



ThoughtSpot Data Integration Guide

Release 8.0

December, 2021

© COPYRIGHT 2015, 2020 THOUGHTSPOT, INC. ALL RIGHTS RESERVED.

910 Hermosa Court, Sunnyvale, California 94085

This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent in writing from ThoughtSpot, Inc.

All rights reserved. The ThoughtSpot products and related documentation are protected by U.S. and international copyright and intellectual property laws. ThoughtSpot and the ThoughtSpot logo are trademarks of ThoughtSpot, Inc. in the United States and certain other jurisdictions. ThoughtSpot, Inc. also uses numerous other registered and unregistered trademarks to identify its goods and services worldwide. All other marks used herein are the trademarks of their respective owners, and ThoughtSpot, Inc. claims no ownership in such marks.

Every effort was made to ensure the accuracy of this document. However, ThoughtSpot, Inc., makes no warranties with respect to this document and disclaims any implied warranties of merchantability and fitness for a particular purpose. ThoughtSpot, Inc. shall not be liable for any error or for incidental or consequential damages in connection with the furnishing, performance, or use of this document or examples herein. The information in this document is subject to change without notice.

Table of Contents

Introduction2

Reaggregation in practice3

Liveboard schedule gating conditions in practice13

Add a hyperlink to a Search18

ThoughtSpot in Practice

Summary: This guide demonstrates the power of ThoughtSpot to solve real solutions we developed for our clients.

Note: ThoughtSpot has renamed Pinboards to Liveboards. The functionality remains the same.

The purpose of this section is to guide you through a few solutions we created for our clients, so you can leverage our experience to quickly and confidently employ ThoughtSpot in meeting your own business objectives.

Each topic and scenario includes a real-world data modeling problem, and how we solved it with ThoughtSpot technology.

- [Scenario 1: Supplier tendering by job \[See page 3\]](#)
- [Scenario 2: Average rates of exchange \[See page 6\]](#)
- [Scenario 3: Average period value for semi-additive numbers I \[See page 8\]](#)
- [Scenario 4: Average period value for semi-additive numbers II \[See page 10\]](#)
- [Liveboard schedule gating conditions in practice \[See page 13\]](#)
- [Add a hyperlink to a Search \[See page 18\]](#)

Reaggregation scenarios in practice

Summary: We provide real world scenarios for using flexible aggregation in ThoughtSpot.

The following scenarios showcase the use of the `group_aggregate` function in the real world. We provide them to demonstrate to you how the function works, and the scenarios where it proved useful.

- [Scenario 1: Supplier tendering by job \[See page 3\]](#)
- [Scenario 2: Average rates of exchange \[See page 6\]](#)
- [Scenario 3: Average period value for semi-additive numbers I \[See page 8\]](#)
- [Scenario 4: Average period value for semi-additive numbers II \[See page 10\]](#)

Best practices for flexible aggregations

The `group_aggregate` function enables you to calculate a result at a specific aggregation level, and then returns it at a different aggregation level. For this reaggregation result to return correctly, follow these syntax guidelines:

- Wrap `group_aggregate` in an aggregate function, such as `sum` or `average`
- The wrapping function must be the immediate preceding function, such as `sum(group_aggregate(...))`
- Do not use with conditional operators. For example, the following expression does not reaggregate the data because the `if` precedes `group_aggregate` :

```
(if(group_aggregate(...)))
```

Scenario 1: Supplier tendering by job

We have a fact table at a job or supplier tender response aggregation level. There are many rows for each job, where each row is a single row from a supplier. A competitive tender is a situation when multiple suppliers bid on the same job.

Our objective is to determine what percentage of jobs had more than 1 supplier response. We want to see high numbers, which indicate that many suppliers bid on the job, so we can select the best response.

