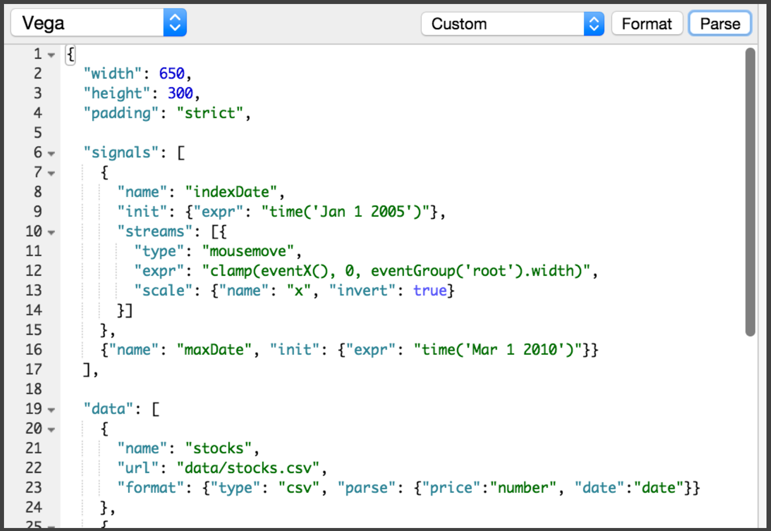
Debugging in the Former Vega IDE

As described in Section 4 of the paper, the former Vega development environment did not have debugging support. Instead, users were required to manually traverse the underlying dataflow graph and scenegraph of Vega’s system internals. In order to provide readers with a sense of this debugging process, we walk through a series of steps that would be required to debug the index chart example described in the paper.

The Former Vega IDE

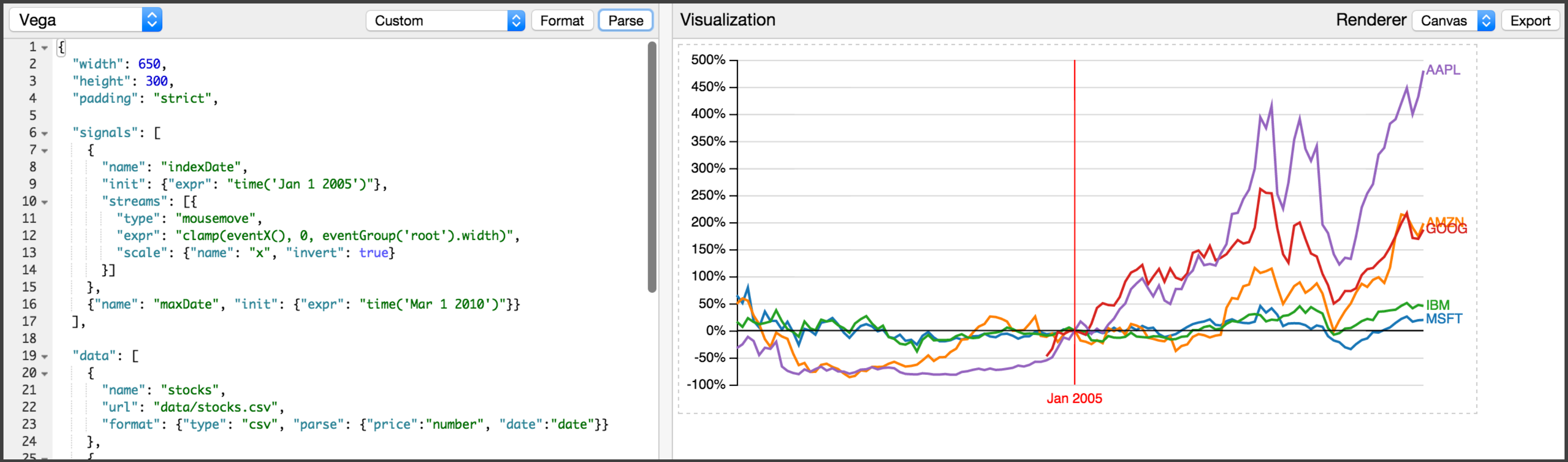
The former Vega IDE consists of **a user-defined specification…**



```
{
  "width": 650,
  "height": 300,
  "padding": "strict",

  "signals": [
    {
      "name": "indexDate",
      "init": {"expr": "time('Jan 1 2005')"},
      "streams": [{
        "type": "mousemove",
        "expr": "clamp(eventX(), 0, eventGroup('root').width)",
        "scale": {"name": "x", "invert": true}
      }]
    },
    {"name": "maxDate", "init": {"expr": "time('Mar 1 2010')"}}
  ],

  "data": [
    {
      "name": "stocks",
      "url": "data/stocks.csv",
      "format": {"type": "csv", "parse": {"price":"number", "date":"date"}}
    },
```

The Former Vega IDE

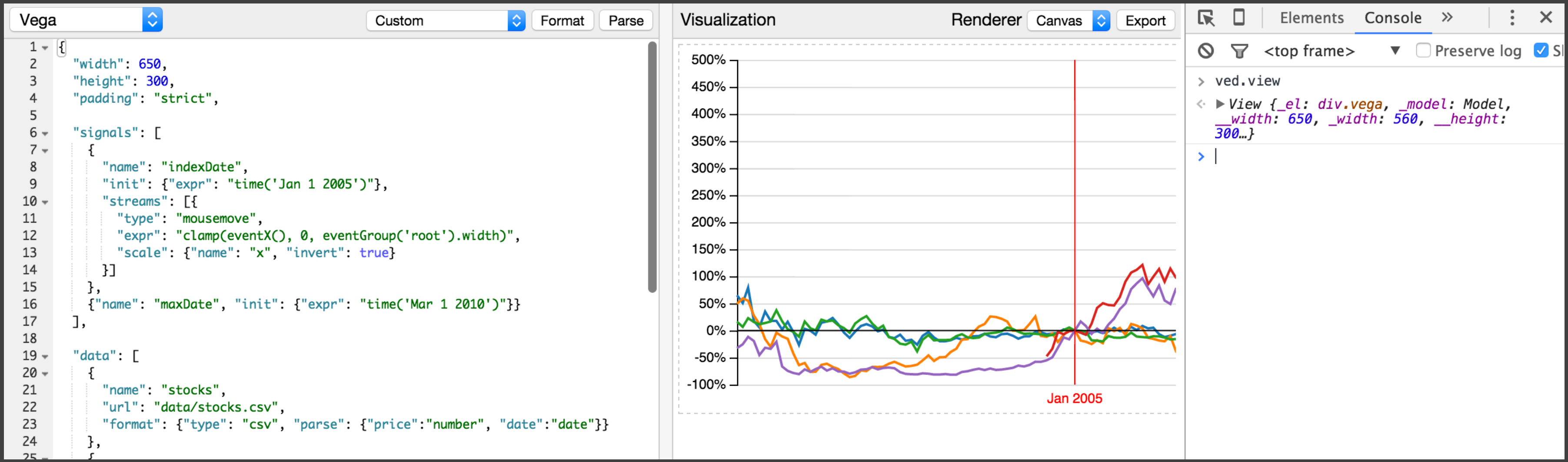
The former Vega IDE consists of a user-defined specification, **an output visualization…**



```
{
  "width": 650,
  "height": 300,
  "padding": "strict",

  "signals": [
    {
      "name": "indexDate",
      "init": {"expr": "time('Jan 1 2005')"},
      "streams": [{
        "type": "mousemove",
        "expr": "clamp(eventX(), 0, eventGroup('root').width)",
        "scale": {"name": "x", "invert": true}
      }]
    },
    {"name": "maxDate", "init": {"expr": "time('Mar 1 2010')"}}
  ],

  "data": [
    {
      "name": "stocks",
      "url": "data/stocks.csv",
      "format": {"type": "csv", "parse": {"price":"number", "date":"date"}}
    },
    {
```

The Former Vega IDE

The former Vega IDE consists of a user-defined specification, an output visualization, **and access to the system internals via the JavaScript console.**

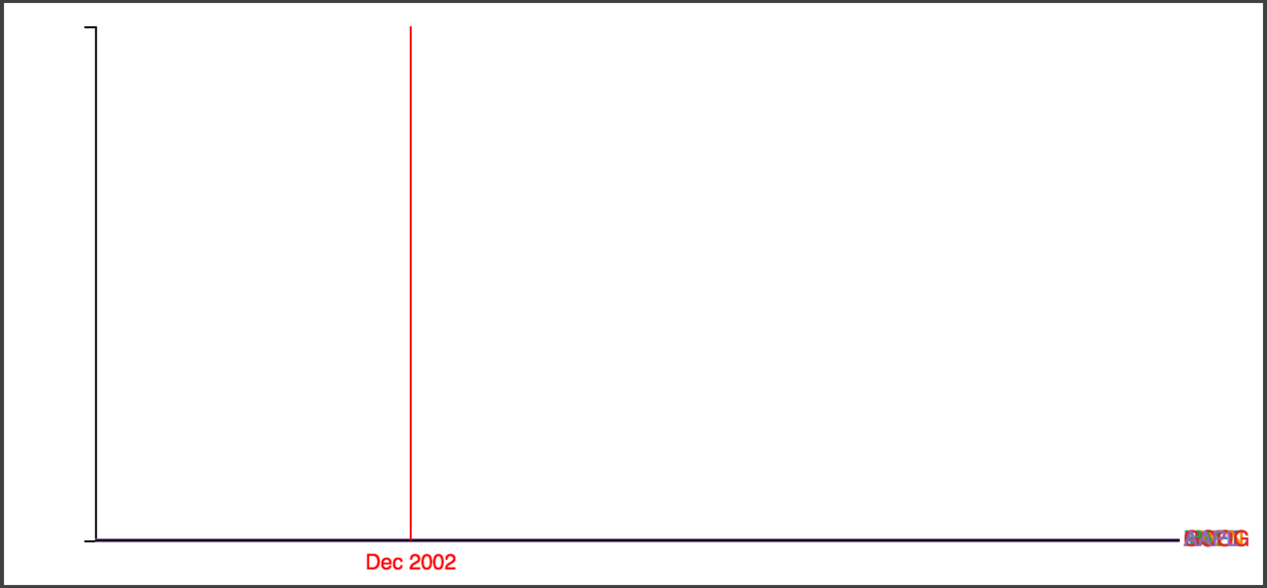


```
{
  "width": 650,
  "height": 300,
  "padding": "strict",

  "signals": [
    {
      "name": "indexDate",
      "init": {"expr": "time('Jan 1 2005')"},
      "streams": [{
        "type": "mousemove",
        "expr": "clamp(eventX(), 0, eventGroup('root').width)",
        "scale": {"name": "x", "invert": true}
      }]
    },
    {"name": "maxDate", "init": {"expr": "time('Mar 1 2010')"}}
  ],

  "data": [
    {
      "name": "stocks",
      "url": "data/stocks.csv",
      "format": {"type": "csv", "parse": {"price":"number", "date":"date"}}
    },
    {
```

```
> ved.view
< ▶View {_el: div.vega, _model: Model, __width: 650, _width: 560, __height: 300…}
> |
```

