Growable Vectors and In-Place Row Operations

A data.table Perspective

Benjamin Schwendinger

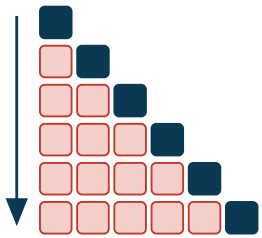
University of Applied Sciences Kufstein

Fraunhofer Austria Research GmbH

useR!2026 · July 8, 2026

The append problem

```
x = integer(0)
for (i in 1:1e5) x = c(x, i)   # quadratic: O(n^2) copies
```



new element

allocated + copied, again

n appends

1 + 2 + … + n copies

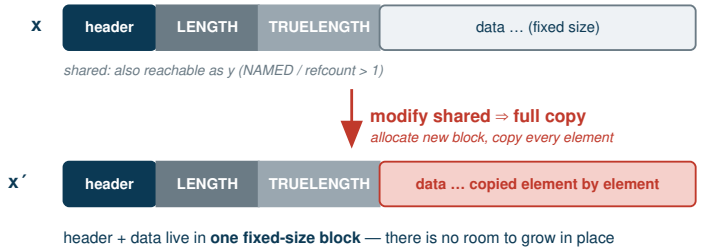
= O(n²)

Every `c()`, every `rbind()` in a loop: **allocate new, copy all, free old**

Measured: **~50× slower** at 32 768 appends — doubling n doubles the gap

Why: copy-on-modify + fixed-size allocations

- R vectors are allocated with a **fixed length**, header + data in one block
- Modifying a shared object ⇒ copy first (`NAMED` / reference counting)
- No public way to say *“give me room to grow”* … until now

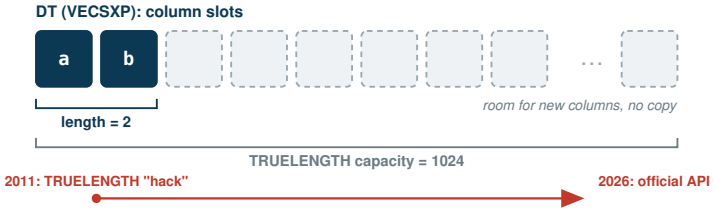


The old workaround: over-allocation

Since ~2011 (1.7.3), `data.table` has over-allocated the **column list** using TRUELENGTH — that's why `DT[, newcol := ...]` never copies the table.

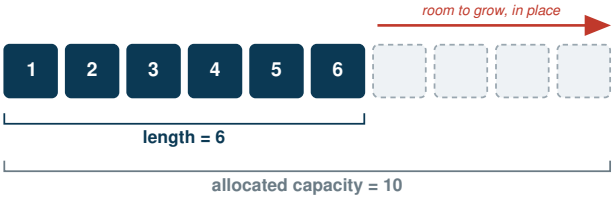
```
DT = data.table(a = 1:5)
truelength(DT)   # 1024+ column slots, length 1
```

Caveat: it worked, but it used **internal, undocumented** fields — a hack R-core tolerated until the recent tightening of non-API usage on CRAN.



New in R: the resizable vector API

```
SEXP  R_allocResizableVector(SEXPTYPE type, R_xlen_t maxlen);
void  R_resizeVector(SEXP x, R_xlen_t newlen);   // up to maxlen, in place
bool R_isResizable(SEXP x);
R_xlen_t R_maxLength(SEXP x);
SEXP R_duplicateAsResizable(SEXP x);
```



Allocate with **capacity** but resize freely within it **no reallocation, no copy**

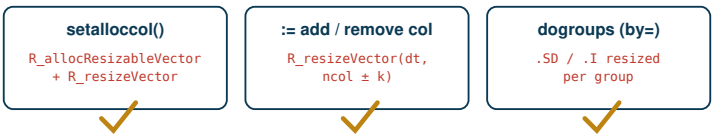
What this buys you

Pattern	Before	With capacity
Append n rows	$O(n^2)$ copies (or manual doubling + hack)	**amortized O(1)** per append
Shrink / delete	full copy of what's kept	**in-place compaction**
Grow column list	`TRUELENGTH` hack	official, checkable (`R_isResizable`)

data.table has already migrated

The internals now call the official API where the hack used to be:

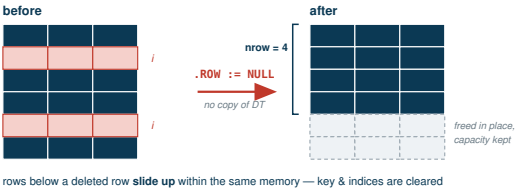
- `setalloccol()` → `R_allocResizableVector` + `R_resizeVector` for the column list
- `:=` add/remove column → `R_resizeVector(dt, ncol±k)` — *removing a column is just shrinking*
- `dogroups` (by=): one **capacity-`maxGrpSize`** `.SD`/`.I`, resized per group instead of reallocated per group



New feature: delete rows by reference

```
DT[speed > 100, .ROW := NULL]       # rows gone, no copy of DT
```

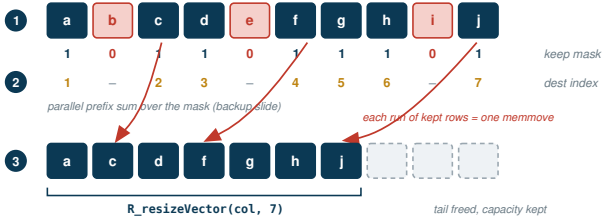
- Feature request **#635** — open since 2014, one of the most upvoted issues
- Shipped in current dev version (1.18.99 and will probably be in 1.18.6)
- `i` selects the victims; by/keyby not allowed; key & indices are cleared



