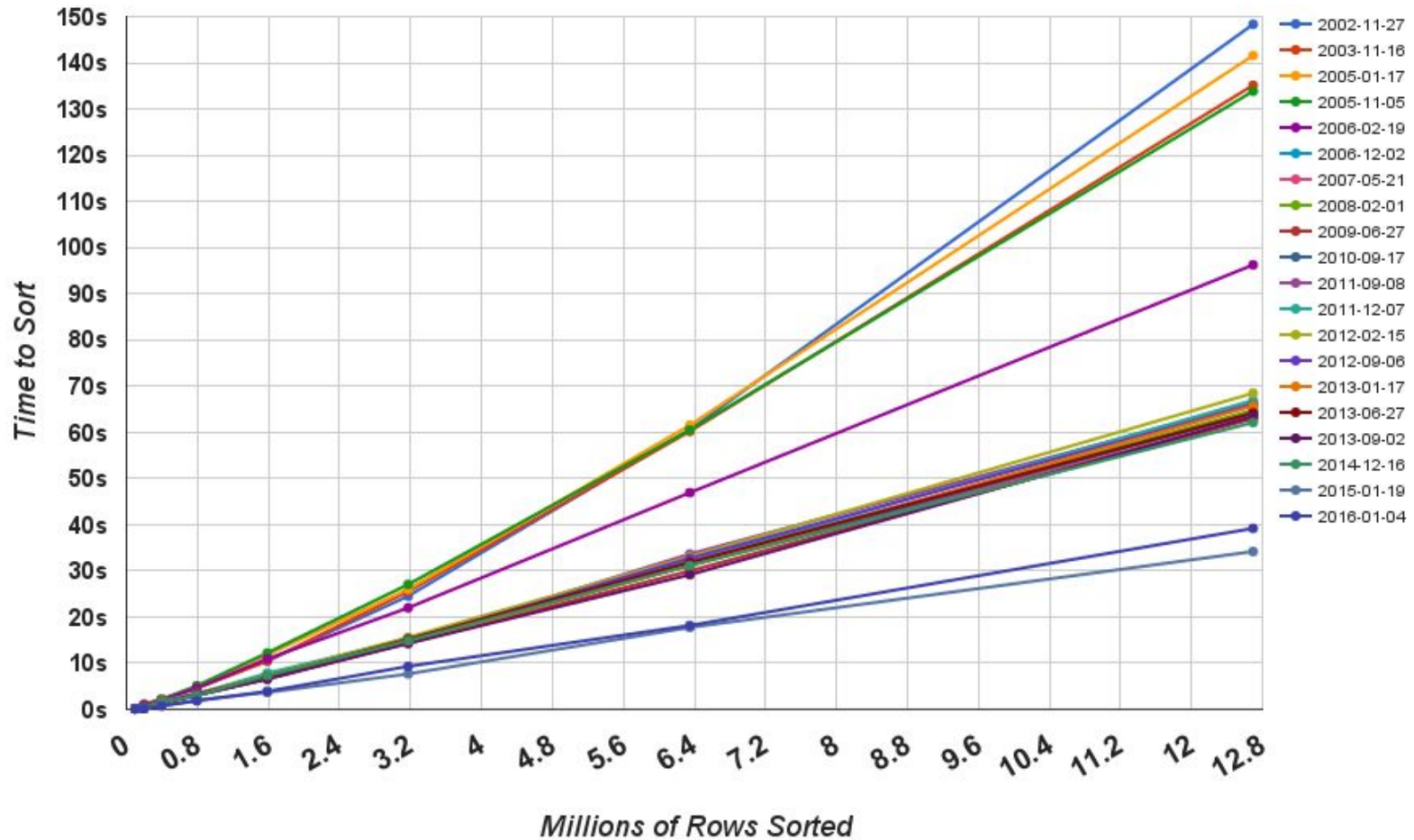
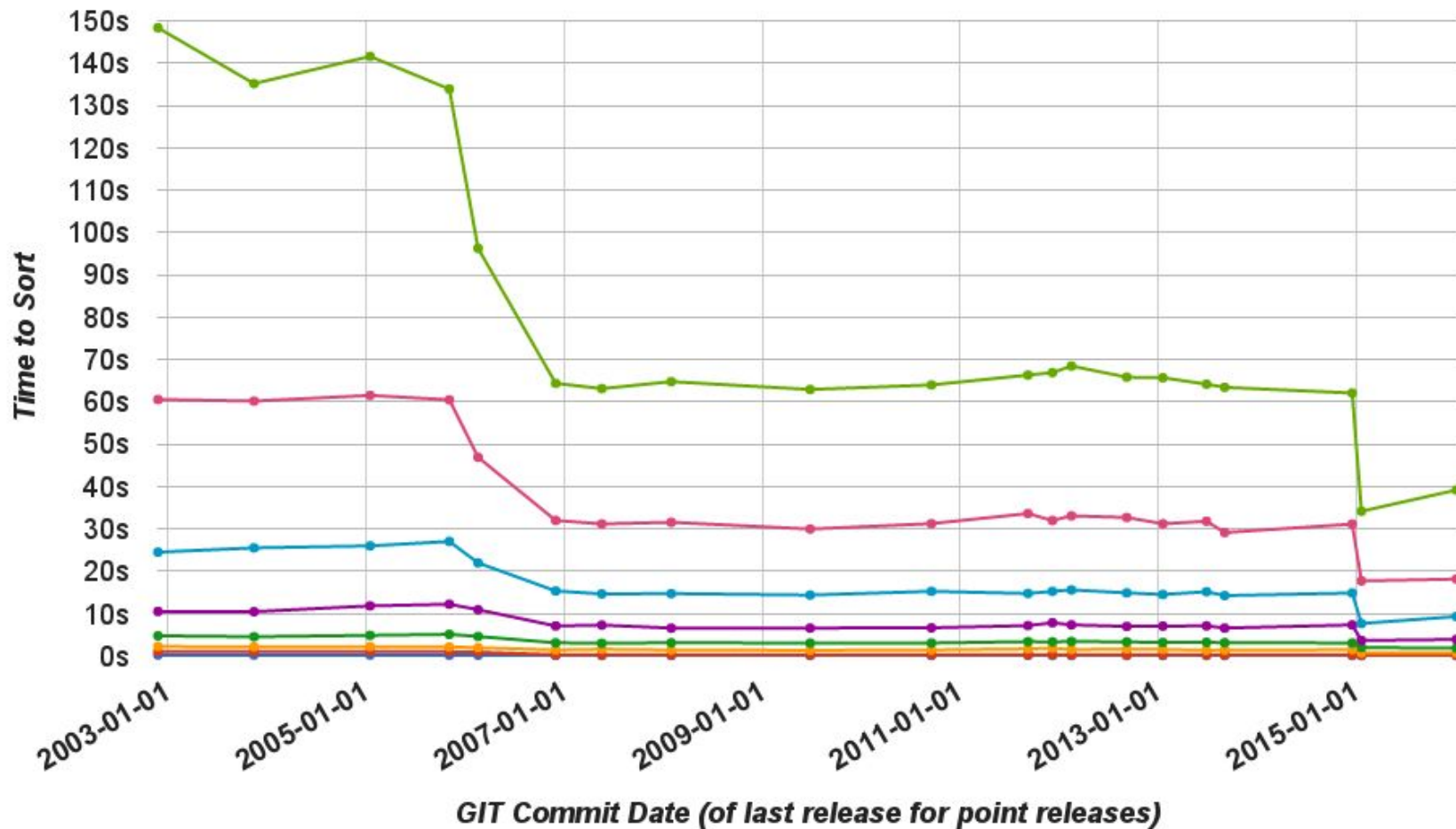