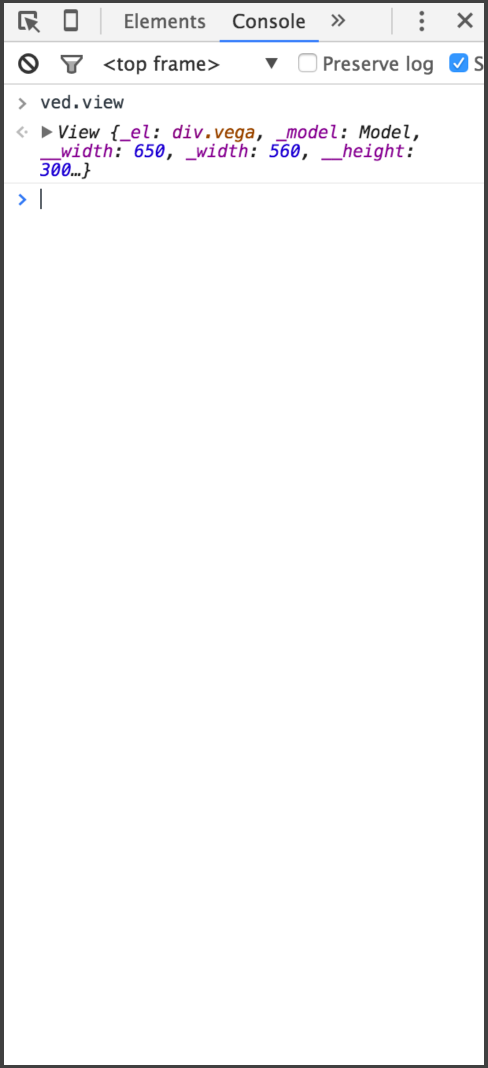
(1) Identify the problem state.

Upon observing an error, the user must navigate to the problem state. In the index chart, the error occurs at a given time location, which is only a few pixels wide and thus hard to hit exactly.



(2) View the system internals.

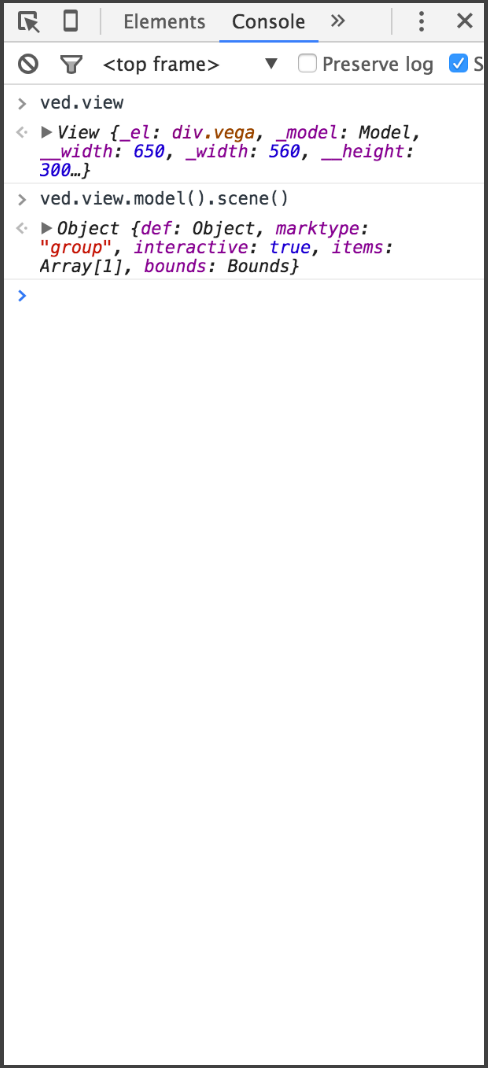
The user must then know that the system internals can be accessed from the command line via the command shown here.



```
> ved.view
< ▶View {_el: div.vega, _model: Model, __width: 650, _width: 560, __height: 300…}
> |
```

(3) What should the user look for?

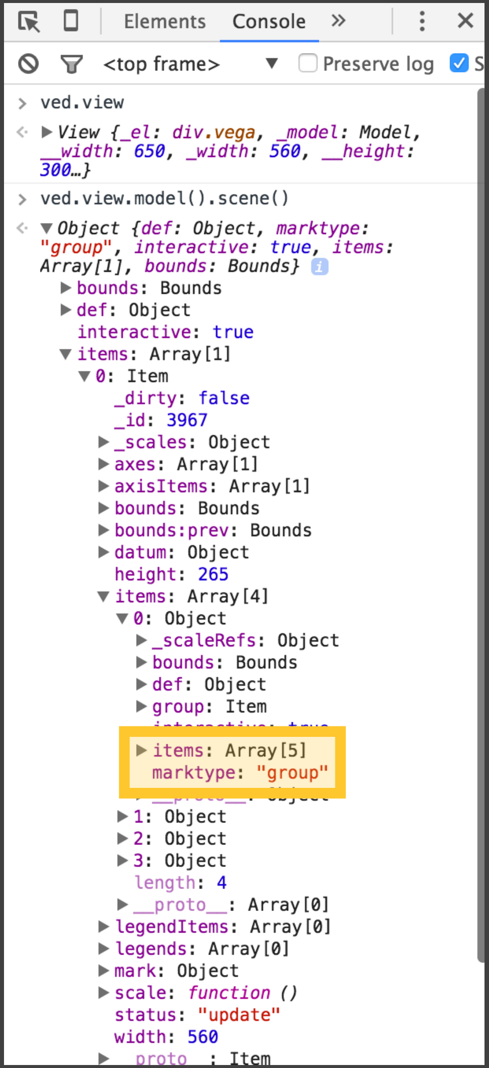
The user notices that no lines appear to be drawn in this error case. In response, the user wants to navigate the scenegraph to identify if this is in fact the case.



```
> ved.view
<· ▶View {_el: div.vega, _model: Model,
  __width: 650, _width: 560, __height:
  300…}
> ved.view.model().scene()
<· ▶Object {def: Object, marktype:
  "group", interactive: true, items:
  Array[1], bounds: Bounds}
>
```

(4) Navigating the scenegraph.

By navigating the scenegraph, the user can locate the group that corresponds to the set of lines…



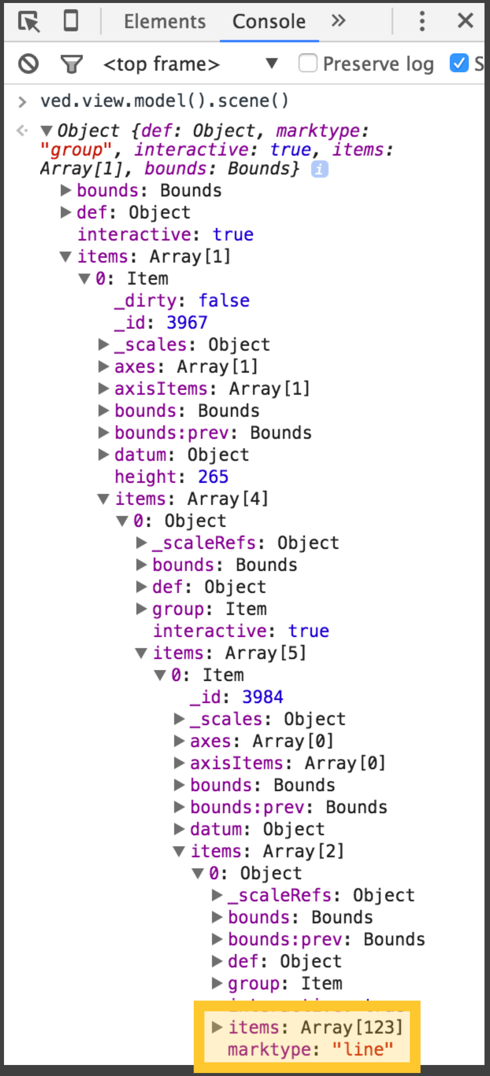
```
> ved.view
<· ▶View {_el: div.vega, _model: Model,
  __width: 650, _width: 560, __height:
  300…}
> ved.view.model().scene()
<· ▼Object {def: Object, marktype:
  "group", interactive: true, items:
  Array[1], bounds: Bounds}
    ▶bounds: Bounds
    ▶def: Object
     interactive: true
    ▼items: Array[1]
      ▼0: Item
         _dirty: false
         _id: 3967
        ▶_scales: Object
        ▶axes: Array[1]
        ▶axisItems: Array[1]
        ▶bounds: Bounds
        ▶bounds:prev: Bounds
        ▶datum: Object
         height: 265
        ▼items: Array[4]
          ▼0: Object
            ▶_scaleRefs: Object
            ▶bounds: Bounds
            ▶def: Object
            ▶group: Item
            ▶items: Array[5]
             marktype: "group"
          ▶1: Object
          ▶2: Object
          ▶3: Object
           length: 4
          ▶__proto__: Array[0]
        ▶legendItems: Array[0]
        ▶legends: Array[0]
        ▶mark: Object
        ▶scale: function ()
         status: "update"
         width: 560
        ▶__proto__: Item
```

(4) Navigating the scenegraph.