Valid solution

A valid query that meets our objective may look something like this:

```
sum(group_aggregate(if(sum(# trades tendered ) > 1) then 1 else 0,
                    query_groups() + {claimid, packageid},
                    query_filters()))
```

Yearly DateLogged - Fiscal	# Competitive Tendering
FY 2011	6,893
FY 2017	15,614
FY 2016	13,191

Resolution

1. The `sum (# trades tendered)` function aggregates to these attributes:

- `{claimid, packageid}`

The job-level identifier

- `query_groups()`

Adds any additional columns in the search to this aggregation. Here, this is the `dateLogged` column at the yearly level.

- `query_filters ()`

Applies any filters entered in the search. Here, there are no filters.

2. For each row in this virtual table, the conditional `if() then else` function applies. So, if the sum of tendered responses is greater than 1, then the result returns 1, or else it returns 0.
3. The outer function, `sum()`, reagggregates the final output as a single row for each `datelogged` yearly value.
 - This reaggregation is possible because the conditional statement is inside the `group_aggregate` function.
 - Rather than return a row for each `{claimid,packageid}`, the function returns a single row for `datelogged yearly`.
 - The default aggregation setting does not reaggregate the result set.

Non-Aggregated Result

We include the following result to provide contrast to an example where ThoughtSpot does not reaggregate the result set. Reaggregation requires the aggregate function, `sum`, to precede the `group_aggregate` function.

In the following scenario, the next statement is the conditional `if` clause. Because of this, the overall expression does not reaggregate. The returned result is a row for each `{claimid,packageid}`.

```
sum(if(group_aggregate (sum (# trades tendered),
                        query_groups() + {claimid, packag
eid},
                        query_filters ( ) )>1) then 1 els
e 0)
```

Search: `datelogged yearly` `# competitive tendering (invalid)`

Competitive Tendering (invalid) by Yearly DateLogged - Fiscal

Yearly DateLogged ~...	# Competitive Tendering...
FY 2011	1
FY 2011	1
FY 2011	1

Scenario 2: Average rates of exchange

The Average rate of exchange calculates for the selected period. These average rates provide a mechanism to hedge the value of loans against price fluctuations in the selected period. We apply the average rate *after the aggregation*.

The pseudo-logic that governs the value of loans is `sum(loans) * average(rate)`.

The data model has two tables: a primary fact table, and a dimension table for `rates`.

- The `loans` column is from the primary fact table.
- The `rate` column is from the `rates` table.

These tables are at different levels of aggregation:

- The primary fact table uses a lower level of aggregation, on `product`, `department`, or `customer`.
- The `rates` dimension table use a higher level of aggregation, on `daily`, `transaction currency`, or `reporting currency`.

The two tables are joined through a relationship join on `date` and `transaction currency`.

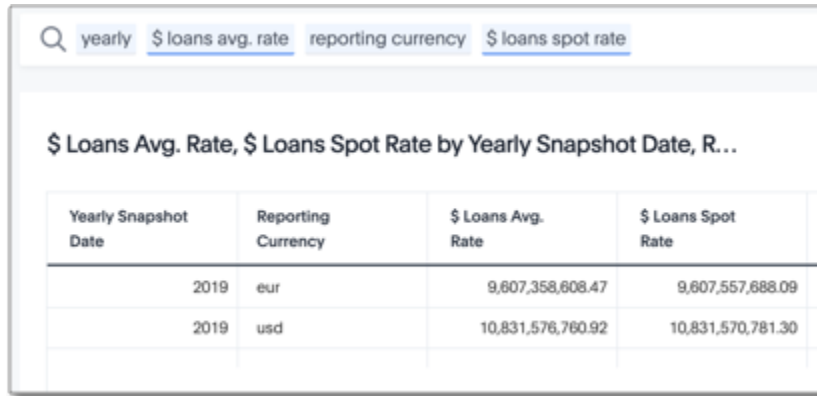
To simplify the scenario, we only use a single `reporting currency`. The join ensures that a single rate value returns each day for each transaction currency.

Valid solution

A valid query that meets our objective may look something like this:

```
sum(group_aggregate (sum(loans)*average (rate),
                    query_groups () + {transaction_currency},
                    query_filters () ))
```


The following search and resulting response returns the dollar value for each year, for each target reporting currency. Note that the dataset contains both euro (€) and US dollars (\$). The `$ Loans Avg. Rate` calculates the average rate of exchange for the entire period. The `$ Loans Spot Rate` applies the rate of exchange on the day of the transaction.