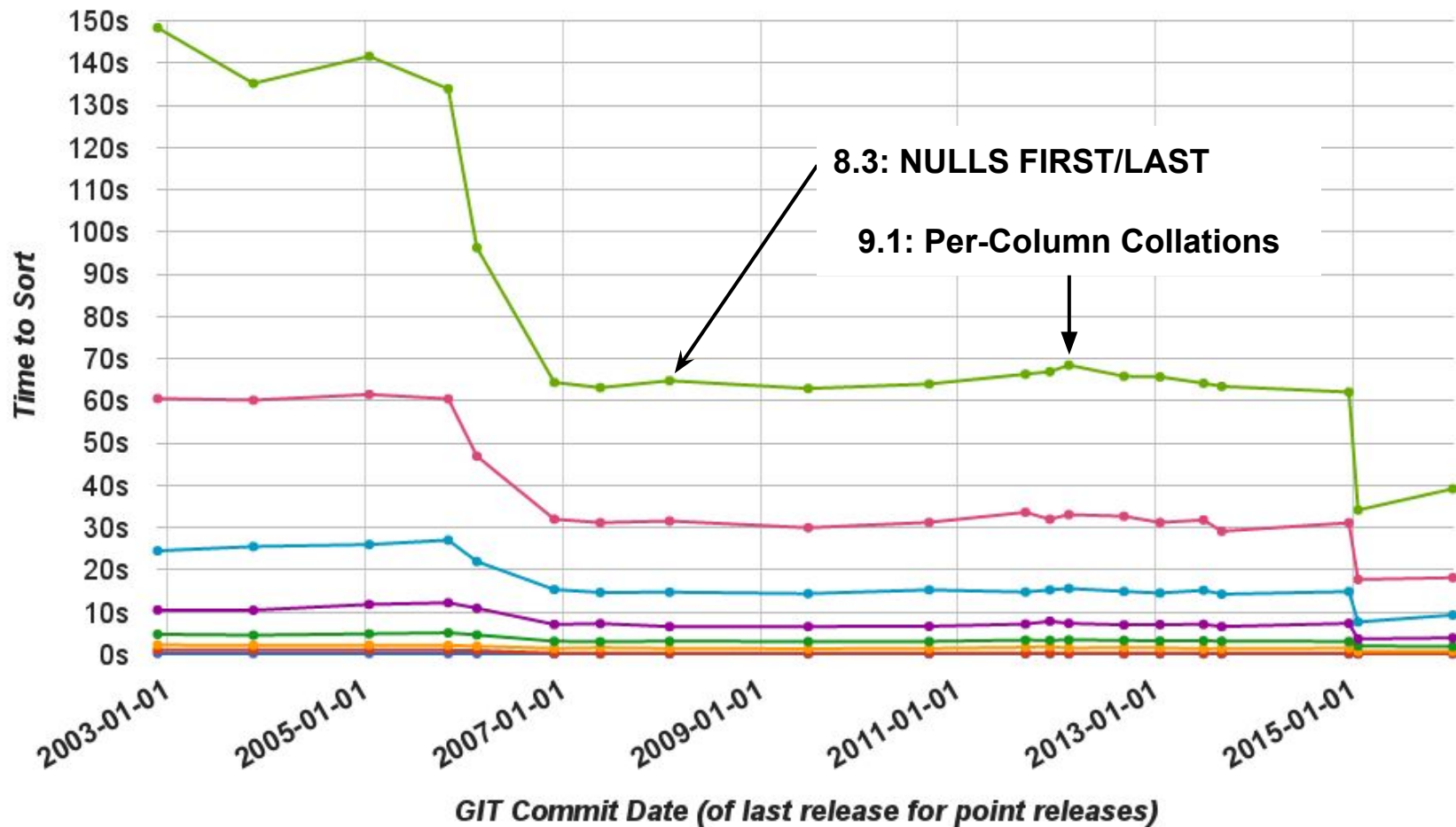
# Sorting Through The Ages



# Sorting Through The Ages



# Sorting Through The Ages

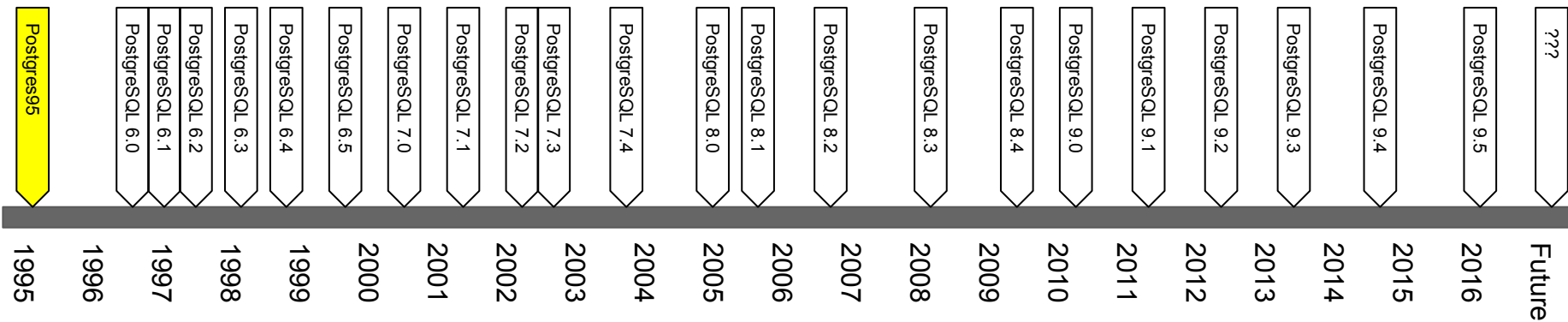


# Postgres95 - First public release



Sort code in psort.c is 618 lines long.

It implements only one sort algorithm, an external (on-disk) algorithm out of Knuth called Replacement Selection.



# PostgreSQL 6.2 - Use quicksort when data fits in memory

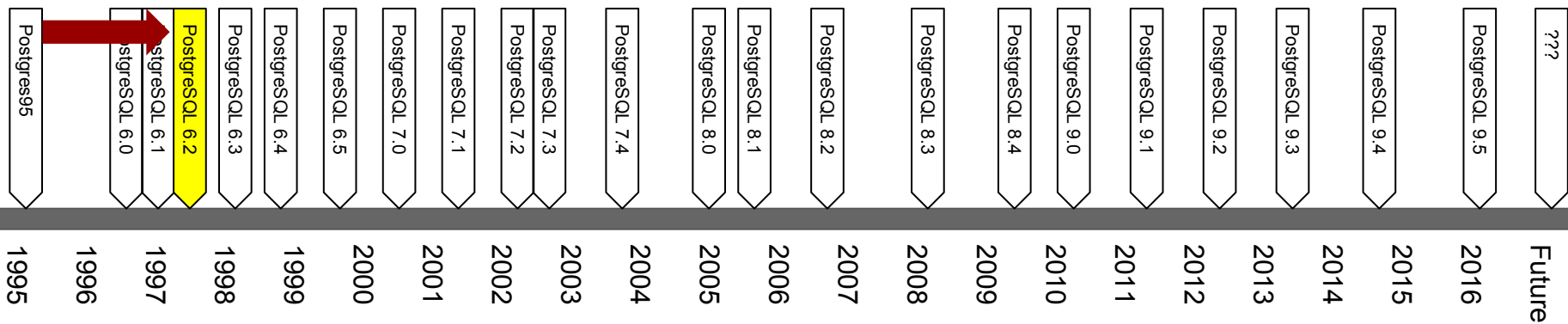


commit 712ea2507ef7f3ea4a7149962c85de0b35245a64

Author: Vadim B. Mikheev <vadim4o@yahoo.com>

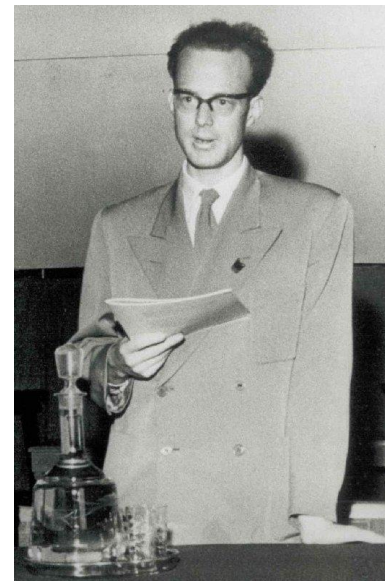
Date: Thu Sep 18 05:37:31 1997 +0000

1. Use qsort for first run
2. Limit number of tuples in leftist trees:
  - put one tuple from current tree to disk if limit reached;
  - end run creation if limit reached by nextrun.
3. Avoid mergeruns() if first run is single one!



## A quick digression -- Quicksort

- ❖ Invented in 1959 by Tony Hoare
- ❖ In-memory only algorithm
- ❖  $n \cdot \log(n)$  average run-time
- ❖ Very low constant factor
- ❖ Very efficient use of CPU cache and registers
- ❖ “Cache-oblivious”
  - Does not depend on tuning for specific cache sizes
- ❖ Available in standard C library
- ❖ Has  $O(n^2)$  worst-case (but unlikely)
  - Worse cases tend to happen in cases like partially sorted data or “organ-pipe” data (increasing then decreasing).
  - Various strategies to mitigate -- choice of pivot, randomizing inputs