By navigating the scenegraph, the user can locate the group that corresponds to the set of lines…

…and locate the part of the scenegraph corresponding to the lines.

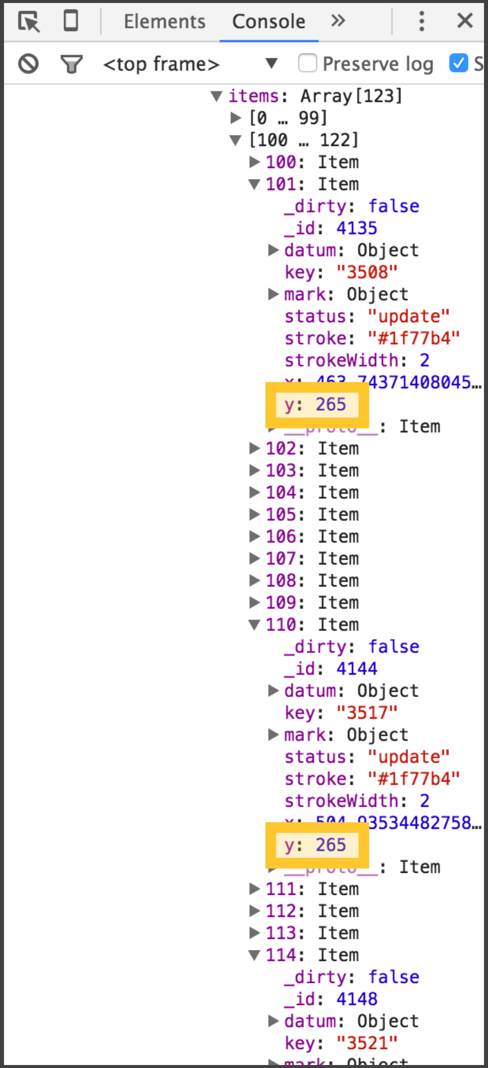
The lines appear to exist so why are they not visible?



```
> ved.view.model().scene()
<· ▼Object {def: Object, marktype:
  "group", interactive: true, items:
  Array[1], bounds: Bounds}
    ▶bounds: Bounds
    ▶def: Object
     interactive: true
    ▼items: Array[1]
      ▼0: Item
         _dirty: false
         _id: 3967
        ▶_scales: Object
        ▶axes: Array[1]
        ▶axisItems: Array[1]
        ▶bounds: Bounds
        ▶bounds:prev: Bounds
        ▶datum: Object
         height: 265
        ▼items: Array[4]
          ▼0: Object
            ▶_scaleRefs: Object
            ▶bounds: Bounds
            ▶def: Object
            ▶group: Item
             interactive: true
            ▼items: Array[5]
              ▼0: Item
                 _id: 3984
                ▶_scales: Object
                ▶axes: Array[0]
                ▶axisItems: Array[0]
                ▶bounds: Bounds
                ▶bounds:prev: Bounds
                ▶datum: Object
                ▼items: Array[2]
                  ▼0: Object
                    ▶_scaleRefs: Object
                    ▶bounds: Bounds
                    ▶bounds:prev: Bounds
                    ▶def: Object
                    ▶group: Item
                    ▶items: Array[123]
                     marktype: "line"
```

(5) Inspect the encoding.

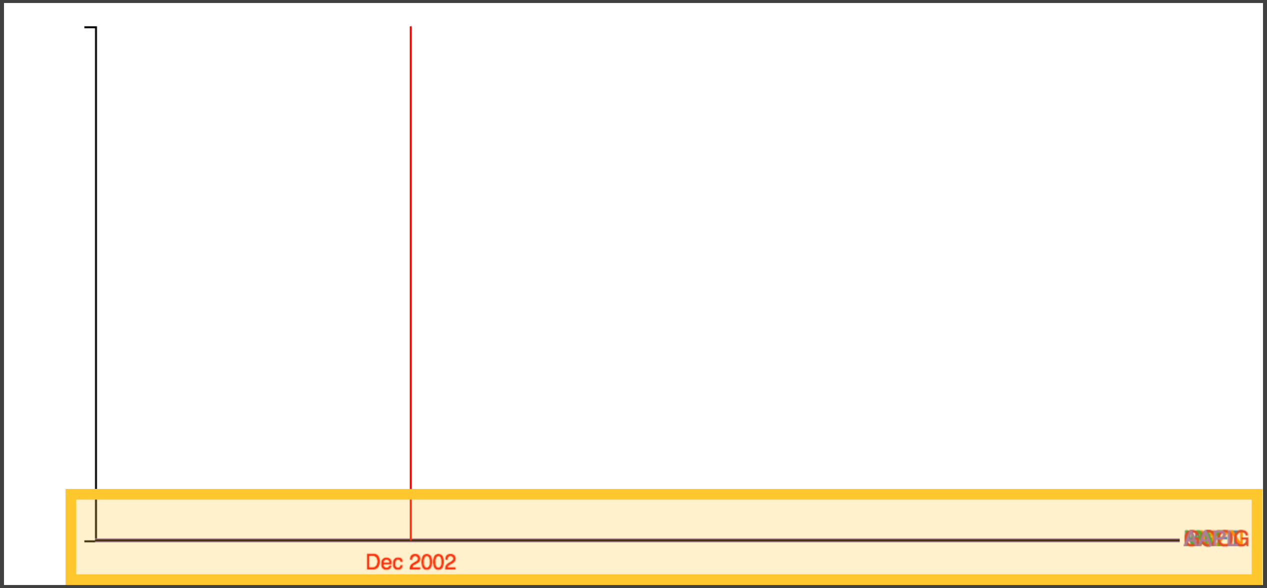
The line is defined by multiple points with an x and y. Based on a random sampling of the values, it seems that while the x value varies the y is always the same (265).



```
▼ items: Array[123]
  ▶ [0 … 99]
  ▼ [100 … 122]
    ▶ 100: Item
    ▼ 101: Item
        _dirty: false
        _id: 4135
      ▶ datum: Object
        key: "3508"
      ▶ mark: Object
        status: "update"
        stroke: "#1f77b4"
        strokeWidth: 2
        x: 463.74371408045…
        y: 265
      ▶ __proto__: Item
    ▶ 102: Item
    ▶ 103: Item
    ▶ 104: Item
    ▶ 105: Item
    ▶ 106: Item
    ▶ 107: Item
    ▶ 108: Item
    ▶ 109: Item
    ▼ 110: Item
        _dirty: false
        _id: 4144
      ▶ datum: Object
        key: "3517"
      ▶ mark: Object
        status: "update"
        stroke: "#1f77b4"
        strokeWidth: 2
        x: 504.03534482758…
        y: 265
      ▶ __proto__: Item
    ▶ 111: Item
    ▶ 112: Item
    ▶ 113: Item
    ▼ 114: Item
        _dirty: false
        _id: 4148
      ▶ datum: Object
        key: "3521"
      ▶ mark: Object
```

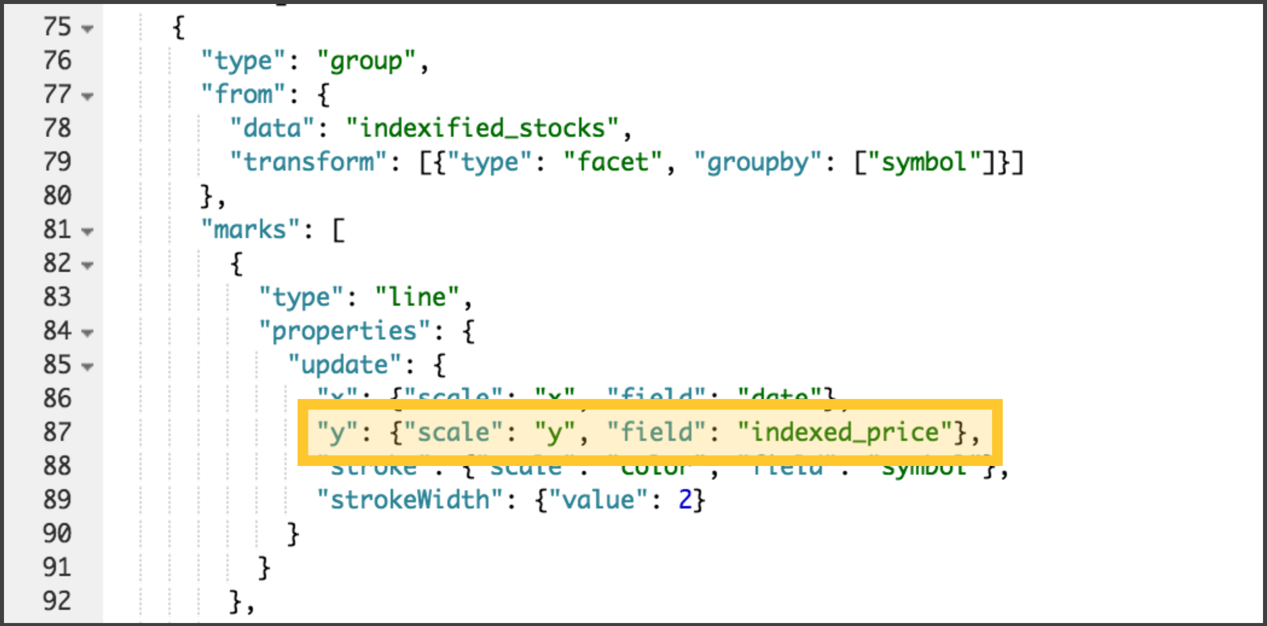
(6) How is y determined?

