An R Interface to Hail

Michael Lawrence (Genentech)

May 31, 2019
A platform and language for distributed genomics on Apache Spark

Provides:
- General tabular data manipulation,
- Specialized, scalable data structures stored in Parquet, and
- Efficient implementation of domain-specific algorithms in C++

Designed to be interfaced through high-level languages
- Python interface comes out of the box
Bioconductor is an R platform for integrative genomics

- Started 2002
- Led by Martin Morgan
- Core infrastructure maintained by about 8 people, based in Roswell Park CRC in Buffalo, NY
- >2000 R software packages that form a unified platform
- Well-used and respected.
  - 53k unique IP downloads / month.
  - 21,700 PubMedCentral citations.
Bioconductor is an integrated platform
Hail and Bioconductor share their central data type

A SummarizedExperiment (Bioc) or MatrixTable (Hail)

```
se <- SummarizedExperiment(
    assays,
    rowData,
    colData,
    exptData
)
```
The hailr package
hailr pushes compute to Hail via *lazy evaluation*

\[ \texttt{> head(sort(x))} \]

Eager evaluation

Lazy evaluation

- \texttt{sort(x)}
- \texttt{Sorted full dataset}
- \texttt{sorted(x)}
- \texttt{Sorted head}
- \texttt{head(sort(x))}
- \texttt{Sorted full dataset}
- \texttt{R}
- \texttt{R}
R already uses lazy evaluation

```r
plot(1:10)
```
R function call arguments are promises

```r
fun <- function(arg) substitute(arg)
fun(1:10)
```

1:10
Lazy evaluation delays work until absolutely necessary

```r
fun <- function(arg) {
    z <- arg
    substitute(z)
}
fun(1:10)

[1] 1 2 3 4 5 6 7 8 9 10
```
It often pays to be lazy

- Provenance capture
- Interactive graphics pipelines, streaming
- Responsiveness through asynchronicity
- Optimization
- Compact representations
- Querying databases and files
- Distributed computing
It often pays to be lazy

- Provenance capture
- Interactive graphics pipelines, streaming
- Responsiveness through asynchronicity
- Optimization
- Compact representations
- Querying databases and files
- Distributed computing
It often pays to be lazy

- Provenance capture
- Interactive graphics pipelines, streaming
- Responsiveness through asynchronicity
- Optimization
- Compact representations
- Querying databases and files
- Distributed computing
It often pays to be lazy

- Provenance capture
- Interactive graphics pipelines, streaming
- Responsiveness through asynchronicity
- **Optimization**
- Compact representations
- Querying databases and files
- Distributed computing
It often pays to be lazy

- Provenance capture
- Interactive graphics pipelines, streaming
- Responsiveness through asynchronicity
- Optimization
- Compact representations
- Querying databases and files
- Distributed computing

New in R 3.5:

```r
system.time(1:1e12)
```

```
user  system elapsed
0      0       0
```

CompressedList in S4Vectors:

```r
splitAsList(1:8, rep(1:4, c(3, 2, 1, 2))
```

```
3 5 6 8
```
It often pays to be lazy

- Provenance capture
- Interactive graphics pipelines, streaming
- Responsiveness through asynchronicity
- Optimization
- Compact representations
- Querying databases and files
- Distributed computing
It often pays to be lazy

- Provenance capture
- Interactive graphics pipelines, streaming
- Responsiveness through asynchronicity
- Optimization
- Compact representations
- Querying databases and files
- Distributed computing
Deferred data structures allow for eager evaluation

For some promise “x”:

> head(sort(x))
Deferred data structures allow for eager evaluation

For some promise “x”:

\[
\text{head}(	ext{sort}(x))
\]
Deferred data structures allow for eager evaluation

For some promise “\(x\)“:

\[ \text{head(sort(x))} \]