The screenshot shows a search interface with filters: `yearly`, `$ loans avg. rate`, `reporting currency`, and `$ loans spot rate`. Below the filters, the title is "\$ Loans Avg. Rate, \$ Loans Spot Rate by Yearly Snapshot Date, R...". The table below shows the results:

Yearly Snapshot Date	Reporting Currency	\$ Loans Avg. Rate	\$ Loans Spot Rate
2019	eur	9,607,358,608.47	9,607,557,688.09
2019	usd	10,831,576,760.92	10,831,570,781.30

Resolution

1. The `sum(loans)` function aggregates to these attributes:

- `{transaction_currency}` and `query_groups()`

Add additional search columns to this aggregation. Here, this at the level of `reporting currency` and `year`.

- `query_filters()`

Applies any filters entered in the search. Here, there are no filters.

2. Similarly, the `average(rate)` function aggregates to these attributes:

- `{transaction_currency}` and `query_groups()`

Add additional search columns to this aggregation. Here, this at the level of `reporting currency` and `year`.

- `query_filters()`

Applies any filters entered in the search. Here, there are no filters.

3. For each row in this virtual table, the exchange rate applies to the sum of loans:

`sum(loans) * average(rate)`.

- The outer `sum()` function reaggregates the final output as a single row for each yearly reporting currency value.

Note that the default aggregation setting does not reaggregate the result set.

Non-Aggregated Result

We include the following result to provide contrast to an example where ThoughtSpot does not reaggregate the result set. Reaggregation requires the aggregate function, `sum`, to precede the `group_aggregate` function.

In the following scenario, the formula assumes that the default aggregation applies. Here, the result returns 1 row for each `transaction_currency`.

```
group_aggregate (sum(loans )*average (rate ),
                  query_groups() + {transaction_currency},
                  query_filters())
```

Yearly Snapshot Date	Reporting Currency	\$ Loans Avg. Rate (invalid)	\$ Loans Spot Rate by Yearly Snapsho...
2019	eur	319,085,546.96	
2019	eur	424,622,301.81	
2019	eur	8,547,221,568.85	
2019	eur	316,449,200.85	
2019	usd	356,833,240.29	
2019	usd	359,790,133.19	

Yearly Snapshot Date	Reporting Currency	Currency	\$ Loans Avg. Rate (invalid)	\$ Loans Spot Rate by Yearly Snapsho...
2019	eur	nok	319,085,546.96	
2019	eur	dkk	424,622,301.81	
2019	eur	usd	8,547,221,568.85	
2019	eur	sek	316,449,200.85	
2019	usd	sek	356,833,240.29	
2019	usd	nok	359,790,133.19	

Scenario 3: Average period value for semi-additive numbers I

Semi-additive numbers may be aggregated across some, but not all, dimensions. They commonly apply to specific time positions. In this scenario, we have daily position values for home loans, and therefore cannot aggregate on the date dimension.

Valid solution

A valid query that meets our objective may look something like this:

```
average(group_aggregate(sum(loan balance),
                        query_groups() + {date(balance date)},
                        query_filters()))
```

The screenshot shows a data visualization interface with a search bar at the top containing the terms 'yearly', 'first balance', 'last balance', and 'average balance'. Below the search bar, the title 'First Balance, Last Balance, Average Balance by Yearly Balance Date' is displayed. The main content area features a table with the following data:

Yearly Balance Date	First Balance	Last Balance	Average Balance
2018	2,633	2,696	3,470.67

Below the table, it indicates '1 rows total'. At the bottom, there are four summary cards:

- 2018 - 2018** (Yearly Balance Date)
- 2.63K** (First Balance Total)
- 2.7K** (Last Balance Total)
- 3.47K** (Average Balance Min)

Resolution

1. The `sum(loan balance)` function aggregates to the following attributes:

- `{date(balance date)}` and `query_groups()`

Add additional search columns to this aggregation. Here, this at the `yearly` level.