Inspecting the visualization, the user notices that the lines are not missing, they are flat against the y axis (the text is visible at the end of the line).



(6) How is y determined?

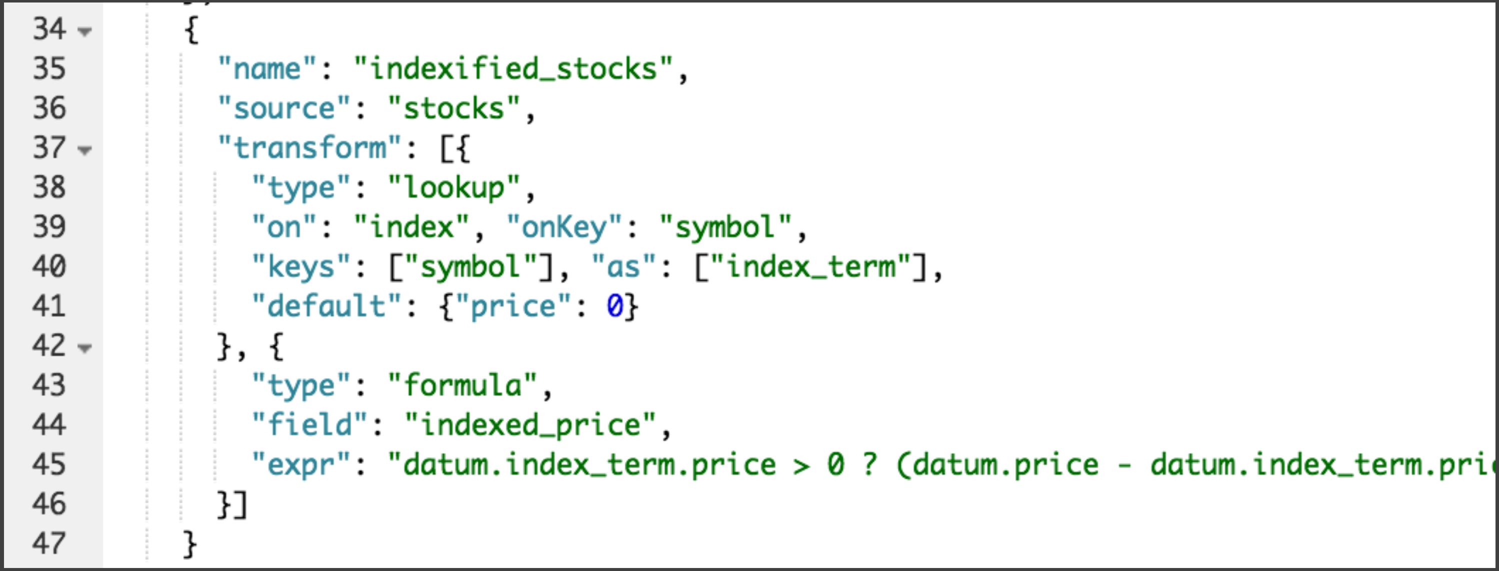
By locating the point in the specification where y is defined, the user can identify that his value is drawn from a data field, indexed_price.



```
{
  "type": "group",
  "from": {
    "data": "indexified_stocks",
    "transform": [{"type": "facet", "groupby": ["symbol"]}]
  },
  "marks": [
    {
      "type": "line",
      "properties": {
        "update": {
          "x": {"scale": "x", "field": "date"},
          "y": {"scale": "y", "field": "indexed_price"},
          "stroke": {"scale": "color", "field": "symbol"},
          "strokeWidth": {"value": 2}
        }
      }
    },
```

(7) How is indexed_price defined?

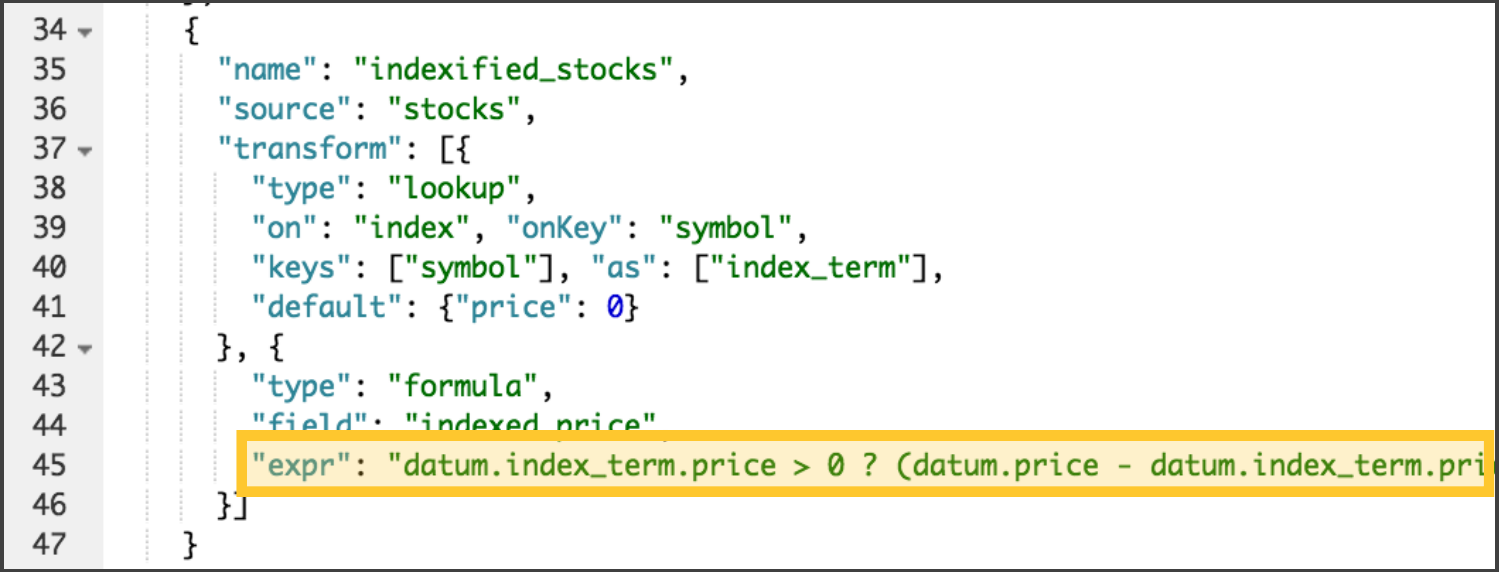
indexed_price is a field in the indexified_stocks dataset, so the user can locate that part of the code…



```
{
  "name": "indexified_stocks",
  "source": "stocks",
  "transform": [{
    "type": "lookup",
    "on": "index", "onKey": "symbol",
    "keys": ["symbol"], "as": ["index_term"],
    "default": {"price": 0}
  }, {
    "type": "formula",
    "field": "indexed_price",
    "expr": "datum.index_term.price > 0 ? (datum.price - datum.index_term.pri
  }]
}
```

(7) How is indexed_price defined?

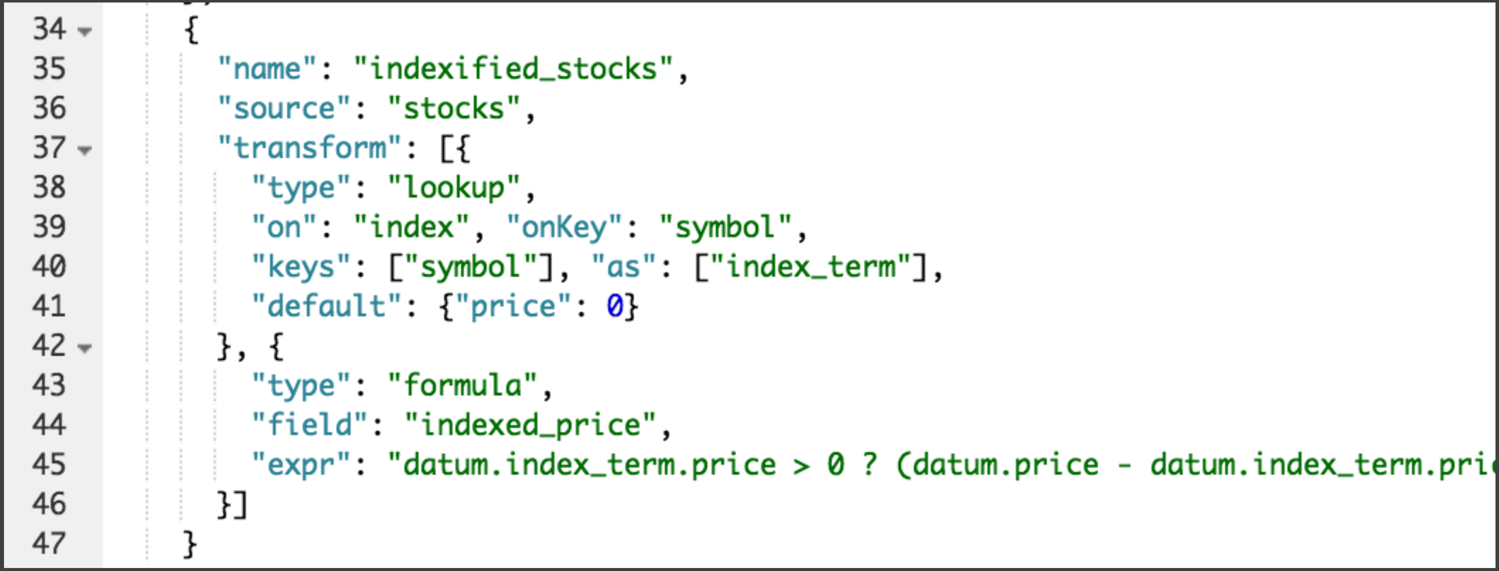
… and find the calculation for indexed_price (which is so long it doesn't fit in the editor!)



```
    {
      "name": "indexified_stocks",
      "source": "stocks",
      "transform": [{
        "type": "lookup",
        "on": "index", "onKey": "symbol",
        "keys": ["symbol"], "as": ["index_term"],
        "default": {"price": 0}
      }, {
        "type": "formula",
        "field": "indexed_price",
        "expr": "datum.index_term.price > 0 ? (datum.price - datum.index_term.pri
      }]
    }
```

(7) How is indexed_price defined?