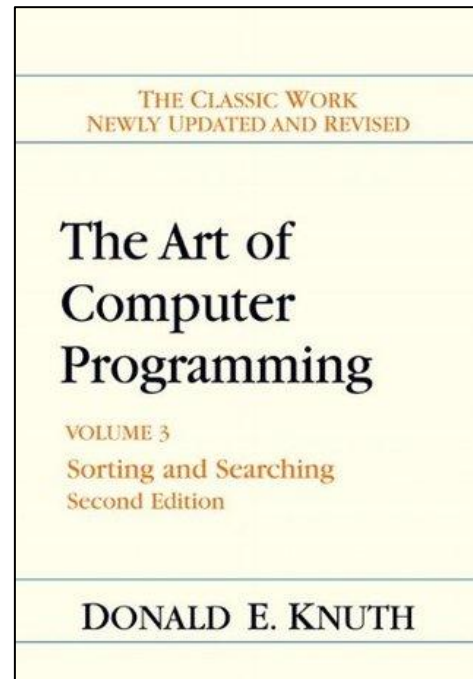
Tony Hoare

## A longer digression -- Replacement Selection

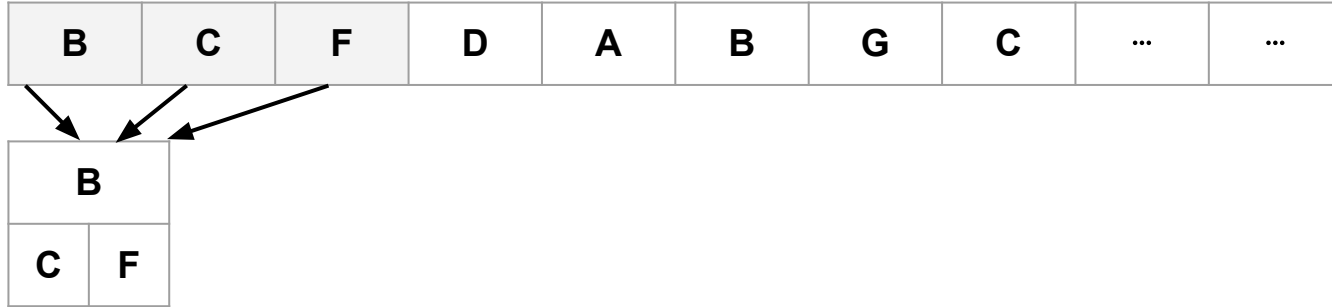
Postgres implements an external sort using Replacement Selection with Polyphase Merge straight out of Knuth first published in 1973.

- ❖ External (on-disk) sort which works with a limited subset of the data in-memory at any one time
- ❖ Writes data out to disk in sorted runs then reads it back into memory to merge into longer runs
- ❖ Repeats the merge process until only one tape is left
- ❖ Replacement selection (R1-R3)--Knuth, Vol.3, p.257
- ❖ Polyphase merge Alg.D (D1-D6)--Knuth, Vol.3, p.270-271

It always generates files on disk for temporary runs and always generates a new file on disk for the resulting sorted relation.



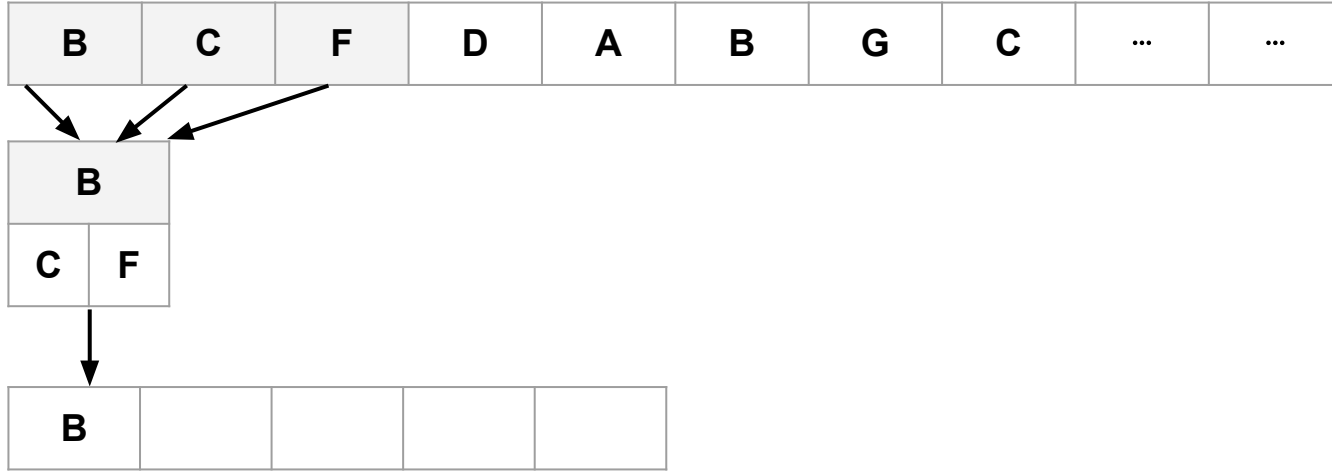
## Replacement Selection: Generate Runs



- ❖ Load as many values as possible in working memory (work\_mem)
- ❖ Build a priority queue using a heap



# Replacement Selection: Generate Runs



- ❖ Load as many values as possible in working memory (work\_mem)
- ❖ Build a priority queue using a heap
- ❖ Extract Lowest value and output to a sorted “run”