- `query_filters ()`

Applies any filters entered in the search. Here, there are no filters.

2. The `sum(loan balance)` function returns a result for each row in this virtual table.

3. The outer `average()` function reaggregates the final output as a single row for each `year` value.

Scenario 4: Average period value for semi-additive numbers II

Semi-additive numbers may be aggregated across some, but not all, dimensions. They commonly apply to specific time positions. In this scenario, we have daily position values for home loans, and therefore cannot aggregate on the date dimension.

Here, we consider a somewhat different situation than in [Scenario 3 \[See page 8\]](#). In some financial circumstances, the average daily balance has to be calculated, even if the balance does not exist. For example, if a banking account was opened on the 15th of June, business requirements have to consider all the days in the same month, starting with the 1st of June. Importantly, we cannot add these 'missing' data rows to the data set; note that the solution used in [Scenario 3 \[See page 8\]](#) returns an average only for the period that has data, such as June 15th to 30th, not for the entire month of June. The challenge is to ensure that in the daily average formula, the denominator returns the total days in the selected period, not just the days that have transactions:

```
sum(loans) / sum(days_in_period)
```

To solve for this, consider the data model:

- The fact table `transactions` reports the daily position for each account, and uses a `loan` column.
- The dimension table `date` tracks information for each date, starting with the very first transaction, all the way through the most recent transaction. This table includes the expected `date` column, and `days_in_period` column that has a value of 1 in each row.
- Worksheets use the `date` column with keywords such as *weekly*, *monthly*, *yearly* to change the selected period.
- When users run a search with the *monthly* keyword, the denominator must reflect the number of days in each month.

Valid solution

A valid query that meets our objective may look something like this:

The following code *in the denominator definition* returns the total number of days for the period, regardless whether there are transactions, or what filters apply:

```
group_aggregate (sum(days_in_period), {Date}, {})
```

Resolution

1. The `sum(days_in_period)` function aggregates to:

- `{Date}`

No other search columns appear.

- `{}`

We require the entire period, so there are no filters.

Note that the `date` keywords *yearly*, *quarterly*, *monthly*, and *weekly* apply because we use the same column in both the search and the aggregation function. So, the function will result in the following output when it runs with the *yearly* keyword in search:

Year	Result
2016	366
2017	365
2018	365
2019	365
2020	366

2. This data is not reaggregated because we want to return the result at the appropriate `date` level.

Alternate Solution

To return only the number of days that have existing transactions, use the following code in the denominator:

```
sum(days_in_period)
```

Liveboard schedule gating conditions in practice

Note: ThoughtSpot has renamed Pinboards to Liveboards. The functionality remains the same.

When you [schedule a Liveboard job](#) [See page 0], you can add a gating condition that triggers the Liveboard email. If the condition evaluates to `true` at the scheduled time, ThoughtSpot emails the Liveboard to the specified recipients. If it evaluates to `false`, ThoughtSpot does not send the Liveboard.

This article walks you through an example scenario in which you use a gating condition to determine whether to email the Liveboard, sending an alert to specified users.

Gating condition functionality

A gating condition is a statement that returns a single boolean value (`true` or `false`). For example, `sum (revenue) > 100` is a valid condition, but `is_weekend (commit_date)` is not, since it returns a result per row of data. You can use any data source (table, worksheet, or view) for the gating condition, since ThoughtSpot executes the query as an admin with access to all data sources. The gating condition formula and any tables you use in it do not need to be related to the Liveboard the gating condition is for. You can use any valid formula in your statement. ThoughtSpot checks the formula syntax, but does not validate if the formula returns a valid single boolean.

At the scheduled time, ThoughtSpot executes the gating condition query as an admin user. If the condition evaluates to `true`, ThoughtSpot processes the Liveboard.

For a list of valid formulas, see [Formula function reference](#) [See page 0].

Gating condition example

You can leverage gating conditions to send alerts. For example, you begin to notice invalid data appearing in your recent data loads, even though they seem to be successful.