If index_term is not greater than zero, than indexed_price is zero. So what is index_term?

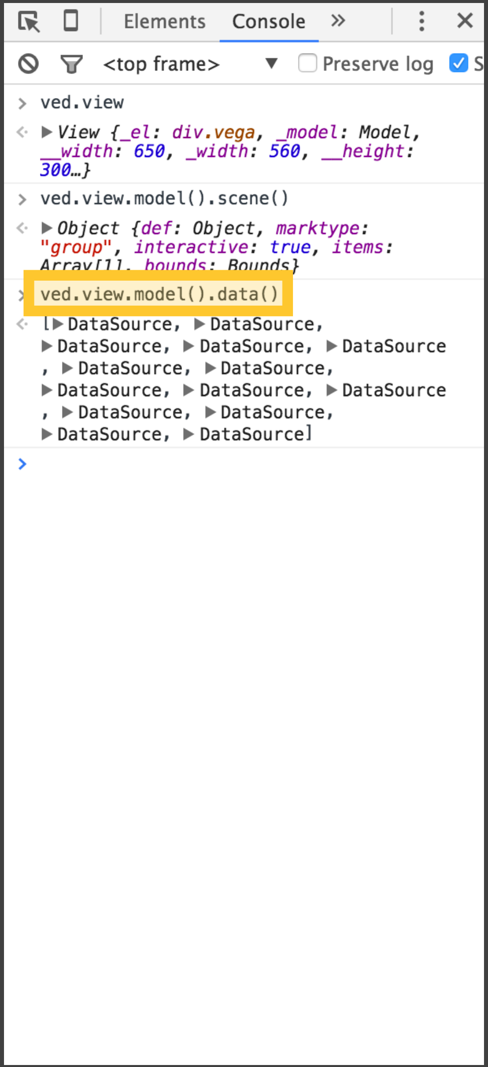


```
{
  "name": "indexified_stocks",
  "source": "stocks",
  "transform": [{
    "type": "lookup",
    "on": "index", "onKey": "symbol",
    "keys": ["symbol"], "as": ["index_term"],
    "default": {"price": 0}
  }, {
    "type": "formula",
    "field": "indexed_price",
    "expr": "datum.index_term.price > 0 ? (datum.price - datum.index_term.pri
  }]
}
```

(8) Inspect the data.

If index_term is not greater than zero, than indexed_price is zero. So what is index_term?

The user can return to the console to locate the data.

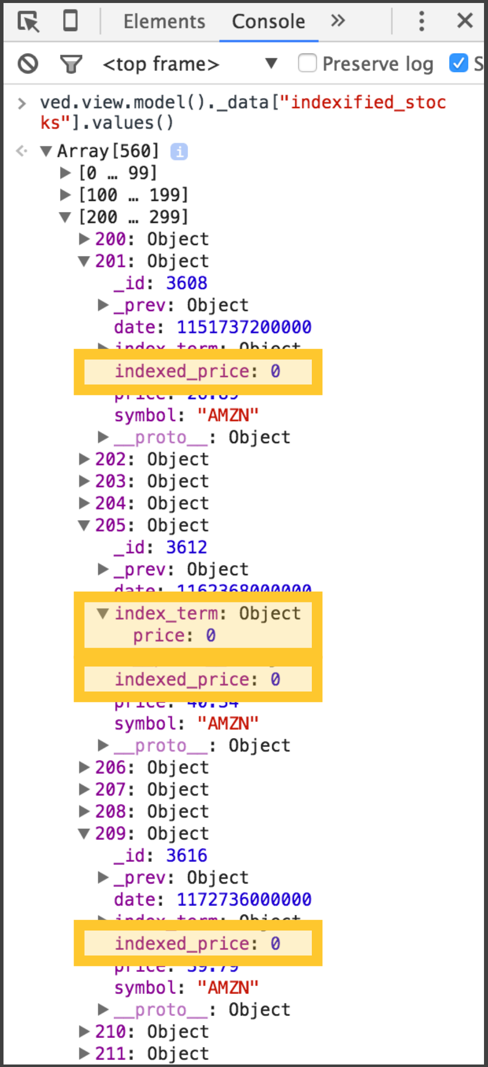


```
> ved.view
<· ▶View {_el: div.vega, _model: Model,
  __width: 650, _width: 560, __height:
  300…}
> ved.view.model().scene()
<· ▶Object {def: Object, marktype:
  "group", interactive: true, items:
  Array[1], bounds: Bounds}
> ved.view.model().data()
<· [▶DataSource, ▶DataSource,
  ▶DataSource, ▶DataSource, ▶DataSource
  , ▶DataSource, ▶DataSource,
  ▶DataSource, ▶DataSource, ▶DataSource
  , ▶DataSource, ▶DataSource,
  ▶DataSource, ▶DataSource]
>
```

(8) Inspect the data.

A sampling of the data for indexified_stocks shows that for all points, both indexed_price and index_term are zero.

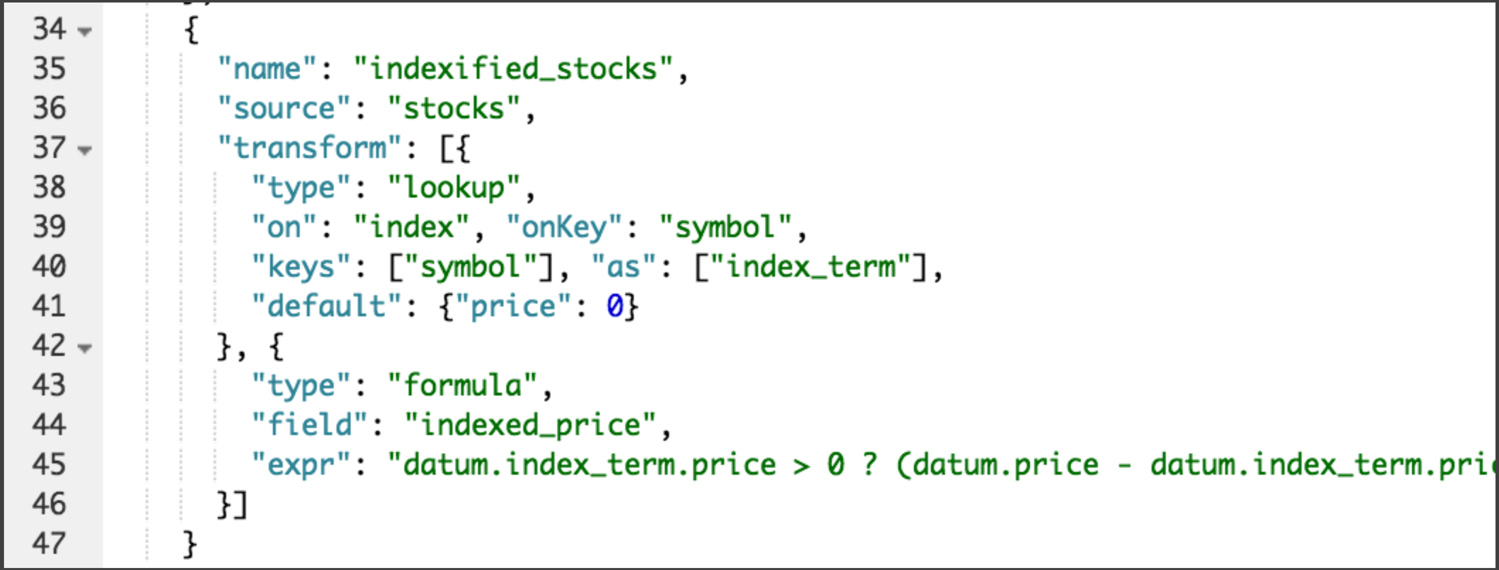
Why is index_term always zero?



```
> ved.view.model()._data["indexified_stoc
  ks"].values()
<· ▼Array[560]
   ▶[0 … 99]
   ▶[100 … 199]
   ▼[200 … 299]
     ▶200: Object
     ▼201: Object
        _id: 3608
       ▶_prev: Object
        date: 1151737200000
       ▶index_term: Object
        indexed_price: 0
        price: 20.89
        symbol: "AMZN"
       ▶__proto__: Object
     ▶202: Object
     ▶203: Object
     ▶204: Object
     ▼205: Object
        _id: 3612
       ▶_prev: Object
        date: 1162368000000
       ▼index_term: Object
          price: 0
        indexed_price: 0
        price: 40.34
        symbol: "AMZN"
       ▶__proto__: Object
     ▶206: Object
     ▶207: Object
     ▶208: Object
     ▼209: Object
        _id: 3616
       ▶_prev: Object
        date: 1172736000000
       ▶index_term: Object
        indexed_price: 0
        price: 39.79
        symbol: "AMZN"
       ▶__proto__: Object
     ▶210: Object
     ▶211: Object
```

(9) Why is index_term zero?

index_term is defined as a lookup on the "index" dataset with a default of zero. What does index look like?