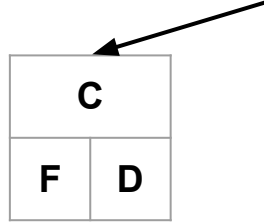
# Replacement Selection: Generate Runs

|          |          |          |          |          |          |          |          |     |     |
|----------|----------|----------|----------|----------|----------|----------|----------|-----|-----|
| <b>B</b> | <b>C</b> | <b>F</b> | <b>D</b> | <b>A</b> | <b>B</b> | <b>G</b> | <b>C</b> | ... | ... |
|----------|----------|----------|----------|----------|----------|----------|----------|-----|-----|

|          |  |
|----------|--|
| <b>C</b> |  |
| <b>F</b> |  |

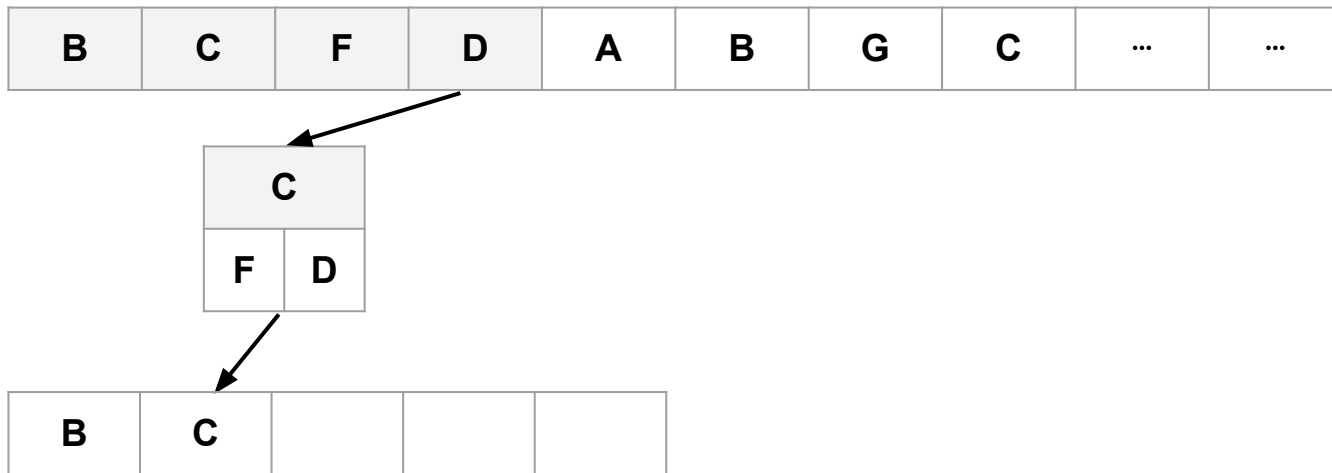
|          |  |  |  |  |
|----------|--|--|--|--|
| <b>B</b> |  |  |  |  |
|----------|--|--|--|--|

## Replacement Selection: Generate Runs



- ❖ Load next value from input stream to replace value that was output to run

## Replacement Selection: Generate Runs



- ❖ Load next value from input stream to replace value that was output to run
- ❖ Output new lowest value to sorted run

## Replacement Selection: Generate Runs

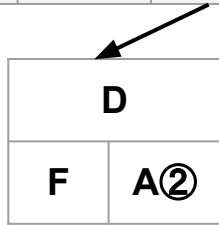
|          |          |          |          |          |          |          |          |     |     |
|----------|----------|----------|----------|----------|----------|----------|----------|-----|-----|
| <b>B</b> | <b>C</b> | <b>F</b> | <b>D</b> | <b>A</b> | <b>B</b> | <b>G</b> | <b>C</b> | ... | ... |
|----------|----------|----------|----------|----------|----------|----------|----------|-----|-----|

|          |  |
|----------|--|
| <b>D</b> |  |
| <b>F</b> |  |

|          |          |  |  |  |
|----------|----------|--|--|--|
| <b>B</b> | <b>C</b> |  |  |  |
|----------|----------|--|--|--|

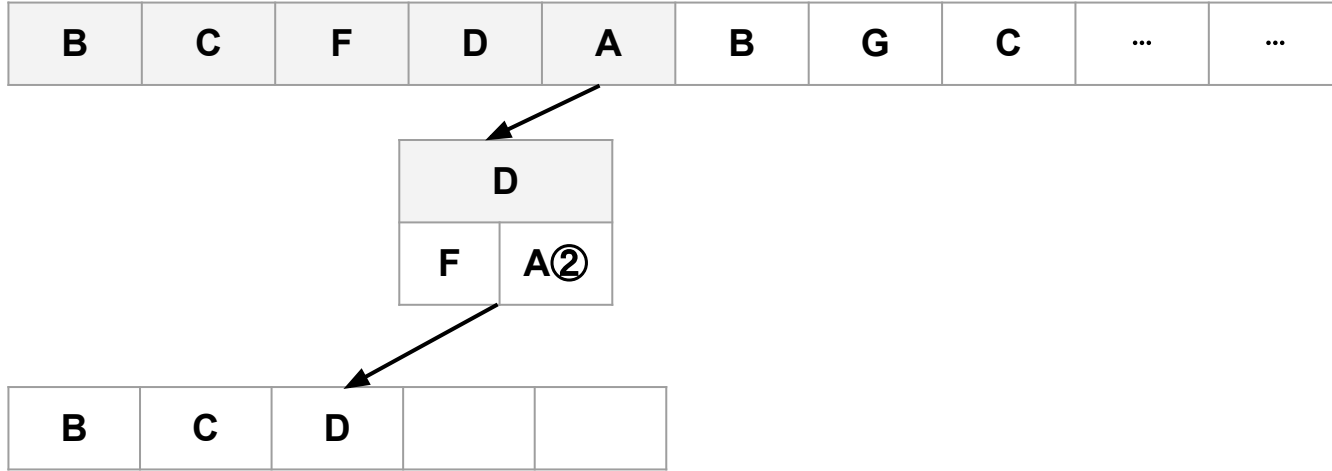
- ❖ We see that it's “too late” for A as we have already output values later than A.
- ❖ Instead we must save A for later and output A in a later run

