# Introduction to rsolr

Michael Lawrence

April 20, 2021

## Contents

1 Introduction 1

2 Demonstration: nycflights13 2
   2.1 The Dataset 2
   2.2 Populating a Solr core 3
   2.3 Restricting by row 5
   2.4 Sorting 9
   2.5 Restricting by field 11
   2.6 Transformation 14
      2.6.1 Advanced note 15
   2.7 Summarization 16

3 Cleaning up 20

## 1 Introduction

The rsolr package provides an idiomatic (R-like) and extensible interface between R and Solr, a search engine and database. Like an onion, the interface consists of several layers, along a gradient of abstraction, so that simple problems are solved simply, while more complex problems may require some peeling and perhaps tears. The interface is idiomatic, syntactically but also in terms of intent. While Solr provides a search-oriented interface, we recognize it as a document-oriented database. While not entirely schemaless, its schema is extremely flexible, which makes Solr an effective database for prototyping and adhoc analysis. R is designed for manipulating data, so rsolr maps common R data manipulation verbs to the Solr database and its (limited) support for analytics. In other words, rsolr is for analysis, not
search, which has presented some fun challenges in design. Hopefully it is useful — we had not tried it until writing this document.

We have interfaced with all of the Solr features that are relevant to data analysis, with the aim of implementing many of the fundamental data munging operations. Those operations are listed in the table below, along with how we have mapped those operations to existing and well-known functions in the base R API, with some important extensions. When called on rsolr data structures, those functions should behave analogously to the existing implementations for data.frame. Note that more complex operations, such as joining and reshaping tables, are best left to more sophisticated frameworks, and we encourage others to implement our extended base R API on top of such systems. After all, Solr is a search engine. Give it a break.

<table>
<thead>
<tr>
<th>Operation</th>
<th>R function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filtering</td>
<td>subset</td>
</tr>
<tr>
<td>Transformation</td>
<td>transform</td>
</tr>
<tr>
<td>Sorting</td>
<td>sort</td>
</tr>
<tr>
<td>Aggregation</td>
<td>aggregate</td>
</tr>
</tbody>
</table>

2 Demonstration: nycflights13

2.1 The Dataset

As part demonstration and part proof of concept, we will attempt to follow the introductory workflow from the dplyr vignette. The dataset describes all of the airline flights departing New York City in 2013. It is provided by the nycflights13 package, so please see its documentation for more details.

```r
> library(nycflights13)
> dim(flights)

[1] 336776 19

> head(flights)

# A tibble: 6 x 19
   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>    <dbl>    <int>          <int>
1   2013    1    1       517           515      2       830           819
2   2013    1    1       533           529      4       850           830
3   2013    1    1       542           540      2       923           850
```
2.2 Populating a Solr core

The first step is getting the data into a Solr core, which is what Solr calls a database. This involves writing a schema in XML, installing and configuring Solr, launching the server, and populating the core with the actual data. Our expectation is that most use cases of rsolr will involve accessing an existing, centrally deployed, usually read-only Solr instance, so those are typically not major concerns. However, to conveniently demonstrate the software, we need to violate all of those assumptions. Luckily, we have managed to embed an example Solr installation within rsolr. We also provide a mechanism for autogenerating a Solr schema from a data.frame. This could be useful in practice for producing a template schema that can be tweaked and deployed in shared Solr installations. Taken together, the process turns out to not be very intimidating.

We begin by generating the schema and starting the demo Solr instance. Note that this instance is really only meant for demonstrations. You should not abuse it like the people abused the poor built-in R HTTP daemon.

> library(rsolr)
> schema <- deriveSolrSchema(flights)
> solr <- TestSolr(schema)

Next, we need to populate the core with our data. This requires a way to interact with the core from R. rsolr provides direct access to cores, as well as two high-level interfaces that represent a dataset derived from a core (rather than the core itself). The two interfaces each correspond to a particular shape of data. SolrList behaves like a list, while SolrFrame behaves like a table (data frame). SolrList is useful for when the data are ragged, as is often the case for data stored in Solr. The Solr schema is so dynamic that we could trivially define a schema with a virtually infinite number of fields, and each document could have its own unique set of fields. However, since our data are tabular, we will use SolrFrame for this exercise.