Diagram:
- \(\text{sort()}\)
- \(\text{head()}\)
- \(\text{Dispatch (S3/S4)}\)
- \(\text{sort,Promise()}\)
- \(\text{head,Promise()}\)
- \(\text{sort}\)
- \(\text{limit}\)
- \(\text{get}\)
- \(\text{x}\)
Deferred data structures allow for eager evaluation

For some promise “x”:

\[ \text{head}(\text{sort}(x)) \]

```
sort() > head(sort(x))

Dispatch (S3/S4)

sort, Promise()  head, Promise()

sort
get
limit
x
Serialization
(limit (sort (get x)))
```
Deferred data structures allow for eager evaluation

For some promise “x”:

> head(sort(x))

Programmability is fundamental

DSLs should rest on programmatic API:

> df %>% arrange(x) %>% head()

DSL implementation

Base

sort() \quad \text{sort,Promise()}

head() \quad \text{head,Promise()}

Dispatch (S3/S4)
Bioconductor containers rely on base API to be generic

- Bioconductor containers assume elements implement key functions from the base API
  - *DataFrame* allows anything "vector-like" to be a column
  - *SummarizedExperiment* allows anything "matrix-like" to hold assay values
- Since our promises implement the base API, they just work
- But we still want to map DataFrame operations to Hail Table operations
hailr is a hierarchical extension of Bioconductor
hailr is a hierarchical extension of Bioconductor
We can load data into Hail

Directly from a text file:

```r
library(hailr)
data_dir <- system.file("extdata", package="hailr")
.tsv1 <- file.path(data_dir, "kt_example1.tsv")
df <- readHailDataFrameFromText(tsv1, header=TRUE)
```

Copying from an R data.frame:

```r
df <- copy(read.table(tsv1, header=TRUE), hail())
```
...and get it back out

```r
df

HailDataFrame with 4 rows and 8 columns

<table>
<thead>
<tr>
<th>ID</th>
<th>HT</th>
<th>SEX</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>65</td>
<td>M</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>72</td>
<td>M</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>70</td>
<td>F</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>60</td>
<td>F</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>81</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>90</td>
</tr>
</tbody>
</table>

df$ID

[1] 1 2 3 4
```
A glimpse into the compiler

```r
as.character(df$ID@expr)

[1] "(GetField ID (Ref row))"
```
Abstractions enable mixed evaluation
Abstractions enable mixed evaluation
Looking forward: generalized, integrated compute
Intermediate algebra, optimization with backend-informed cost model

For some promise “x”:

```plaintext
> head(sort(x))
```

```
sort()
head()
```

```
Dispatch (S3/S4)
sort Promise() head Promise()
```

```
sort
head
x
```
Looking forward: generalized, integrated compute
Intermediate algebra, optimization with backend-informed cost model

For some promise “x”:

\[ \text{sort}(x)[1:5] \]

Diagram:

- sort()
- head()
- Dispatch (S3/S4)
- sort,Promise()
- head,Promise()
- x
Looking forward: generalized, integrated compute  
Intermediate algebra, optimization with backend-informed cost model

For some promise “$x$”:

```plaintext
> head(sort(x))
```

```
sort()
head()
```

**Dispatch (S3/S4)**

```
sort,Promise()
head,Promise()
sort
head
x
Optimization
Partial 
sort`
Looking forward: generalized, integrated compute
Intermediate algebra, optimization with backend-informed cost model

For some promise “x”:

\[
> \text{head(sort(x))}
\]
Looking forward: generalized, integrated compute
Intermediate algebra, optimization with backend-informed cost model

For some promise “x”:

> head(sort(x))

```
sort()
head()
```

Dispatch (S3/S4)

```
sort, Promise()
head, Promise()
sort
head
```

Optimization
Partial 
sort
Translation
Serialization
solr hail SQL
Acknowledgements

- Javier Luraschi and Kevin Kuo (responsive sparklyr support)
- Samuel Macêdo (recent contributions)
- Cotton Seed (leader of Hail project)