主键、基于二分法搜索的subset

重新读取数据集

```
setwd("C:\\Users\\lenovo\\Documents\\软件学院\\大数据班\\R语言基础课件") #改变工作目录到csv文件所在目录
library(data.table)
flights <- fread("flights14.csv")
```

主键

本节内容

- 介绍“主键”的概念，在参数i里面，设置并使用主键进行基于快速二分法搜索的subset。
- 学习如何将基于主键的subset，与参数i和by相结合，就像以前做的一样。
- 学习另外两个有用的参数 mult 和 nomatch
- 总结一下主键的优越性：基于快速二分法搜索的subset的表现，并和传统的vector scan approach对比。

什么是主键（1/2）

- 所有的data.frame都有一个行名的属性。
- 行名，或多或少，算是一个data.frame的索引。然而，

  1. 每行都有且只有一个行名。 但是，一个人可能有两个名字，比如名字和中间名。当编纂电话簿的时候，这就非常有用。

  2. 行名必须是独一无二的。

- 当我们使用as.data.table()将data.frame转化为data.table时，行名被重置。因为**data.table从不使用行名**。

什么是主键（2/2）

- 在data.table里，我们使用主键。主键是更有效的行名。
- **主键**及其特性
  - 我们可以对**多个列设置主键**，这些列可能是不同的类型－integer, numeric, character, factor, integer64等等。但还不支持list和complex。
  - **不强制唯一性**，也就是说，**不同列的主键可以是一样的**。既然行可以通过主键排序，那么排序的时候，具有同样主键的一些行，会被排在一起。
  - **设置主键**这个**过程**分两步：
    1. 根据指定的列，对data.table重新排序，而且总是按升序排列。
    2. 对于data.table，通过设置一个叫做 sorted 的属性，来把那些列标记为主键列。
       - 既然是排序，**一个data.table最多只能有一个主键**，因为它不能按照两种方法排序。

设置 / 获取 / 使用主键（1/3）

- 将 **origin**列设置为主键

  setkey(flights, **origin)**
- 说明：
  - 可以给函数setkey() 传入列名作为参数，不需要引号。这在交互式使用的时候特别方便。
  - 还可以给函数setkey**v**() 传入一个列名向量，以便**设置多个列名作为主键**。
  - 我们不需要将结果赋值给一个变量。因为，setkey() 和 setkeyv() 可以直接更新输入的data.table。在data.table里，所有的以set开头函数和操作符":="一样，它们都会更新输入的原数据。
  - 现在data.table已经按照origin列重新排序了。虽然是重新排序，但只需额外一列的内存空间。很节省内存。
  - 也可以在创建data.table的时候，调用函数data.table() 的参数key= 直接设置主键，参数key的值是列名的字符型向量。

设置 / 获取 / 使用主键（2/3）

- 一旦将某一列设置成data.table的主键，就可以在参数i里指定 **.()来subset**哪些主键了。
- 例：使用主键origin 来subset所有origin是”JFK”的行

  flights[**.("JFK")**]
- 说明：
  - 因为已经将主键设置为 origin列，所以只要直接指定"JFK"即可。.()用来在data.table的主键（也就是flights 的 origin列）里，查找"JFK"。
  - 如果主键是字符型的列，那么可以省略 .()，就像用行名subset一个data.frame的行的时候。flights["JFK"] 等同于 flights[.("JFK")]。
  - 我们可以根据需要指定多个值：flights[c("JFK", "LGA")] 等同于 flights[.(c("JFK", "LGA"))]，返回所有 origin列是“JFK” 或者“LGA”的所有行。

设置 / 获取 / 使用主键（3/3）

- 获得被设置为data.table的主键的那一列的列名，使用函数 key()。
- key(flights)

```
[1] "origin"
```

- 说明：
  - 函数 key() 返回主键列名的字符型向量。
  - 如果data.table没有设置过主键，返回 NULL。

主键和多个列（1/3）

- 例：将origin列和dest列都设置为主键。
- **setkey**(flights, origin, dest)
- 或：setkey**v**(flights, c("origin", "dest"))
- 说明：
  - 设置为主键的多个列可以是不同的类型。
  - data.table先按origin列排序，再按dest列排序。

主键和多个列（2/3）

- 例：subset所有满足条件 origin是“JFK”、dest是“MIA”的行。

  flights[.("JFK", "MIA")]

- 内部处理步骤：首先，用"JFK"和第一个主键 origin列匹配；然后，在匹配上的这些行里，用“MIA”和第二个主键 dest列匹配，这样来获取所有符合这两个条件的行的索引。

主键和多个列（3/3）

- 例：subset所有仅仅满足条件dest是“MIA”的行
- flights[.(unique(origin), "MIA")]
- 参考上页的处理步骤，首先必须通过unique(origin)获得origin所有可能的取值；之后“MIA”会被自动补足成跟unique(origin) 同样的长度，也就是3。

和参数j、参数by一起使用

在参数j里面select

- 例：返回符合 origin = “LGA” 和 dest = “TPA”这两个条件的 arr_delay列。

  flights[.("LGA", "TPA"), .(arr_delay)]
- 说明：
  - 通过基于主键的subset，我们获得了满足 origin == "LGA" 和 dest == “TPA”这两个条件的行索引。
  - 然后通过参数j请求arr_delay列。
  - 也可以通过指定 with = FALSE来select：

    flights[.("LGA", "TPA"), "arr_delay", with=FALSE]

Chaining表达式

- 将上页结果用chaining表达式按降序排列。

  flights[.("LGA", "TPA"), .(arr_delay)]**[order(-arr_delay)]**

在参数j里运算

- 例：找出符合 origin = “LGA” 和 dest = “TPA”这两个条件的航班的最大到达延误时间。

```
flights[.("LGA", "TPA"), max(arr_delay)]
```