## Replacement Selection: Generate Runs



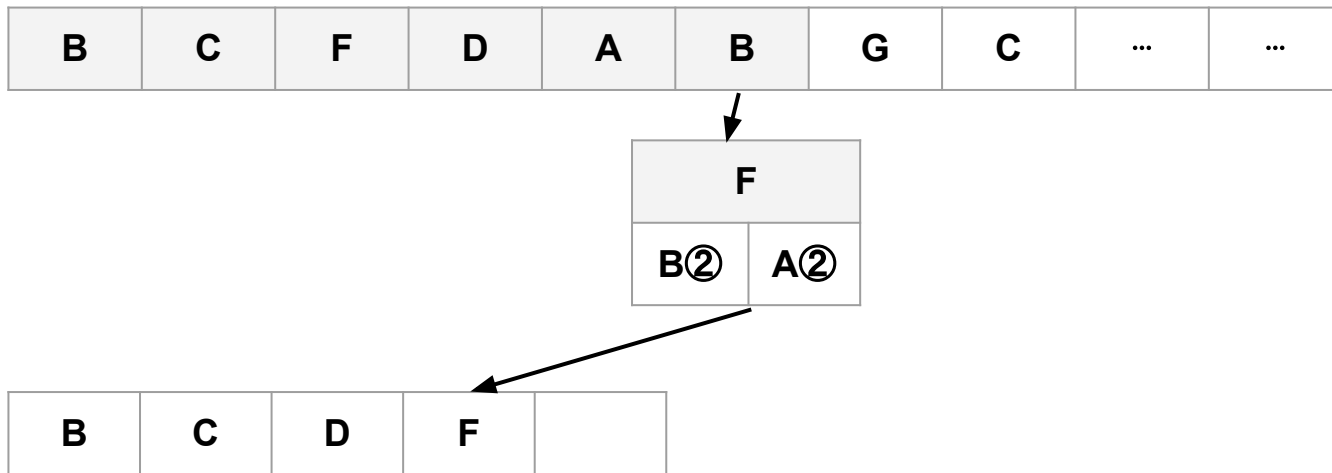
- ❖ We want see that it's “too late” for A as we have already output values later than A.
- ❖ Instead we must save A for later and output A in a later run

# Replacement Selection: Generate Runs



- ❖ We want see that it's “too late” for A as we have already output values later than A.
- ❖ Instead we must save A for later and output A in a later run
- ❖ Continue outputting values for the current sorted run

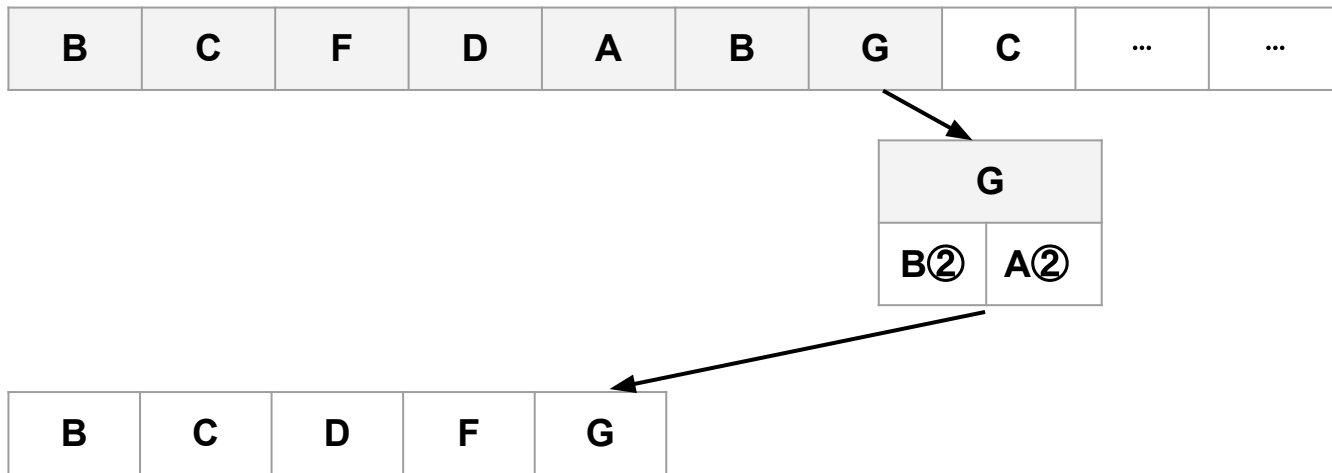
## Replacement Selection: Generate Runs



- ❖ We want see that it's “too late” for A as we have already output values later than A.
- ❖ Instead we must save A for later and output A in a later run
- ❖ Similarly we must save B for run 2
- ❖ But we can still output F in this run