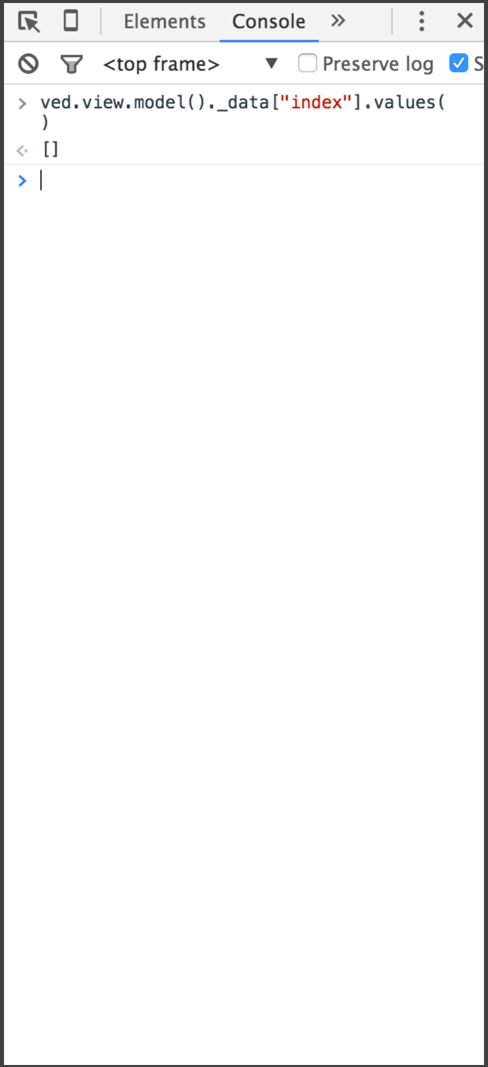


```
  {
    "name": "indexified_stocks",
    "source": "stocks",
    "transform": [{
      "type": "lookup",
      "on": "index", "onKey": "symbol",
      "keys": ["symbol"], "as": ["index_term"],
      "default": {"price": 0}
    }, {
      "type": "formula",
      "field": "indexed_price",
      "expr": "datum.index_term.price > 0 ? (datum.price - datum.index_term.pri
    }]
  }
```

(10) What does index look like?

Returning to the console, the user can look at the values of the index dataset to examine why the lookup is using the default or why the value identified from the lookup is zero…

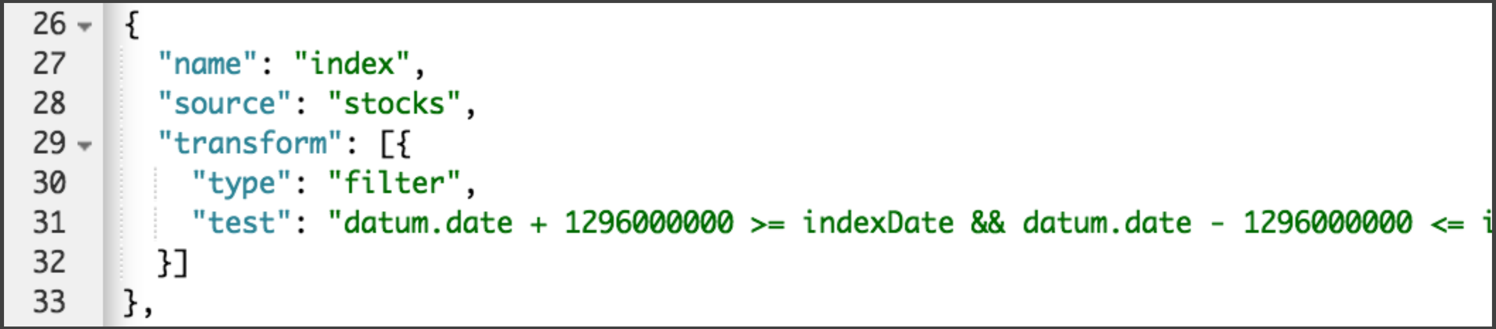
But there are no values. So how is index defined?



```
> ved.view.model()._data["index"].values(
  )
< []
>
```

(11) How is index defined?

Looking at the specification, the user can find the definition of the index dataset.

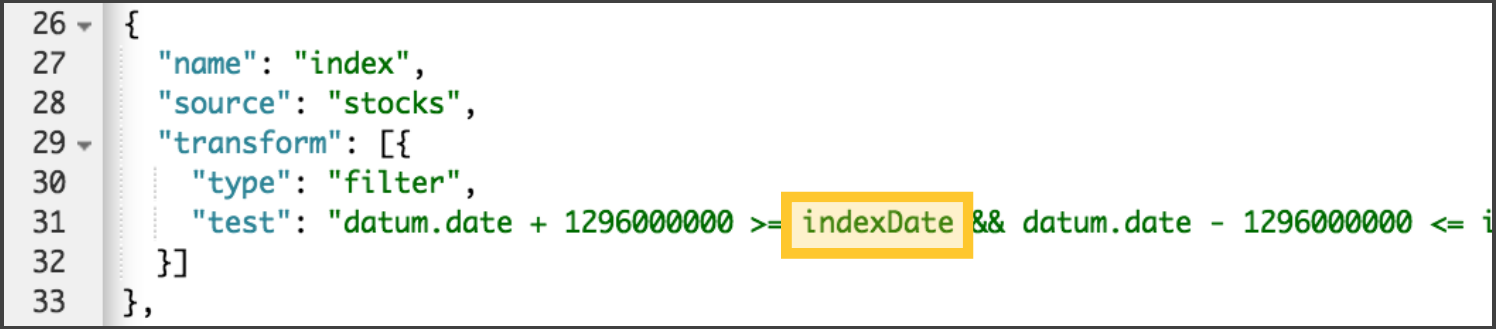


```
{
  "name": "index",
  "source": "stocks",
  "transform": [{
    "type": "filter",
    "test": "datum.date + 1296000000 >= indexDate && datum.date - 1296000000 <= i
  }]
},
```

index is a filter on stocks based on this complex test. The filter seems to remove everything. Why?

(11) How is index defined?

The filter tests each data point in stocks (datum) plus or minus some value against indexDate.

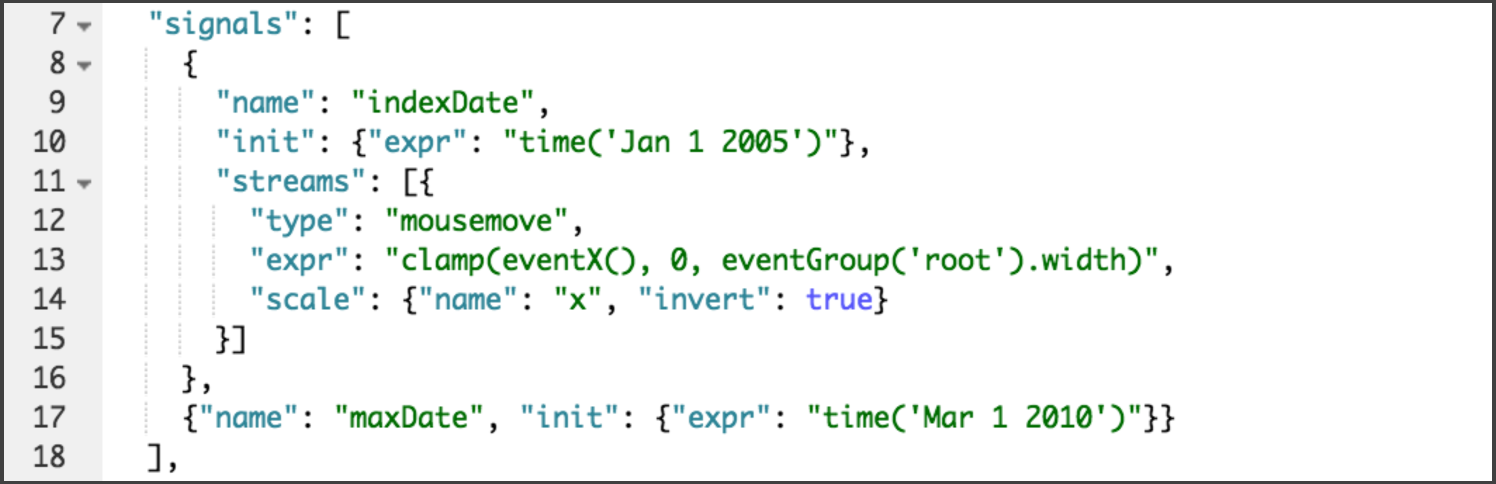


```
{
  "name": "index",
  "source": "stocks",
  "transform": [{
    "type": "filter",
    "test": "datum.date + 1296000000 >= indexDate && datum.date - 1296000000 <= i
  }]
},
```

What is indexDate?

(11a) What is indexDate?

indexDate is defined as a signal that updates on mousemove to get the current time value.