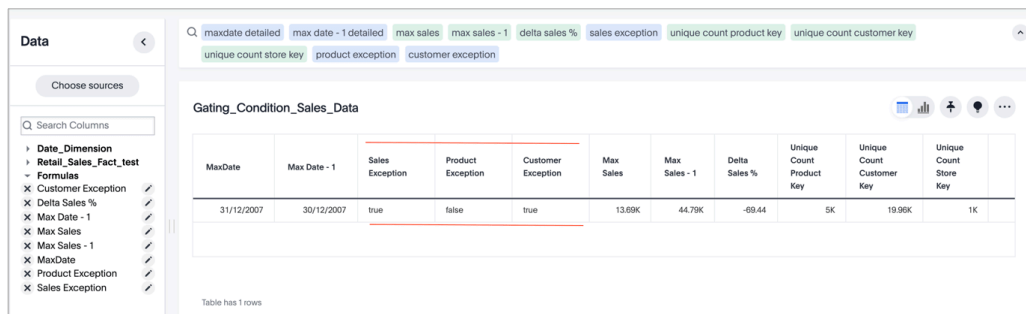
You can validate the data you load by setting several business rules. For example:

- **Sales validation:** The sum of sales today versus the sum of sales yesterday should not vary by more than 20%. A higher amount of variance is unlikely, and probably the result of invalid data.
- **Product validation:** The unique product count should be greater than 4000 but less than 5500. Your company's product count is within that range; any more or less is the result of invalid data.
- **Customer validation:** The unique customer count should be greater than 20000. Your company has more than 20000 customers; any fewer is the result of invalid data.

You can create a view with these business rules, and then create a Liveboard schedule with a gating condition that references these rules. Then, the Liveboard schedule notifies the specified recipients if any of these rules is not met, suggesting a problem with invalid data.

Create a view

After you determine your business rules, create a view that includes these rules. This makes the gating condition formula much simpler and ensures that the formula returns a single result, rather than multiple rows of data.



The screenshot shows a data tool interface with a search bar at the top containing various formulas like 'maxdate detailed', 'max date - 1 detailed', 'max sales', 'max sales - 1', 'delta sales %', 'sales exception', 'unique count product key', 'unique count customer key', 'unique count store key', 'product exception', and 'customer exception'. On the left, there's a 'Data' sidebar with a 'Choose sources' button and a 'Search Columns' field. Below the search field, there's a list of columns including 'Date_Dimension', 'Retail_Sales_Fact_test', and 'Formulas'. The 'Formulas' section is expanded, showing a list of formulas with checkboxes: 'Customer Exception', 'Delta Sales %', 'Max Date - 1', 'Max Sales', 'Max Sales - 1', 'MaxDate', 'Product Exception', and 'Sales Exception'. The main area displays a table titled 'Gating_Condition_Sales_Data' with the following columns: 'MaxDate', 'Max Date - 1', 'Sales Exception', 'Product Exception', 'Customer Exception', 'Max Sales', 'Max Sales - 1', 'Delta Sales %', 'Unique Count Product Key', 'Unique Count Customer Key', and 'Unique Count Store Key'. The table has one row of data for 31/12/2007. Below the table, it says 'Table has 1 rows'.

MaxDate	Max Date - 1	Sales Exception	Product Exception	Customer Exception	Max Sales	Max Sales - 1	Delta Sales %	Unique Count Product Key	Unique Count Customer Key	Unique Count Store Key
31/12/2007	30/12/2007	true	false	true	13.89K	44.79K	-69.44	5K	19.96K	1K

This answer, saved as a view, contains multiple formulas that help determine if a sales, product, or customer exception occurred. The `sales exception` formula uses the `delta sales %` formula, which in turn uses the `max sales` and `max sales - 1` formulas. Because of the complexity of these formulas, it is easier to create a view that you can reference for this gating condition (or multiple gating conditions), rather than trying to create one complicated formula defining the 3 exceptions in the gating condition itself.

