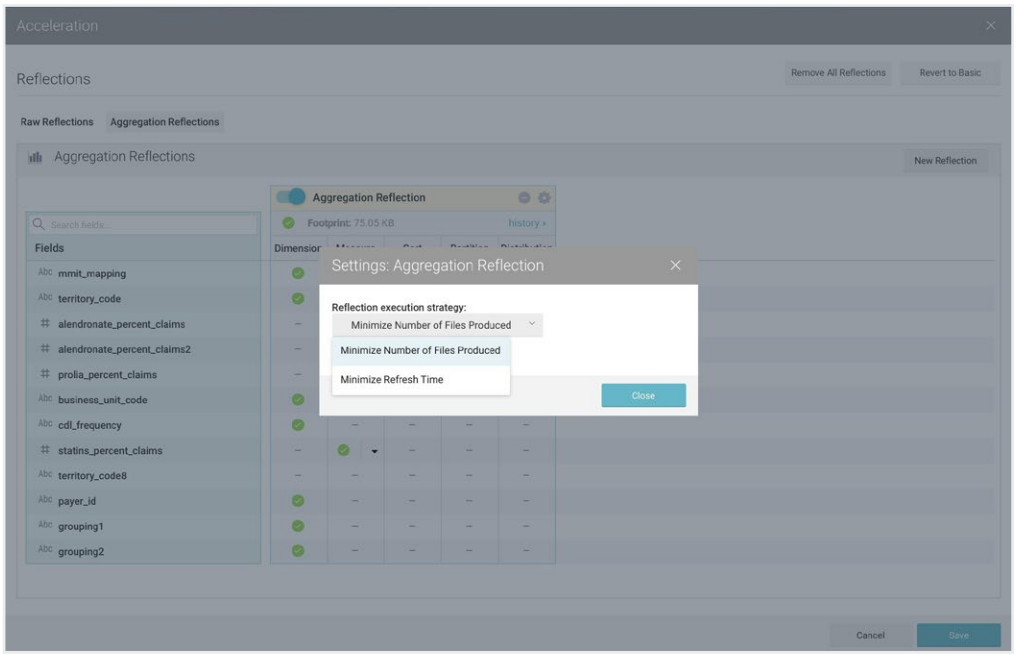
Keep in mind that Dremio makes no guarantees as to the names of these files, and as data reflections are updated the files will be replaced. Furthermore, Dremio may be configured to expire data reflections on a schedule that has a different periodicity than the refreshes, and so it will be important to query Dremio's system tables to know exactly which Parquet files correspond to the data reflection for a given dataset.

Rather than query the data reflection Parquet files, we recommend that applications query Dremio directly. Alternately, users can materialize data into a file system or object store such as HDFS, S3, ADLS, etc. using the high-performance parallel export feature (CTAS). The format used in these exports is very similar to that of data reflections.

Optimizing Data Reflection Persistence

Dremio makes use of multiple compression options for storing data reflections, including dictionary encoding, run-length encoding and delta encoding. For persisting reflections, you can choose to minimize the number of files or minimize time needed to refresh a data reflection.

This option can be accessed from the advanced modal for a data reflection, then clicking on the gear icon:



Select **minimize time needed to refresh** if each thread will have a good amount of data and there aren't many many threads and/or you need the data reflection to be created as fast as possible. With this setting, each final stage thread will open their own writers to write their data. This can result in many small files if each thread has a small amount of data. Select **minimize number of files** if each final stage thread may have a small amount of data, relative to the block size of the underlying data store. With this option, the threads will pool the data and ensure that the fewest number of files are written to the reflection store. Optimizing for a smaller number of files generally improves read performance as there tend to be fewer seeks performed for a given query.

Configuring Metadata Refresh and Data Reflection Refresh Optimally

Refresh intervals should be equal to or higher (i.e., more time) than the metadata refresh interval for those objects. This ensures that data reflection refreshes always take into consideration the most recent data in the refresh process.

Updating Data Reflections When Dataset Schemas Change

When the definition of a dataset changes, this does not automatically update any associated data reflections. The data reflection will be updated during the next scheduled or triggered refresh. If new columns are added, or column names are changed, they will not automatically be included in the next refresh. Administrators should update their data reflections to incorporate any schema changes to physical or virtual datasets.

Managing Data Reflections With SQL and the REST API

Dremio provides SQL functions for managing data reflections. For more information, see *Creating Data Reflections* in the documentation. Dremio also provides REST endpoints for creating and managing data reflections. For more information, see the *Reflection API* in the documentation.

The following methods are used to refresh data reflections and are typically implemented at the end of an ETL job

REFRESHING A REFLECTION USING A SQL STATEMENT

ALTER PDS <PHYSICAL-DATASET-PATH> REFRESH METADATA

For in-depth detail please see *Managing Physical Datasets*

REFRESHING A REFLECTION USING API

POST /api/v3/sql

For in-depth detail please see Dremio's *API doc.*

REFRESHING A REFLECTION USING THE DREMIO PYTHON CLIENT

Users can execute either one of the following functions:

1. refresh_vds_reflection_by_path
2. refresh_reflections_of_one_dataset

For in-depth detail please see the *Dremio Client documentation.*

ABOUT DREMIO CORPORATION

Dremio's Data Lake Engine delivers fast query speed and a self-service semantic layer operating directly against data lake storage. Dremio eliminates the need to copy and move data to proprietary data warehouses or create cubes, aggregation tables and BI extracts, providing flexibility and control for Data Architects, and self-service for Data Consumers. For more information, visit www.dremio.com.

Founded in 2015, Dremio is headquartered in Santa Clara, CA. Investors include Cisco Investments, Lightspeed Venture Partners, Norwest Venture Partners and Redpoint Ventures. Connect with Dremio on GitHub, LinkedIn, Twitter, and Facebook.