Memory Management in TIBCO® Enterprise Runtime for R

Michael Sannella
msannell@tibco.com
UseR!2013
What is TERR?

- **TERR**: TIBCO® Enterprise Runtime for R
- A new R-compatible statistics engine.
- The latest in a family of scripting engines.  
  - S, S-PLUS®, R, TERR
- Recently released with Spotfire® 5.0.  
  - Used to implement “predictive analytic tools”
- Free Developer's Edition available at:  
  www.TIBCOmmunity.com/community/products/analytics/terr
- Commercially available for Spotfire and custom integration.
Building TERR

- We rebuilt the engine internals from scratch.
  - Redesigned data object representation
  - Redesigned memory management facilities
  - Attempted to address long-standing problems with S, S-PLUS®, R

- This talk will compare R and TERR designs for:
  - Data representation
  - Reference counting
  - Garbage collection
R: Data Representation

- Basic R data objects have a fixed representation in memory.
  - The contents of a vector is represented as a C array
  - Data must be in memory
  - Only one representation for a given type

- Example: An R numeric vector is represented as an object containing a C `double*` pointer to an array in memory.

- Can access object elements via simple memory access operations.
  - If `USE_RINTERNALS` is defined, `REAL(pObj)` will be compiled as a simple memory access
TERR: Data Representation

- TERR implements data objects via abstract C++ classes with one or more representations.
  - Manipulate object via C++ methods
  - Doesn’t support direct access
  - USE_RINTERNALS is not supported

- Scalar vs small vector vs big vector
  - Different representations with different length fields
  - Matrix of basic types and representations implemented via C++ templates
**TERR: Data Representation**

- **Sequence objects: 1:1e6**
  
  ```r
  > x <- 1:1e6
  > object.size(x)
  104 bytes
  > x[1] <- 0
  > object.size(x)
  8000088 bytes
  ```

- **String sequences: as.character(1:1e6)**
  
  Used for `data.frame` row names

- **Two representations for logical vectors:**
  
  - Normal logical vectors have 1-byte elements
  - C code `NEW_LOGICAL(n)` creates logical vectors with 4-byte elements
TERR: Potential Future Data Representation

- Extend preliminary support for out-of-memory vectors.
  - Transparent “bigdata” objects
  - Only use external files when necessary
- Access shared memory between applications.
  - Example: Send data between Spotfire and TERR
- Access database tables.
- Access streaming data.
- Represent deferred evaluation “futures”
  - Example: An object representing the result of adding two vectors B and C, only performing the addition when needed
  - See: Riposte runtime, Talbot et al., UseR2012
Reference Counts

• R and TERR use reference counts to efficiently implement object modification semantics.
• If an object has only one reference, we can modify it without copying it.

```r
x <- rnorm(1e6)
x[1] <- 0 # only one reference: modify in place
y <- x
x[2] <- 0 # two references: copy before modify
```
Reference Count Field

• **R objects contain 2-bit reference count**
  – “named” field
  – Easy to make examples that set reference count to “max” value that will never be decreased
  – Max reference count → copy on change

• **TERR objects contain 16-bit reference counts**
  – Tracks number of references up and down
  – Identify more objects that can be modified in place, without copying
refcnt.0 <- function(n) {
    z <- as.numeric(1:n)
    for(x in 1:n) {
        # only one reference to vector
        # so no copy is done
        z[x] <- z[x]+1
    }
    z
}
Two References

refcnt.1 <- function(n) {
  z <- as.numeric(1:n)
  for(x in 1:n) {
    z2 <- z       # temporarily add
    z2 <- NULL    # second reference
    z[x] <- z[x]+1
  }
  z
}

refcnt.2 <- function(n) {
    z <- as.numeric(1:n)
    fn <- function(vec, idx) vec[idx] + 1
    for(x in 1:n) {
        # temporarily add second reference
        # while calling function
        z[x] <- fn(z, x)
    }
    z
}
Reference Count Tests: Performance

The graph shows the performance of reference count tests with different n values. The x-axis represents the value of n ranging from 0 to 100000, and the y-axis represents the time in seconds ranging from 0 to 40.

Key observations:
- R-3.0.1 refcnt.1
- R-3.0.1 refcnt.2
- TERR refcnt.0
- TERR refcnt.1
- TERR refcnt.2

The performance trend indicates an increase in time as the value of n increases.
Reference Count Tests: Performance
R: Garbage Collection

- R uses a mark-and-sweep garbage collector to reclaim storage for unused objects.
- The R garbage collector is a generational garbage collector.
  - Quick sweep through “young” objects to reclaim temporary objects
TERR: Garbage Collection

- TERR uses a mark-and-sweep garbage collector along with reference counts to reclaim storage.
  - When an object reference count goes to zero, can immediately reclaim temporary object
  - Calls full mark-and-sweep garbage collection (infrequently)
- After engine initialization:
  - Set “static” objects to have the max reference count
  - Don’t scan on future garbage collections
Suppose we have a SAS® “data step” with complex logic for processing records one row at a time.

Suppose we hand-translate each line directly into R code.

– Of course, this is not going to be efficient, since R is not suited to row-by-row operations

– However, the user may not want to entirely recode in a vectorized form, distorting the original logic

Question: How fast will this “non-optimized” code run in R and TERR?
```sas
... ELSE IF event='complaint' AND startDate NE . THEN DO;
  lastComplaintDate = date;
  weeknum = 1+(date-startDate)/(7*24*60*60);
  IF weeknum<2 THEN DO;
    comp1 = comp1+1;
  END;
  ELSE IF weeknum<3 THEN DO;
    comp2 = comp2+1;
  END;
  ELSE DO;
    compN = compN+1;
  END;
...```
... 

} else if (df.event[i]=="complaint" && !is.na(startDate)) {
    lastComplaintDate <- df.date[i]
    weeknum <- 1 + as.numeric(difftime(df.date[i],
                                        startDate, units="days"))/7
    if (weeknum<2) {
        comp1 <- comp1+1
    } else if (weeknum<3) {
        comp2 <- comp2+1
    } else {
        compN <- compN+1
    }
...
Row-By-Row Test: Performance

The graph shows the relationship between the number of rows and the time taken in seconds. As the number of rows increases, the time taken increases significantly. The graph includes data from R 3.0.1 and TERR.
Row-By-Row Test: Performance

![Graph showing the performance of R 3.0.1 and TERR with increasing number of rows.](image)
Row-By-Row Test: Garbage Collection

• R-3.0.1:
  # before processing 1M rows
  Garbage collection 67 = 26+6+35 (level 2)
  # after processing 1M rows
  Garbage collection 40300 = 30966+7745+1589 (level 2)

• TERR:
  # before processing 1M rows
  40 collections
  # after processing 1M rows
  42 collections
• We strongly suspect that the R performance problem is due to frequent garbage collections to reclaim temporary objects.

• In TERR, most temporary objects are immediately reclaimed using reference counts.
Conclusion

• TERR internals were redesigned from scratch to support better memory management and performance.
• TERR internal data representation supports further experiments with more efficient objects.
• Full reference counting reduces unnecessary copies, and improves garbage collection performance.