> sr <- SolrFrame(solr$uri)
Finally, we load our data into the Solr dataset:

```r
> sr[] <- flights
```

This takes a while, since Solr has to generate all sorts of indices, etc.

As `SolrFrame` behaves much like a base R data frame, we can retrieve
the dimensions and look at the head of the dataset:

```r
> dim(sr)
[1] 336776  19

> head(sr)
```

```
DocDataFrame (6x19)

year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
1 2013  1  1     517          515       2      830           819
2 2013  1  1     533          529       4      850           830
3 2013  1  1     542          540       2      923           850
4 2013  1  1     544          545      -1     1004          1022
5 2013  1  1     554          600      -6     812           837
6 2013  1  1     554          558      -4     740           728

arr_delay carrier flight tailnum origin dest air_time distance hour minute
date_hour
1 11   UA  1545 N14228  EWR IAH  227   1400    5    15
2 20   UA  1714 N24211  LGA IAH  227   1416    5    29
3 33   AA  1141 N619AA  JFK MIA  160   1089    5    40
4 -18  B6  725  N804JB  JFK BQN  183   1576    5    45
5 -25  DL  461 N668DN  LGA ATL  116    762    6     0
6 12   UA 1696 N39463  EWR ORD  150    719    5    58
```

Comparing the output above that of the earlier call to `head(flights)`
reveals that the data are virtually identical. As Solr is just a search engine
(on steroids), a significant amount of engineering was required to achieve
that result.
2.3 Restricting by row

The simplest operation is filtering the data, i.e., restricting it to a subset of interest. Even a search engine should be good at that. Below, we use `subset` to restrict to the flights to those departing on January 1 (2013).

```r
> subset(sr, month == 1 & day == 1)

'flights' (ndoc:842, nfield:19)

gender: row: 1

year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
1  2013   1   1   517     515   2     830      819
2  2013   1   1   533     529   4     850      830
3  2013   1   1   542     540   2     923      850
4  2013   1   1   544     545  -1    1004     1022
5  2013   1   1   554     600  -6     812      837

... ...
838 2013   1   1  2356    2359  -3     425      437
839 2013   1   1 <NA>   1630 <NA> <NA>    1815
840 2013   1   1 <NA>   1935 <NA> <NA>    2240
841 2013   1   1 <NA>   1500 <NA> <NA>    1825
842 2013   1   1 <NA>   600  <NA> <NA>    901

arr_delay carrier flight tailnum origin dest air_time distance hour minute
1   11   UA 1545 N14228 EWR IAH  227    1400   5   15
2   20   UA 1714 N24211 LGA IAH  227    1416   5   29
3   33   AA 1141 N619AA JFK MIA  160    1089   5   40
4  -18   B6  725 N804JB JFK BQN  183    1576   5   45
5  -25   DL  461 N668DN LGA ATL  116     762   6    0

... ...
838 -12   B6  727 N588JB JFK BQN  186    1576   23   59
839 <NA> EV 4308 N18120 EWR RDU <NA>   416    16    30
840 <NA> AA  791 N3EHAA LGA DFW <NA>  1389    19   35
841 <NA> AA 1925 N3EVAA LGA MIA <NA>  1096   15   0
842 <NA> B6 125 N618JB JFK FLL <NA>  1069    6   0

time_hour
1 2013-01-01 10:00:00
2 2013-01-01 10:00:00
3 2013-01-01 10:00:00
4 2013-01-01 10:00:00
5 2013-01-01 11:00:00

... ...
838 2013-01-02 04:00:00
```
Note how the records at the bottom contain missing values. Solr does not provide any facilities for missing value representation, but we mimic it by excluding those fields from those documents.