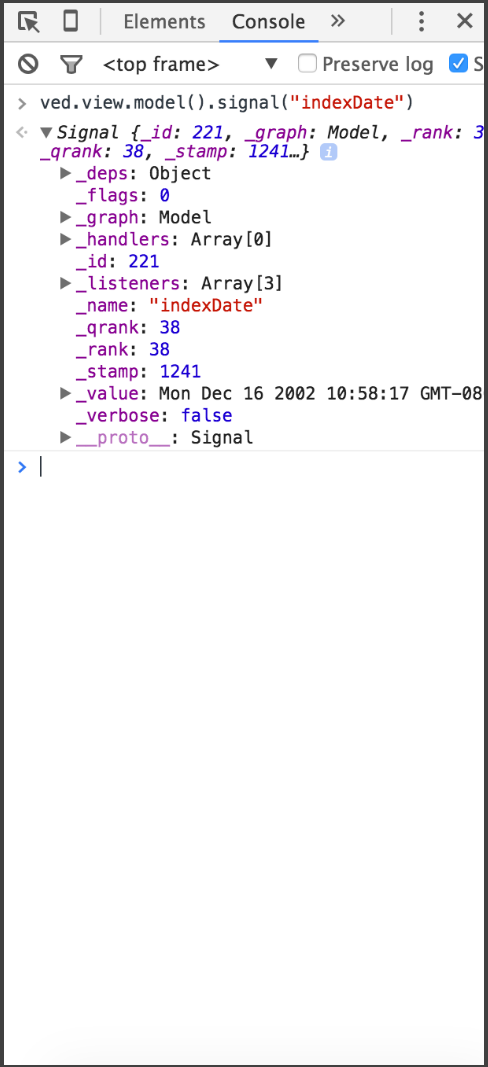


```
"signals": [
  {
    "name": "indexDate",
    "init": {"expr": "time('Jan 1 2005')"},
    "streams": [{
      "type": "mousemove",
      "expr": "clamp(eventX(), 0, eventGroup('root').width)",
      "scale": {"name": "x", "invert": true}
    }]
  },
  {"name": "maxDate", "init": {"expr": "time('Mar 1 2010')"}}
],
```

(11a) What is indexDate?

Using the console, we can find the value of the signal for indexDate to see if there is a problem in the interaction.

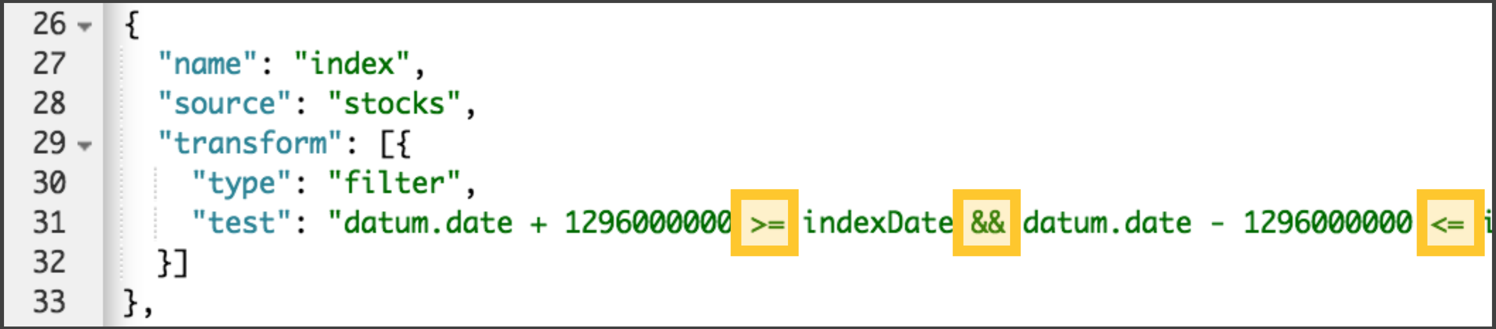
However, it seems that the value (Dec. 16, 2002) is reasonable.



```
> ved.view.model().signal("indexDate")
<· ▼Signal {_id: 221, _graph: Model, _rank: 3
   _qrank: 38, _stamp: 1241…}
    ▶_deps: Object
     _flags: 0
    ▶_graph: Model
    ▶_handlers: Array[0]
     _id: 221
    ▶_listeners: Array[3]
     _name: "indexDate"
     _qrank: 38
     _rank: 38
     _stamp: 1241
    ▶_value: Mon Dec 16 2002 10:58:17 GMT-08
     _verbose: false
    ▶__proto__: Signal
> |
```

(11) How is index defined?

The filter tests each data point in stocks (datum) plus or minus some value against indexDate.



```
{
  "name": "index",
  "source": "stocks",
  "transform": [{
    "type": "filter",
    "test": "datum.date + 1296000000 >= indexDate && datum.date - 1296000000 <= i
  }]
},
```

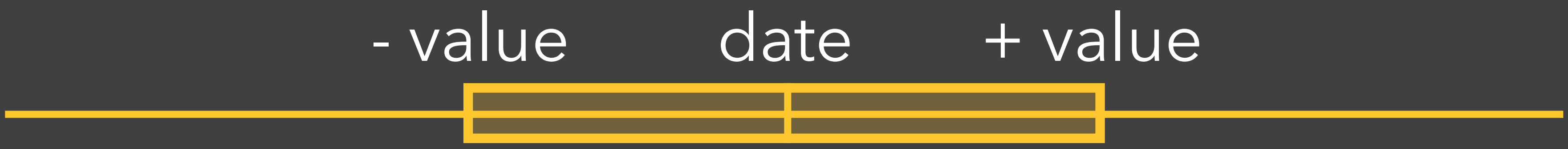
How does the equation work?

(11b) How does the equation work?

The date is kept if:

date - value <= indexDate <= date + value

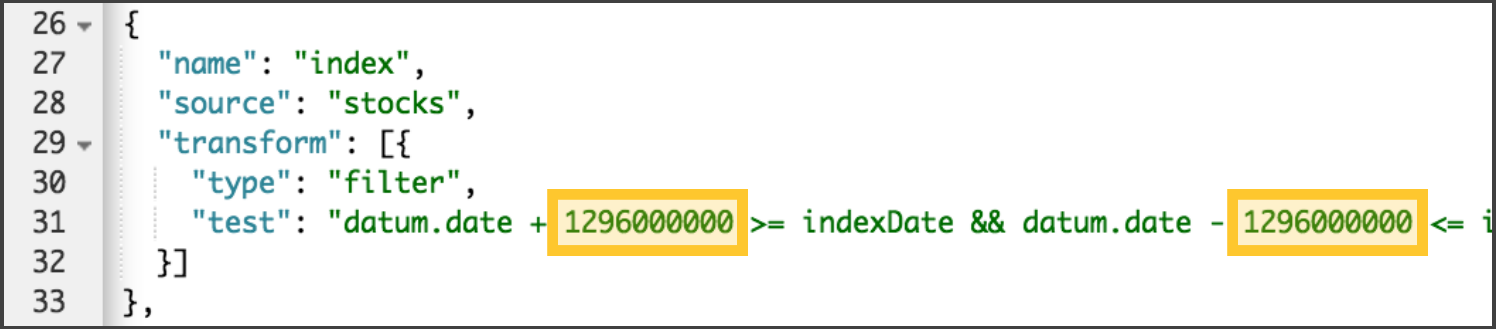
In other words:



if index is in this range, it passes the filter.

(11) How is index defined?

The filter tests each data point in stocks (datum) plus or minus some value against indexDate.

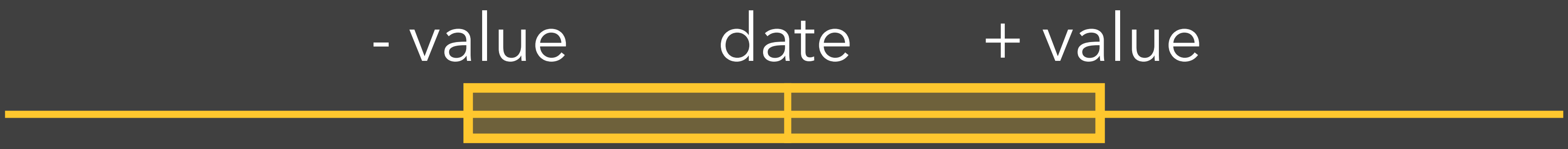


```
{
  "name": "index",
  "source": "stocks",
  "transform": [{
    "type": "filter",
    "test": "datum.date + 1296000000 >= indexDate && datum.date - 1296000000 <= i
  }]
},
```

What is the constant?

(11c) What is the constant?

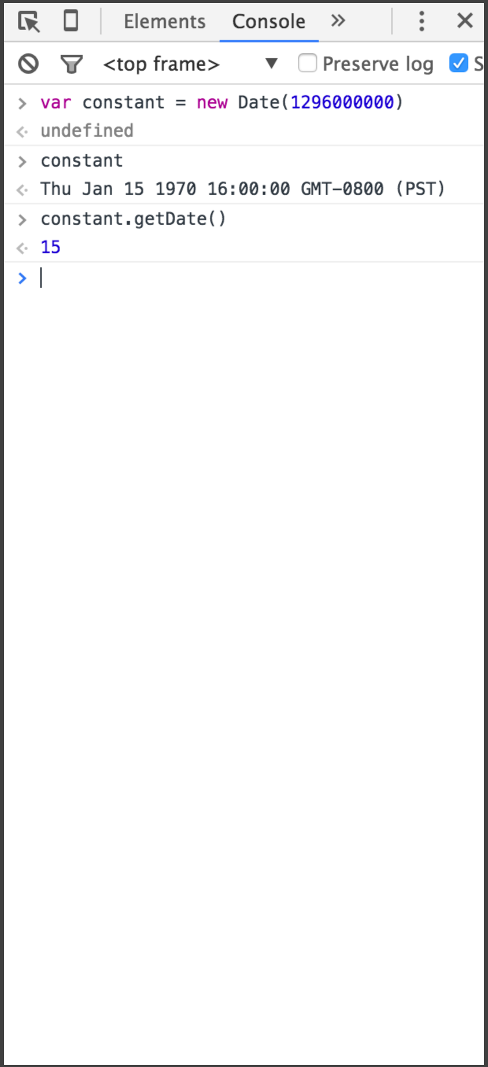
The constant specifies the range in which the data tuples pass the filter, so what is the constant doing?



(11c) What is the constant?