In the image above, the sales and customer exceptions are `true` , which should result in an alert after you create the Liveboard schedule with the gating condition.

Create a Liveboard to schedule

Now you have a view where you define the three exceptions that suggest your data loads have invalid data. Next, you must create the Liveboard ThoughtSpot sends out at the scheduled time if the gating condition is met.

ThoughtSpot Validation Alert - Data Validation

Retail sales data validation. If any business rule is exception is TRUE then alert is sent.

Administrator +1

Summary of Exception Rules

Sales Exception	Product Exception	Customer Exception
true	false	true

Table has 1 rows

Sales Exception Rule

% Delta cannot be greater than 20%

Sales Exception	Total Max Sales	Total Max Sales - 1	Total Delta Sales %
true	13.69K	44.79K	-69.44

Table has 1 rows

This Liveboard contains optional answers to provide information about which exceptions failed. The Liveboard can contain any information relevant to the alert.

Note that the Liveboard title, **ThoughtSpot Validation Alert - Data Validation** is the automatic subject of the Liveboard schedule email.

Create the Liveboard schedule

After you create your view and Liveboard, follow the directions in [Schedule a Liveboard job \[See page 0\]](#) to create the Liveboard schedule.

Edit schedule Alert Sales Data Validation for pinboard [ThoughtSpot Validation Alert - Data Validation](#)

Repeats Daily on Every weekday at 09 : 00 hours

UTC

Name* Alert Sales Data Validation

Description Send email alert if Sales Data data validation rules raise exception.

Type ☐ CSV ☒ PDF [Configure Layout Options](#)

Gating condition + Edit condition Delete

damian.walczon@thoughtspot.com

Add Recipients *

Users or groups Add

Emails Add

In this example, the **Repeats** value, which determines how frequently ThoughtSpot emails the Liveboard, is set to send the email every weekday at 9 A.M. You may want to set this value to a more or less frequent cadence, depending on your business needs.

Note that the description in the Liveboard schedule is the automatic body of the Liveboard schedule email.

Define the gating condition

The last step of creating the Liveboard schedule is to define the gating condition. If this condition resolves to **true**, ThoughtSpot sends the Liveboard schedule email at the specified time, to the specified recipients.

In this scenario, you want ThoughtSpot to send the alert if any of the three exceptions is **true**. If they are all false, there is no need to send the alert, since that means that your data loads are successful and the data involved is valid.

Because the view already defines the 3 exceptions, the formula is simple:

```
If (sales exception or product exception or customer exception) then true else false
```

In the gating condition formula editor, it looks like this:

```
if (sales exception or product exception or customer exception ) then true else false
```

The operators (if...then...else) are in blue, and the columns, defined by the view, are in purple.

This formula returns a single boolean value. The gating condition **must** return a single boolean value. It must not return a result per row of data.

Save the gating condition. To create the Liveboard schedule, select **Schedule**.

Now, at the specified time(s) and day(s), ThoughtSpot determines if any of the 3 business exceptions that suggest invalid data are met, and if any of them are `true`, ThoughtSpot sends the alert to the specified recipients, who can fix the issue.

Add a hyperlink to a search

Summary: "Learn how to add a hyperlink to a search."

Note: ThoughtSpot has renamed Pinboards to Liveboards. The functionality remains the same.


You can add a hyperlink directly within a search. Using the concat formula, you can create an external hyperlink, or link your answer to another Liveboard.