We can also extract ranges of data using the canonical `window()` function:

```r
> window(sr, start=1L, end=10L)
```

```
DocDataFrame (10x19)
   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
1  2013    1    1       517          515          2       830        819
2  2013    1    1       533          529          4       850        830
3  2013    1    1       542          540          2       923        850
4  2013    1    1       544          545         -1       1004       1022
5  2013    1    1       554          600         -6       812        837
6  2013    1    1       554          558         -4       740        728
7  2013    1    1       555          600         -5       913        854
8  2013    1    1       557          600         -3       709        723
9  2013    1    1       557          600         -3       838        846
10 2013    1    1       558          600         -2       753        745

   arr_delay carrier     flight     tailnum  origin  dest  air_time  distance  hour  minute
1         11     UA    1545      N14228   EWR   IAH      227     1400      5      15
2         20     UA    1714      N24211   LGA   IAH      227     1416      5      29
3         33     AA    1141      N619AA  JFK   MIA      160     1089      5      40
4        -18     B6     725      N804JB  JFK   BQN      183     1576      5      45
5        -25     DL     461      N668DN   LGA   ATL      116      762      6      0
6        12     UA    1696      N39463  EWR   ORD      150      719      5      58
7         19     B6     507      N516JB  EWR   FLL      158     1065      6      0
8        -14     EV    5708      N829AS  LGA   IAD       53      229      6      0
9        -8      B6      79      N593JB  JFK   MCO      140      944      6      0
10         8     AA    301      N3ALAA  LGA   ORD      138      733      6      0

   time_hour
1  2013-01-01 10:00:00
2  2013-01-01 10:00:00
3  2013-01-01 10:00:00
4  2013-01-01 10:00:00
Or, as we have already seen, the more convenient:

```r
> head(sr, 10L)
```

```
DocDataFrame (10x19)

year  month  day  dep_time  sched_dep_time  dep_delay  arr_time  sched_arr_time
1     2013     1     1       517        515         2     830          819
2     2013     1     1       533        529         4     850          830
3     2013     1     1       542        540         2     923          850
4     2013     1     1       544        545        -1     1004         1022
5     2013     1     1       554        600        -6     812          837
6     2013     1     1       554        558        -4     740          728
7     2013     1     1       555        600        -5     913          854
8     2013     1     1       557        600        -3     709          723
9     2013     1     1       557        600        -3     838          846
10    2013     1     1       558        600        -2     753          745

arr_delay  carrier  flight  tailnum  origin  dest  air_time  distance  hour  minute
1     11    UA    1545     N14228    EWR    IAH    227      1400     5     15
2     20    UA    1714     N24211    LGA    IAH    227      1416     5     29
3     33    AA    1141    N619AA     JFK    MIA    160      1089     5     40
4     -18    B6     725     N804JB    JFK    BQN    183      1576     5     45
5     -25    DL     461     N668DN    LGA    ATL    116       762     6     0
6     12    UA    1696     N39463    EWR    ORD    150       719     5     58
7     19    B6     507     N516JB    EWR    FLL    158      1065     6     0
8     -14    EV    5708     N829AS    LGA    IAD    53       229     6     0
9      -8    B6      79     N593JB    JFK    MCO    140       944     6     0
10      8    AA     301     N3ALAA    LGA    ORD    138       733     6     0

time_hour
1     2013-01-01 10:00:00
2     2013-01-01 10:00:00
3     2013-01-01 10:00:00
4     2013-01-01 10:00:00
5     2013-01-01 11:00:00
6     2013-01-01 10:00:00
```
We could also call:

to generate a contiguous sequence:

```r
> sr[1:10,]
```

'flights' (ndoc:10, nfield:19)

```r
table(year, month, day, dep_time, sched_dep_time, dep_delay, arr_time, sched_arr_time, arr_delay, carrier, flight, tailnum, origin, dest, air_time, distance, hour, minute)
```

```r
table(arr_delay, carrier, flight, tailnum, origin, dest, air_time, distance, hour, minute)
```

```r
table(time_hour)
```
Unfortunately, it is generally infeasible to randomly access Solr records by index, because numeric indexing is a foreign concept to a search engine. Solr does however support retrieval by a key that has a unique value for each document. These data lack such a key, but it is easy to add one and indicate as such to `deriveSolrSchema()`.

### 2.4 Sorting

To sort the data, we just call `sort()` and describe the order by passing a formula via the `by` argument. For example, we sort by year, breaking ties with month, then day:

```r
> sort(sr, by = ~ year + month + day)
```

<table>
<thead>
<tr>
<th>'flights' (ndoc:336776, nfield:19)</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

... ... ... ... ... ... ... ... ...

| 336772 | 2013 | 12 | 31 | <NA> | 705 | <NA> | <NA> | 931 |
| 336773 | 2013 | 12 | 31 | <NA> | 825 | <NA> | <NA> | 1029 |
| 336774 | 2013 | 12 | 31 | <NA> | 1615 | <NA> | <NA> | 1800 |
| 336775 | 2013 | 12 | 31 | <NA> | 600 | <NA> | <NA> | 735 |
| 336776 | 2013 | 12 | 31 | <NA> | 830 | <NA> | <NA> | 1154 |

<table>
<thead>
<tr>
<th>arr_delay</th>
<th>carrier</th>
<th>flight</th>
<th>tailnum</th>
<th>origin</th>
<th>dest</th>
<th>air_time</th>
<th>distance</th>
<th>hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>UA</td>
<td>1545</td>
<td>N14228</td>
<td>EWR</td>
<td>IAH</td>
<td>227</td>
<td>1400</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>UA</td>
<td>1714</td>
<td>N24211</td>
<td>LGA</td>
<td>IAH</td>
<td>227</td>
<td>1416</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>AA</td>
<td>1141</td>
<td>N619AA</td>
<td>JFK</td>
<td>MIA</td>
<td>160</td>
<td>1089</td>
</tr>
<tr>
<td>4</td>
<td>-18</td>
<td>B6</td>
<td>725</td>
<td>N804JB</td>
<td>JFK</td>
<td>BQN</td>
<td>183</td>
<td>1576</td>
</tr>
<tr>
<td>5</td>
<td>-25</td>
<td>DL</td>
<td>461</td>
<td>N668DN</td>
<td>LGA</td>
<td>ATL</td>
<td>116</td>
<td>762</td>
</tr>
</tbody>
</table>

... ... ... ... ... ... ... ... ...

| 336772 | <NA> | UA | 1729 | <NA> | EWR | DEN | <NA> | 1605 | 7   |
| 336773 | <NA> | US | 1831 | <NA> | JFK | CLT | <NA> | 541  | 8   |
| 336774 | <NA> | MQ | 3301 | N844MQ | LGA | RDU | <NA> | 431  | 16  |
| 336775 | <NA> | UA | 219  | <NA> | EWR | ORD | <NA> | 719  | 6   |
To sort in decreasing order, just pass `decreasing=TRUE` as usual:

```r
> sort(sr, by = ~ arr_delay, decreasing=TRUE)

'flights' (ndoc:336776, nfield:19)

<table>
<thead>
<tr>
<th>year</th>
<th>month</th>
<th>day</th>
<th>dep_time</th>
<th>sched_dep_time</th>
<th>dep_delay</th>
<th>arr_time</th>
<th>sched_arr_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>1</td>
<td>9</td>
<td>641</td>
<td>900</td>
<td>1301</td>
<td>1530</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>6</td>
<td>15</td>
<td>1432</td>
<td>1935</td>
<td>1137</td>
<td>1607</td>
<td>2120</td>
</tr>
<tr>
<td>2013</td>
<td>1</td>
<td>10</td>
<td>1121</td>
<td>1635</td>
<td>1126</td>
<td>1239</td>
<td>1810</td>
</tr>
<tr>
<td>2013</td>
<td>9</td>
<td>20</td>
<td>1139</td>
<td>1845</td>
<td>1014</td>
<td>1457</td>
<td>2210</td>
</tr>
<tr>
<td>2013</td>
<td>7</td>
<td>22</td>
<td>845</td>
<td>1600</td>
<td>1005</td>
<td>1044</td>
<td>1815</td>
</tr>
<tr>
<td>2013</td>
<td>5</td>
<td>4</td>
<td>1816</td>
<td>1820</td>
<td>-4</td>
<td>2017</td>
<td>2131</td>
</tr>
<tr>
<td>2013</td>
<td>5</td>
<td>6</td>
<td>1826</td>
<td>1830</td>
<td>-4</td>
<td>2045</td>
<td>2200</td>
</tr>
<tr>
<td>2013</td>
<td>5</td>
<td>20</td>
<td>719</td>
<td>735</td>
<td>-16</td>
<td>951</td>
<td>1110</td>
</tr>
<tr>
<td>2013</td>
<td>7</td>
<td>1715</td>
<td>1729</td>
<td>1729</td>
<td>-14</td>
<td>1944</td>
<td>2110</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>arr_delay</th>
<th>carrier</th>
<th>flight</th>
<th>tailnum</th>
<th>origin</th>
<th>dest</th>
<th>air_time</th>
<th>distance</th>
<th>hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>-74</td>
<td>AS</td>
<td>7</td>
<td>N551AS</td>
<td>EWR</td>
<td>SEA</td>
<td>281</td>
<td>2402</td>
<td>18</td>
</tr>
<tr>
<td>-75</td>
<td>UA</td>
<td>612</td>
<td>N851UA</td>
<td>EWR</td>
<td>LAX</td>
<td>300</td>
<td>2454</td>
<td>19</td>
</tr>
<tr>
<td>-75</td>
<td>AA</td>
<td>269</td>
<td>N3KCAA</td>
<td>JFK</td>
<td>SEA</td>
<td>289</td>
<td>2422</td>
<td>18</td>
</tr>
</tbody>
</table>