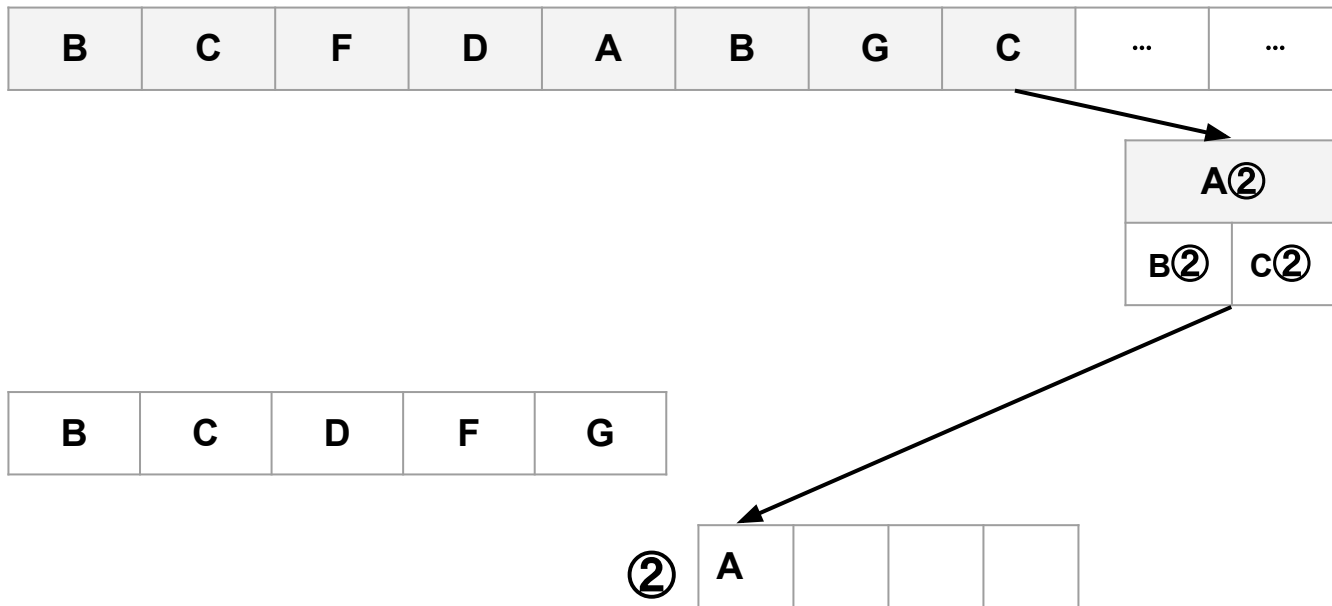


## Replacement Selection: Generate Runs



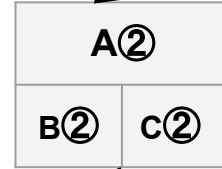
- ❖ We want see that it's “too late” for A as we have already output values later than A.
- ❖ Instead we must save A for later and output A in a later run
- ❖ Similarly we must save B for run 2
- ❖ But we can still output F in this run
- ❖ And we can see that G is still not too late for current run

# Replacement Selection: Generate Runs



- ❖ Eventually all values are saved for run 2
- ❖ Output lowest value in new sorted run

# Replacement Selection: Generate Runs



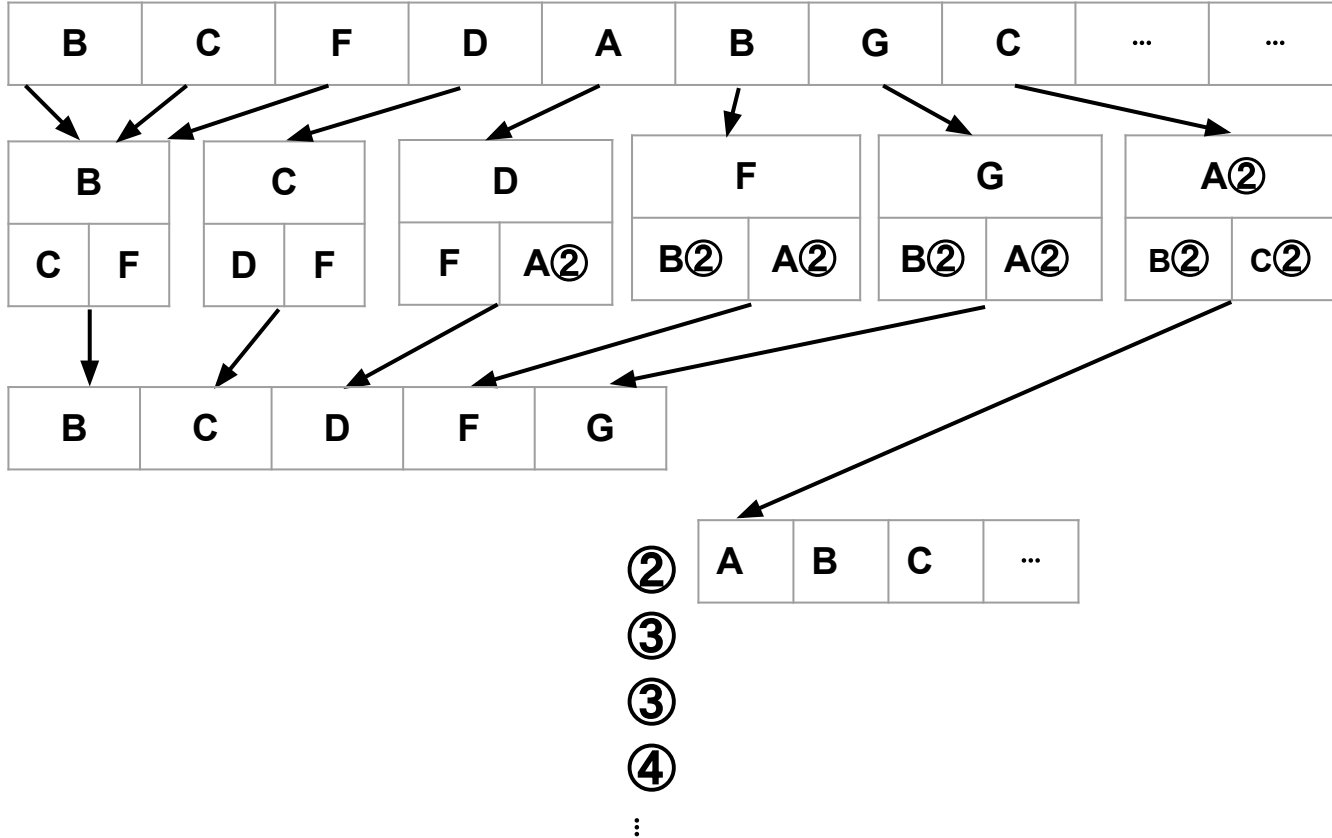
③

③

④

⋮

# Replacement Selection: Generate Runs



## Replacement Selection: Merge Runs

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| ① | B | C | D | F | G |
|---|---|---|---|---|---|