在参数j里使用操作符“:=”来sub-assign

- 之前的例子：hour列有25种不同的取值。把24点全部替换成0点。现在使用主键做：

```
setkey(flights, hour)
key(flights) # 输出：[1] "hour"
flights[.(24), hour := 0L]
key(flights) # 输出：NULL
flights[, sort(unique(hour))]
```

- 说明：
  - 首先将 hour列设置为主键。这会将flights按照 hour列重新排序，并且将 hour列标记为主键。
  - 用 .()标记对hour列来subset。我们subset所有值为24的行的索引。对于这些行，我们将主键列的值替换为0。
  - 既然我们**替换了主键列的值**，flights也不再按照 hour列排序了。因此，**主键被自动去除了**，它被设置为NULL。

用参数by聚合

- 例：获取每个月从“JFK”起飞的航班的最大起飞延误时间，按月排序。

```
setkey(flights, origin, dest) #假设在操作之前，origin列和dest列已被设置为主键。
ans <- flights["JFK", max(dep_delay), keyby=month]
```

- 说明：
  - 对主键 origin列进行subset，得到了所有起飞机场是“JFK”的行索引。
  - 现在我们已经得到这些行的索引了，我们只需要两列—用来分组的month列，和用来计算每组最大值的dep_delay列。data.table的查询都被优化过了，因此在参数i取得的行的基础上，再subset这两列，效率和内存开销都很可观。
  - 在subset的时候，我们按month分组，再计算dep_delay列的最大值。
  - 我们使用参数**keyby**来自动将month设置为**结果**的主键。它使得结果**不仅按month列排序，而且将month设置为主键：**key(ans)

参数mult和nomatch

参数mult

- 对于每次查询，可以通过参数mult，指定所有符合条件的行“all”都被返回，还是只返回第一行“first”或者最后一行“last”。默认是所有的行“all”。
- 例：获取符合origin = “JFK”且 dest = “MIA”的数据的第一行。

```
flights[.("JFK", "MIA"), mult="first"]
```

- 获取符合origin = “LGA”或”JFK”或”EWR” 且 dest = “XNA”的数据的最后一行。

```
flights[.(c("LGA", "JFK", "EWR"), "XNA"), mult="last"]
```

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	cancelled	carrier	tailnum	flight	origin	dest	air_time	distance	hour	min
1:	2014	5	23	1803	163	2003	148	0	MQ	N515MQ	3553	LGA	XNA	158	1147	18	3
2:	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	JFK	XNA	NA	NA	NA	NA
3:	2014	2	3	1208	231	1516	268	0	EV	N14148	4419	EWR	XNA	184	1131	12	8

  - 注：JFK”、“XNA”不匹配flights的任何一条数据，因此返回NA。下页我们介绍如何把这行结果去掉。

参数nomatch

- 我们可以通过参数nomatch，指定在没有找到符合条件的数据的情况下，是返回NA呢，还是跳过（不返回）。
- 上页例：获取符合origin = “LGA”或”JFK”或”EWR” 且 dest = “XNA”的数据的最后一行。

```
flights[.(c("LGA", "JFK", "EWR"), "XNA"),
mult="last", nomatch = 0L]
```

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	cancelled	carrier	tailnum	flight	origin	dest	air_time	distance	hour	min
1:	2014	5	23	1803	163	2003	148	0	MQ	N515MQ	3553	LGA	XNA	158	1147	18	3
2:	2014	2	3	1208	231	1516	268	0	EV	N14148	4419	EWR	XNA	184	1131	12	8

二分法搜索 vs 向量扫描

准备数据集

- 创建一个有两千万行、三列的样本数据，将它的主键设置为x列和y列。

```
set.seed(2L)
N = 2e7L
DT = data.table(x = sample(letters, N, TRUE), y = sample(1000L, N, TRUE), val=runif(N), key = c("x", "y"))
print(object.size(DT), units="Mb")
# 381.5 Mb。不算特别大。
dim(DT)
#[1] 20000000        3
key(DT)
# [1] "x" "y"
```

使用“向量扫描”的方法

- 任务：subset 那些 x = “g” 和 y = 877 的行。

```
t1 <- system.time(ans1 <- DT[x == "g" & y == 877L])
```

```
   用户  系统  流逝
   1.12  0.08  1.20
```

```
head(ans1)
dim(ans1)
# [1] 761   3
```

使用主键：二分法搜索

- t2 <- system.time(ans2 <- DT[.("g", 877L)])
- t2

  用户 系统 流逝
  0 0 0
- head(ans2)
- dim(ans2)
- # [1] 761 3

为什么用主键subset能这么快（1/2）

- 第一种方法：向量扫描
  - 在所有两千万条数据中，逐行搜索 x列里值为“g”的行。这会生成一个有两千行的逻辑向量，根据和x列的批评结果，它每个元素的取值可能是TRUE, FALSE 以及 NA。
  - 相似的，在所有两千万条数据中，逐行搜索 y列里值为“877”的行，再保存在另一个逻辑向量里面。
  - 操作符"&"对上面两个逻辑向量进行“且”运算，返回结果为TRUE的行。
  - 这就是所谓的“向量扫描”。效率非常低，特别是数据量很大、需要重复subset的时候。因为它每次不得不对整个数据全盘扫描。

为什么用主键subset能这么快（2/2）

- 第二种方法：二分法搜索
  - data.table根据主键列进行了排序。既然数据被排序了，我们就不需要再对整个数据进行扫描。我们用二分法搜索的时间开销是 O(log n)，而向量扫描的时间开销是 O(n)，其中n是data.table的行数。
  - 因为data.table的行在内存中是连续存储的，这种subset的操作也很节省缓存，这有利于处理速度。