Just as we can use `subset` to restrict by row, we can also use it to restrict by column:

```r
> subset(sr, select=c(year, month, day))

'flights' (ndoc:336776, nfield:3)

year  month  day  
1 2013    1     1
2 2013    1     1
3 2013    1     1
4 2013    1     1
5 2013    1     1
... ... ... 
336772 2013  9  30
336773 2013  9  30
336774 2013  9  30
336775 2013  9  30
336776 2013  9  30
```

The `select` argument is analogous to that of `subset.data.frame`: it is evaluated to set of field names to which the dataset is restricted. The above example is static, so it is equivalent to:

```r
> sr[c("year", "month", "day")]
```

'flights' (ndoc:336776, nfield:3)

    year  month  day
   1 2013      1 1
   2 2013      1 1
   3 2013      1 1
   4 2013      1 1
   5 2013      1 1
   ...
336772 2013   9 30
336773 2013   9 30
336774 2013   9 30
336775 2013   9 30
336776 2013   9 30

But with subset we can also specify dynamic expressions, including ranges:

> subset(sr, select=year:day)

'flights' (ndoc:336776, nfield:3)

    year  month  day
   1 2013      1 1
   2 2013      1 1
   3 2013      1 1
   4 2013      1 1
   5 2013      1 1
   ...
336772 2013   9 30
336773 2013   9 30
336774 2013   9 30
336775 2013   9 30
336776 2013   9 30

And exclusion:

> subset(sr, select=-(year:day))

'flights' (ndoc:336776, nfield:16)

   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
   1       517         515        2      830         819        11
   2       533         529        4      850         830        20
<table>
<thead>
<tr>
<th>carrier</th>
<th>flight</th>
<th>tailnum</th>
<th>origin</th>
<th>dest</th>
<th>air_time</th>
<th>distance</th>
<th>hour</th>
<th>minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>UA</td>
<td>1545</td>
<td>N14228</td>
<td>EWR</td>
<td>IAH</td>
<td>227</td>
<td>1400</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>UA</td>
<td>1714</td>
<td>N24211</td>
<td>LGA</td>
<td>IAH</td>
<td>227</td>
<td>1416</td>
<td>5</td>
<td>29</td>
</tr>
<tr>
<td>AA</td>
<td>1141</td>
<td>N619AA</td>
<td>JFK</td>
<td>MIA</td>
<td>160</td>
<td>1089</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>B6</td>
<td>725</td>
<td>N804JB</td>
<td>JFK</td>
<td>BQN</td>
<td>183</td>
<td>1576</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>DL</td>
<td>461</td>
<td>N668DN</td>
<td>LGA</td>
<td>ATL</td>
<td>116</td>
<td>762</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>9E</td>
<td>3393</td>
<td>&lt;NA&gt;</td>
<td>JFK</td>
<td>DCA</td>
<td>&lt;NA&gt;</td>
<td>213</td>
<td>14</td>
<td>55</td>
</tr>
<tr>
<td>9E</td>
<td>3525</td>
<td>&lt;NA&gt;</td>
<td>LGA</td>
<td>SYR</td>
<td>&lt;NA&gt;</td>
<td>198</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>MQ</td>
<td>3461</td>
<td>N535MQ</td>
<td>LGA</td>
<td>BNA</td>
<td>&lt;NA&gt;</td>
<td>764</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>MQ</td>
<td>3572</td>
<td>N511MQ</td>
<td>LGA</td>
<td>CLE</td>
<td>&lt;NA&gt;</td>
<td>419</td>
<td>11</td>
<td>59</td>
</tr>
<tr>
<td>MQ</td>
<td>3531</td>
<td>N839MQ</td>
<td>LGA</td>
<td>RDU</td>
<td>&lt;NA&gt;</td>
<td>431</td>
<td>8</td>
<td>40</td>
</tr>
</tbody>
</table>