How it works: stream compaction, in place

Three passes over each column, all within existing memory:

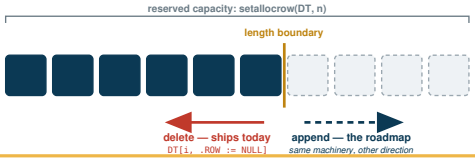
1. Build a **keep mask** from `i`
2. **Prefix sum** over the mask → destination index for every kept row
3. `memmove` **runs of kept rows** left → `R_resizeVector(col, new_nrow)`



The next logical step: row *growth*

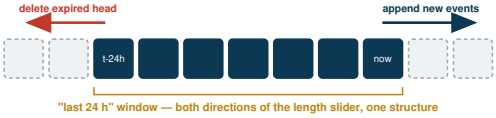
```
setallocrow(DT, 2e6)   # ships today: pre-allocate row capacity
DT[.ROW := new_rows]   # NOT YET — but the foundation is now in place
```

- Today: **fast insertion of rows does not exist** — the idiom is “collect in a list, `rbindlist` once”
- `setallocrow()` already reserves capacity; columns are resizable → append = move the length slider **right**
- With geometric reserve, appends become **amortized O(1)** — supported by R itself, not simulated
- Delete + append reusing the same memory: a data.table as a true **dynamic array**



What cheap appends unlock

- **Streaming ingestion**: sensor data, ticks, logs. Append events as they arrive, no batch ceremony
- **Sliding windows**: append at the tail + delete at the head = ring buffer (“last 24h of events”) in one structure
- **Simulation & agent-based models**: results tables that grow across iterations without O(n^2) blowup
- **Incremental construction**: build a table row-wise when the size genuinely isn’t known upfront



Numbers: bulk deletion

Benchmark: delete 1% / 50% of rows from a 10M-row table (~240 MB), medians of 3 runs

method	time (1% / 50%)	R allocations (1% / 50%)
`DT = DT[!i]` (copy)	0.08 s / 0.10 s	274 MB / 162 MB — ≈ the kept rows
`DT[i, .ROW := NULL]`	0.09 s / 0.19 s	**76 MB / 76 MB — constant, indices only**

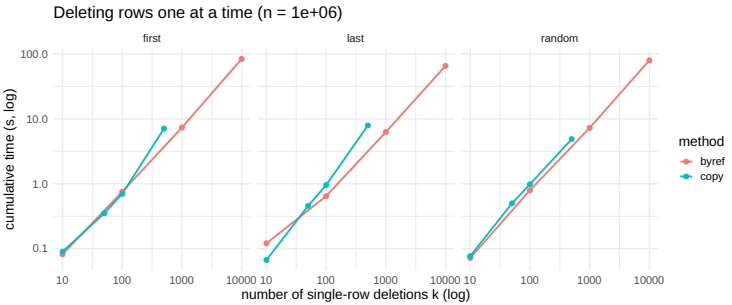
Copy scales with the **table**; by-reference scales with **nothing**.

A 20 GB table still needs only its index vector!

Numbers: one row at a time

```
for (k in 1:10000) DT = DT[-sample(nrow(DT), 1)]          # copy: O(n) each
for (k in 1:10000) DT[sample(nrow(DT), 1), .ROW := NULL]  # in place
```

- Copy: every single deletion reallocates and copies the **whole table**
- By reference: `memmove`s within existing memory



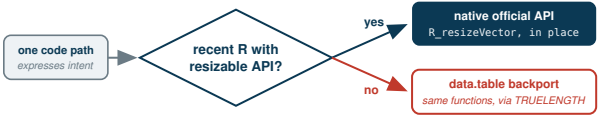
Constraints: sharing, GC, invariants

- **Sharing / NAMED**: resizing a vector someone else references would change *their* length → data.table's selfref discipline guards this
- **Garbage collection**: capacity is committed until the vector dies — `gc()` can't reclaim the unused tail; character/list compaction goes through `SET_STRING_ELT`/`SET_VECTOR_ELT` to keep the **write barrier** honest (no `memmove` for those)
- **ALTREP**: compact sequences etc. aren't resizable → materialize/copy once (`R_isResizable` check), then in-place forever
- **Serialization**: `saveRDS`/`load` drop capacity → a restored table needs `setDT()`/`setallocrow()` again
- **Attribute invariants**: `row.names` rebuilt to match the new length;

keys & indices invalidated by deletion → cleared, by design

Cross-version compatibility

- The API is **experimental** and only in recent R
- data.table **backports** it: on older R, the same functions are provided on top of the historic TRUELENGTH machinery
- **One code path**, identical behavior — `.ROW := NULL` & friends work on every supported R version



Takeaways

1. R now has an **official capacity concept** — growth and shrinkage without reallocation
2. It **formalizes**, not changes, semantics: same rules, safer foundations
3. data.table uses it end-to-end: columns, grouping buffers, and **`DT[i, .ROW := NULL]`**
4. Fast row **insertion** doesn't exist yet — but it's the next logical step, and the foundation now ships
5. Pattern is portable: any package building dynamic structures in C can adopt it

Thanks / pointers

- Thanks to Luke Tierney for introducing the resizable vector API
- This work was supported by the UKRI Medical Research Council (UKRI1573) through the project “Maintaining and expanding the data.table R package for fast and efficient data manipulation”

- Writing R Extensions → *System and foreign language interfaces* → *Resizing vectors*
- Try it: `DT[i, .ROW := NULL]`

Questions?

Backup — The prefix sum is parallel

Two-pass exclusive scan (OpenMP), same trick as in radix sort:

1. Each thread counts kept rows in its chunk
2. Sequential scan over per-thread counts → offsets
3. Each thread writes its local prefix sum starting at its offset