The constant looks like unix time, similar to other values in the data set. We can use the console to explore what it might mean…

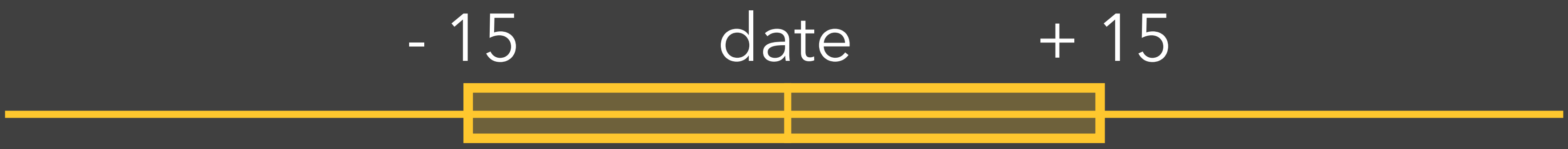
The date is Jan. 15th, which doesn't really make sense for a generic filter. The day is the 15th.



```
> var constant = new Date(1296000000)
< undefined
> constant
< Thu Jan 15 1970 16:00:00 GMT-0800 (PST)
> constant.getDate()
< 15
> |
```

(11c) What is the constant?

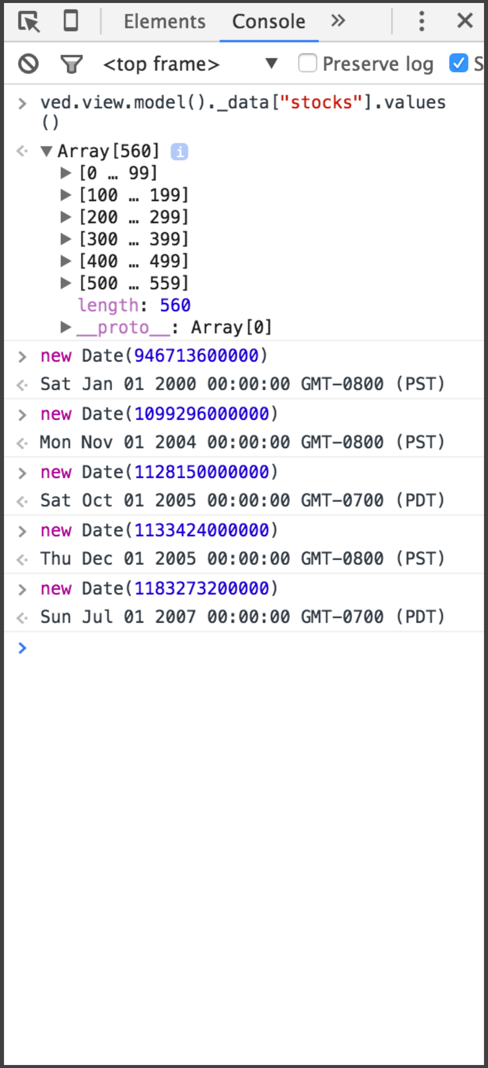
The indexDate was Dec. 16th, so we are looking for all data values such that Dec. 16th falls within this range:



(12) What are the dates in the dataset?

We can use the console to inspect the stocks data set to identify what dates are being tested in the filter…

From a random sampling of points in the dataset, we see that they all correspond to the first of the month.

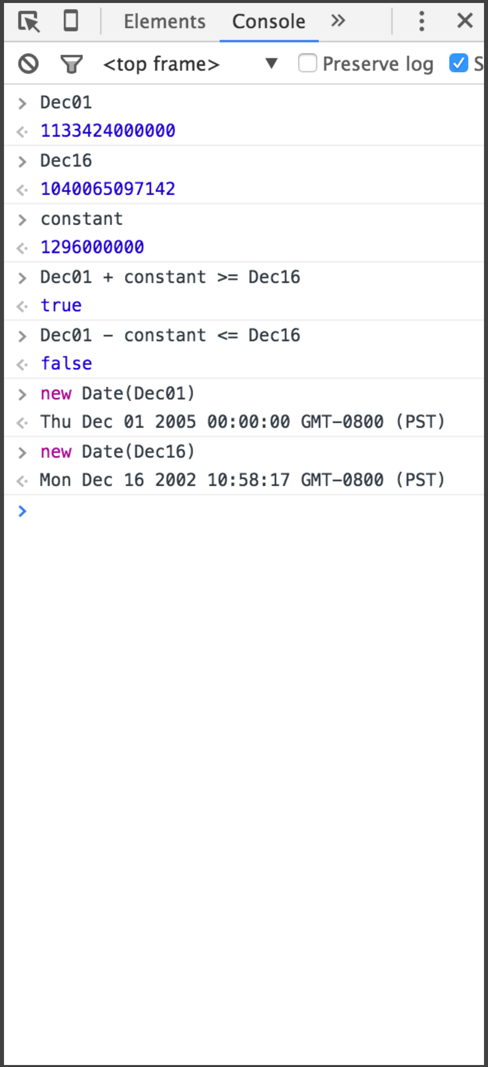


```
> ved.view.model()._data["stocks"].values
  ()
<· ▼Array[560]
   ▶[0 … 99]
   ▶[100 … 199]
   ▶[200 … 299]
   ▶[300 … 399]
   ▶[400 … 499]
   ▶[500 … 559]
    length: 560
   ▶__proto__: Array[0]
> new Date(946713600000)
<· Sat Jan 01 2000 00:00:00 GMT-0800 (PST)
> new Date(1099296000000)
<· Mon Nov 01 2004 00:00:00 GMT-0800 (PST)
> new Date(1128150000000)
<· Sat Oct 01 2005 00:00:00 GMT-0700 (PDT)
> new Date(1133424000000)
<· Thu Dec 01 2005 00:00:00 GMT-0800 (PST)
> new Date(1183273200000)
<· Sun Jul 01 2007 00:00:00 GMT-0700 (PDT)
>
```

(12) What are the dates in the dataset?

So for each data point, what happens with our filter? If we look at Dec. 01, we see that it does *not* pass the filter. And neither do any of the other dates.

Notice the **time** associated with our index point…



```
> Dec01
< 1133424000000
> Dec16
< 1040065097142
> constant
< 1296000000
> Dec01 + constant >= Dec16
< true
> Dec01 - constant <= Dec16
< false
> new Date(Dec01)
< Thu Dec 01 2005 00:00:00 GMT-0800 (PST)
> new Date(Dec16)
< Mon Dec 16 2002 10:58:17 GMT-0800 (PST)
>
```

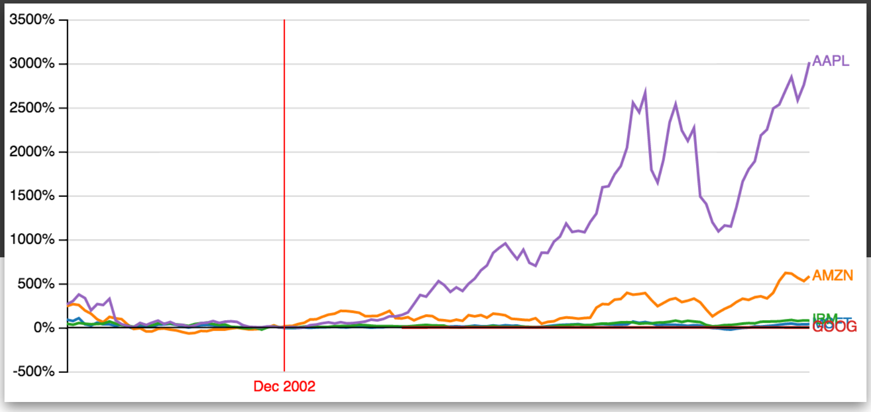
(13) The answer.

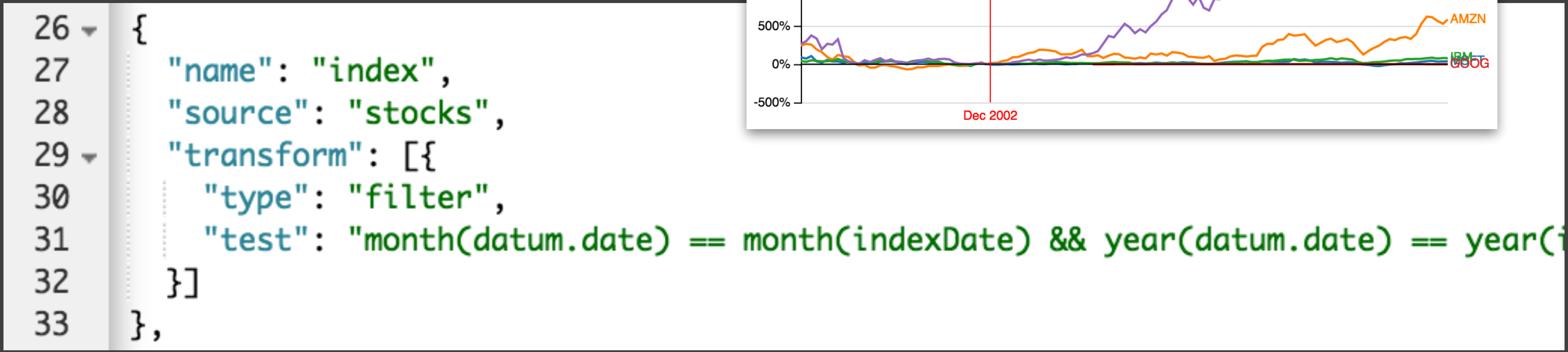
No values pass the filter because the associated time is pushing them out of the range.

The filter is attempting to capture the point associated with the **month** of the index point (the constant is trying to filter a month range around the current date). But this doesn’t work for all index dates and times.

(13) The answer.

By updating the equation for the filter to factor in the **actual** month calculation correctly, the visualization starts to work as expected.





```
{
  "name": "index",
  "source": "stocks",
  "transform": [{
    "type": "filter",
    "test": "month(datum.date) == month(indexDate) && year(datum.date) == year(i
  }]
},
```