### time_hour

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2013-01-01</td>
<td>10:00:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2013-01-01</td>
<td>10:00:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2013-01-01</td>
<td>10:00:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2013-01-01</td>
<td>10:00:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2013-01-01</td>
<td>11:00:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solr also has native support for globs:

```r
> sr[c("arr_*", "dep_*")]

'flights' (ndoc:336776, nfield:4)

    arr_time arr_delay dep_time dep_delay
   1    830       11    517        2
```
While we are dealing with fields, we should mention that renaming is also (in principle) possible:

```r
> ### FIXME: broken in current Solr CSV writer
> ### rename(sr, tail_num = "tailnum")
```

### 2.6 Transformation

To compute new columns from existing ones, we can, as usual, call the `transform` function:

```r
> sr2 <- transform(sr,
+                  gain = arr_delay - dep_delay,
+                  speed = distance / air_time * 60)
> sr2[c("gain", "speed")]
```

'flights' (ndoc:336776, nfield:1)

<table>
<thead>
<tr>
<th>gain</th>
<th>speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>-17</td>
</tr>
<tr>
<td>5</td>
<td>-19</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... ... ...

336772 <NA> <NA> <NA> <NA>
336773 <NA> <NA> <NA> <NA>
336774 <NA> <NA> <NA> <NA>
336775 <NA> <NA> <NA> <NA>
336776 <NA> <NA> <NA> <NA>
2.6.1 Advanced note

The `transform` function essentially quotes and evaluates its arguments in the given frame, and then adds the results as columns in the return value. Direct evaluation affords more flexibility, such as constructing a table with only the newly computed columns. By default, evaluation is completely eager — each referenced column is downloaded in its entirety. But we can make the computation lazier by calling `defer` prior to the evaluation via `with`:

```r
> with(defer(sr), data.frame(gain = head(arr_delay - dep_delay),
+                        speed = head(distance / air_time * 60)))

   gain   speed
1    9 370.0440
2   16 374.2731
3   31 408.3750
4  -17 516.7213
5  -19 394.1379
6   16 287.6000
```

Note that this approach, even though it is partially deferred, is potentially less efficient than `transform` two reasons:

1. It makes two requests to the database, one for each column,

2. The two result columns are downloaded eagerly, since the result must be a `data.frame` (and thus practicalities required us to take the `head` of each promised column prior to constructing the data frame).

We can work around the second limitation by using a more general form of data frame, the `DataFrame` object from S4Vectors:

```r
> with(defer(sr),
+       S4Vectors::DataFrame(gain = arr_delay - dep_delay,
+                              speed = distance / air_time * 60))

DataFrame with 336776 rows and 2 columns

   gain  speed
<SolrFunctionPromise> <SolrFunctionPromise>
1     9 370.0440
2    16 374.2731
3    31 408.3750
```
Note that we did not need to take the head of the individual columns, since `DataFrame` does not require the data to be stored in-memory as a base R vector.

### 2.7 Summarization

Data summarization is about reducing large, complex data to smaller, simpler data that we can understand.

A common type of summarization is aggregation, which is typically defined as a three step process:

1. Split the data into groups, usually by the interaction of some factor set,
2. Summarize each group to a single value,
3. Combine the summaries.

Solr natively supports the following types of data aggregation:

- `mean`
- `min`, `max`
- `median`, `quantile`
- `var`, `sd`
- `sum`
- `count` (table),
- counting of unique values (for which we introduce `nunique`).
The rsolr package combines and modifies these operations to support high-level summaries corresponding to the R functions `any`, `all`, `range`, `weighted.mean`, `IQR`, `mad`, etc.