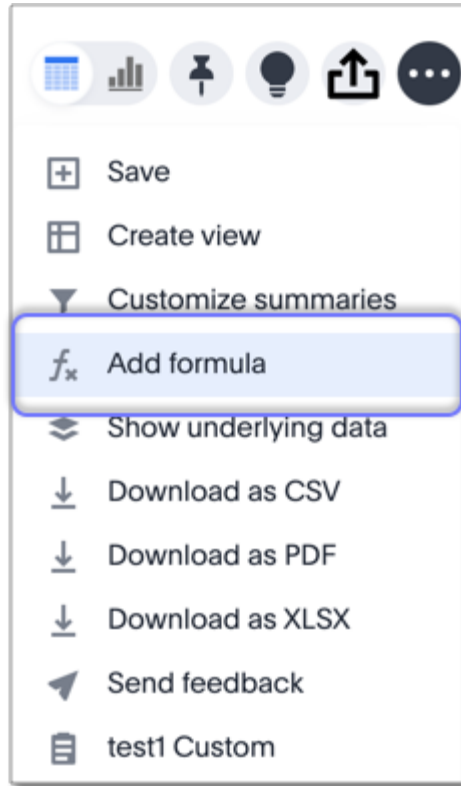
How to add a basic hyperlink

To add an external hyperlink in a search, follow these steps:

1. Start a new search.
Alternatively, choose to edit an existing answer from a Liveboard.
2. If the answer shows a chart, switch to **Table View**.



3. In the upper-right side of the table, click the **More** menu icon , and select **Add formula**.



4. Name your formula.
5. Enter your formula, following the basic syntax:

```
concat({caption}value{/caption}link)
```

- a. For a data set that does not contain properly formed urls, your formula should include the url in full.
For example, to link from the Product Category column to a Google search, use the following syntax to reference the `product category` column, and a Google search using its values:

```
concat("{caption}",product category ,"{/caption}",  
"https://www.google.com/search?q=",product  
category)
```

- b. For a data set that contains properly formed urls, you can simplify the

formula.

For example, to link from the `Fruit` column to the `url`, use the following syntax to reference the two columns, `fruit` and `url`:

```
concat("{caption}",fruit,"{/caption}",url)
```


6. Click **Save**.

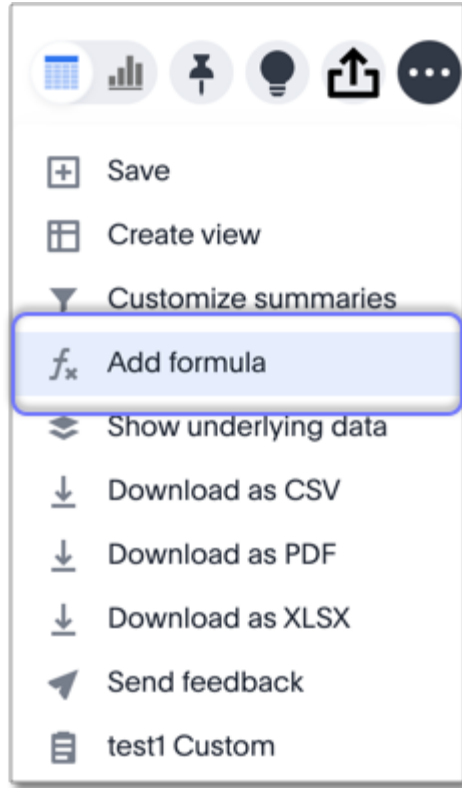
How to link to another Liveboard

To add a hyperlink to another Liveboard, follow these steps:

1. Start a new search.

Alternatively, choose to edit an existing answer from a Liveboard.

2. Choose the target Liveboard for the link.
3. Open your target Liveboard and copy the web address.
4. Within your Search, switch to **Table View**.
5. In the upper-right side of the table, click the **More** menu icon  and select **Add formula**.



6. Name your formula.
7. Enter your formula, following the basic syntax:

```
concat("{caption}pinboard{/caption}http://<thoughtspot_server>:<port>/?<runtime filter>,column name)
```

For example, the formula to link between a search on fruit sales and a Liveboard based on the same data with url `https://.com/#/pinboard/e510f946-f9ce-48ad-a4af-1a40a9cf8add` would be:

```
concat("{caption}pinboard{/caption}https://.com/#/pinboard/e510f946-f9ce-48ad-a4af-1a40a9cf8add/?col1=fruit&op1=eq&val1=", fruit)</code>
```

Here, the runtime filter is operating on the column "fruit," and will only return values that are equal (EQ) to the fruits listed in the columns. To learn more about runtime filters, see [About Runtime Filters \[See page 0\]](#).

8. Click **Save**.