A prerequisite of aggregation is finding the distinct field combinations that correspond to each correspond to a group. Those combinations themselves constitute a useful summary, and we can retrieve them with `unique`:

```r
> unique(sr["tailnum"])
```

```
DocDataFrame (4044x1)
  tailnum
  1  D942DN
  2  NOEGMQ
  3  N10156
  4  N102UW
  5  N103US
...  ...
4040  N998AT
4041  N998DL
4042  N999DN
4043  N9EAMQ
4044  <NA>
```

```r
> unique(sr[c("origin", "tailnum")])
```

```
DocDataFrame (7944x2)
  origin tailnum
  1 EWR  NOEGMQ
  2 EWR  N10156
  3 EWR  N102UW
  4 EWR  N103US
  5 EWR  N104UW
...  ...  ...
7940 LGA  N998AT
7941 LGA  N998DL
7942 LGA  N999DN
7943 LGA  N9EAMQ
7944 LGA  <NA>
```

Solr also supports extracting the top or bottom N documents, after ranking by some field, optionally by group.
The convenient, top-level function for aggregating data is `aggregate`. To compute a global aggregation, we just specify the computation as an expression (via a named argument, mimicking `transform`):

```r
> aggregate(sr, delay = mean(dep_delay, na.rm=TRUE))
     delay
1 12.63907
```

It is also possible to specify a function (as the `FUN` argument), which would be passed the entire frame.

As with `stats::aggregate`, we can pass a grouping as a formula:

```r
> delay <- aggregate(~ tailnum, sr, 
+                    count = TRUE, 
+                    dist = mean(distance, na.rm=TRUE), 
+                    delay = mean(arr_delay, na.rm=TRUE))
> delay <- subset(delay, count > 20 & dist < 2000)
```

The special `count` argument is a convenience for the common case of computing the number of documents in each group.

Here is an example of using `nunique` and `ndoc`:

```r
> head(aggregate(~ dest, sr, 
+                 nplanes = nunique(tailnum), 
+                 nflights = ndoc(tailnum)))
   dest nplanes nflights
   ABQ   108       254
   ACK    58       265
   ALB   172       439
   ANC     6        8
   ATL  1180      17215
   AUS   993      2439
```

There is limited support for dynamic expressions in the aggregation formula. At a minimum, the expression should evaluate to logical. For example, we can condition on whether the distance is more than 1000 miles.

```r
> head(aggregate(~ I(distance > 1000) + tailnum, sr, 
+                  delay = mean(arr_delay, na.rm=TRUE)))
```
I(distance > 1000) tailnum delay
1 FALSE D942DN 31.500000
2 FALSE NOEGMQ 8.986755
3 FALSE N10156 13.701149
4 FALSE N102UW 2.937500
5 FALSE N103US -6.934783
6 FALSE N104UW 1.804348

It also works for values naturally coercible to logical, such as using the modulus to identify odd numbers. For clarity, we label the variable using `transform` prior to aggregating.

```r
> head(aggregate(~ odd + tailnum, transform(sr, odd = distance %% 2),
+              delay = mean(arr_delay, na.rm=TRUE)))
odd tailnum delay
1 FALSE D942DN 31.500000
2 FALSE NOEGMQ 8.589520
3 FALSE N10156 7.797753
4 FALSE N102UW 19.000000
5 FALSE N103US -7.285714
6 FALSE N104UW 20.700000
```

Aggregate and subset in the same command, as with data.frame:

```r
> head(aggregate(~ tailnum, sr,
+               subset = distance > 500,
+               delay = mean(arr_delay, na.rm=TRUE)))
tailnum delay
1 D942DN 31.500000
2 NOEGMQ 8.919580
3 N10156 12.009174
4 N102UW 2.937500
5 N103US -6.934783
6 N104UW 1.804348
```

Aggregate the entire dataset:

```r
> aggregate(sr, delay = mean(arr_delay, na.rm=TRUE))
delay
1 6.895377
```
3 Cleaning up

Having finished our demonstration, we kill our Solr server:

> solr$kill()