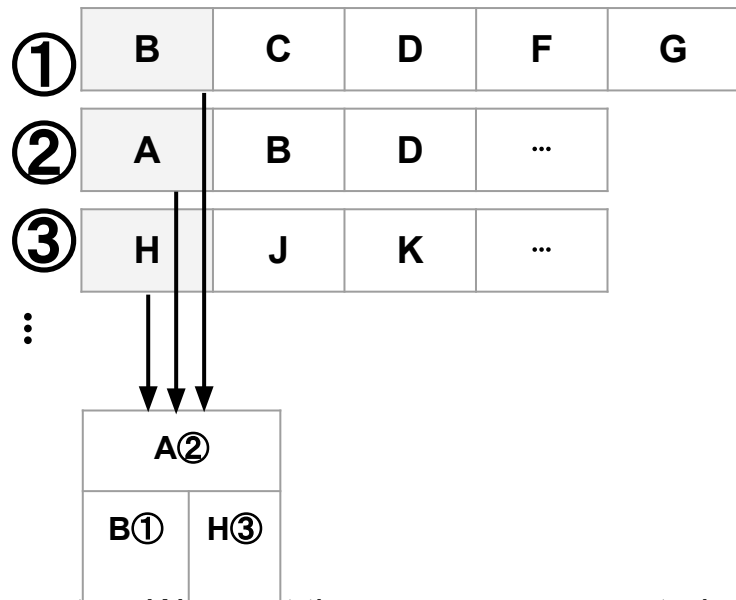
|   |   |   |   |     |
|---|---|---|---|-----|
| ② | A | B | D | ... |
|---|---|---|---|-----|

|   |   |   |   |     |
|---|---|---|---|-----|
| ③ | H | J | K | ... |
|---|---|---|---|-----|

⋮

- ❖ We must then merge many sorted runs into longer sorted runs

# Replacement Selection: Merge Runs

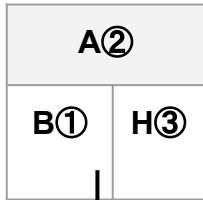


- ❖ We must then merge many sorted runs into longer sorted runs
- ❖ We load first element from sorted runs into priority queue (heap)
- ❖ We mark each element with which run it came from

# Replacement Selection: Merge Runs

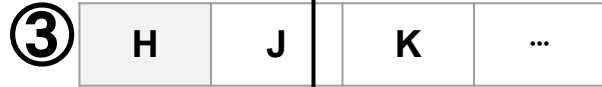


⋮

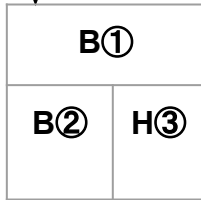


❖ Output lowest value to merged run

# Replacement Selection: Merge Runs



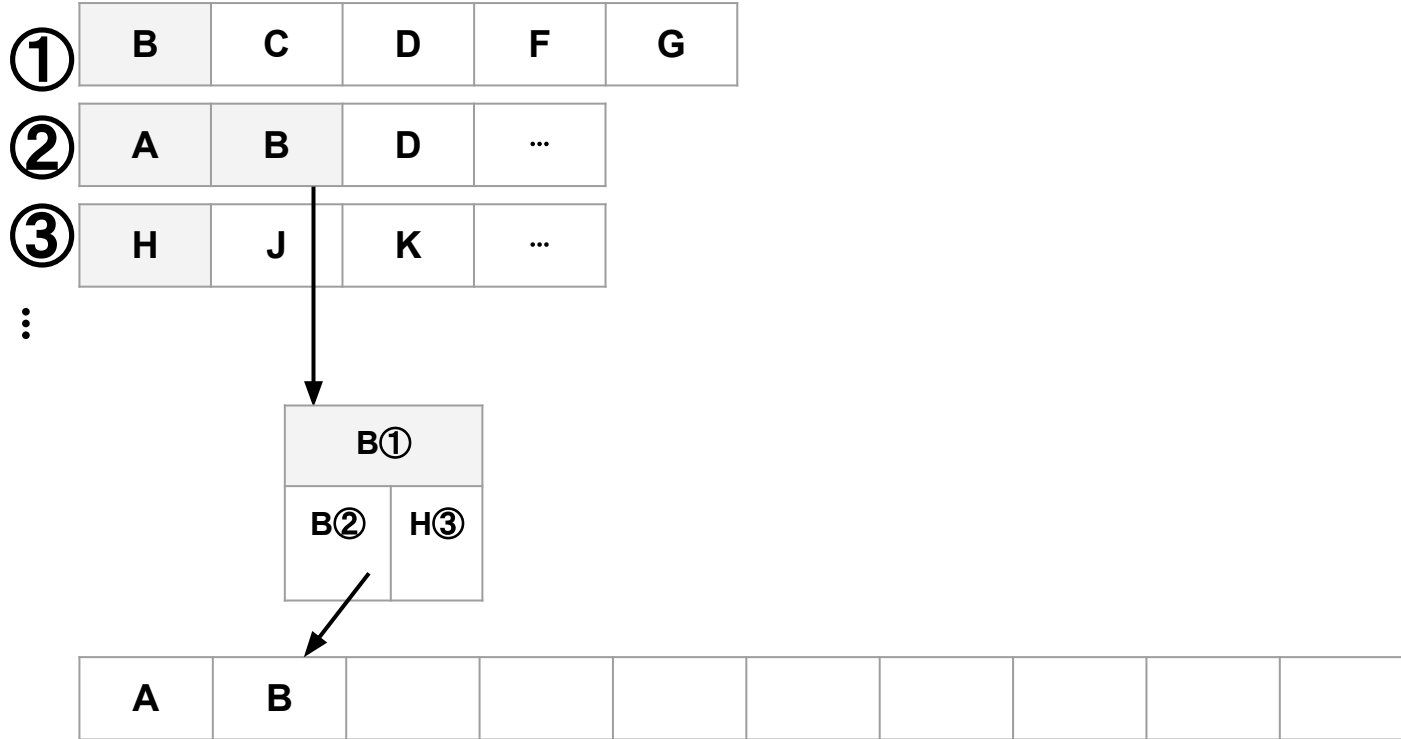
⋮



- ❖ Output lowest value to merged run
- ❖ Replace value with next value from the same run

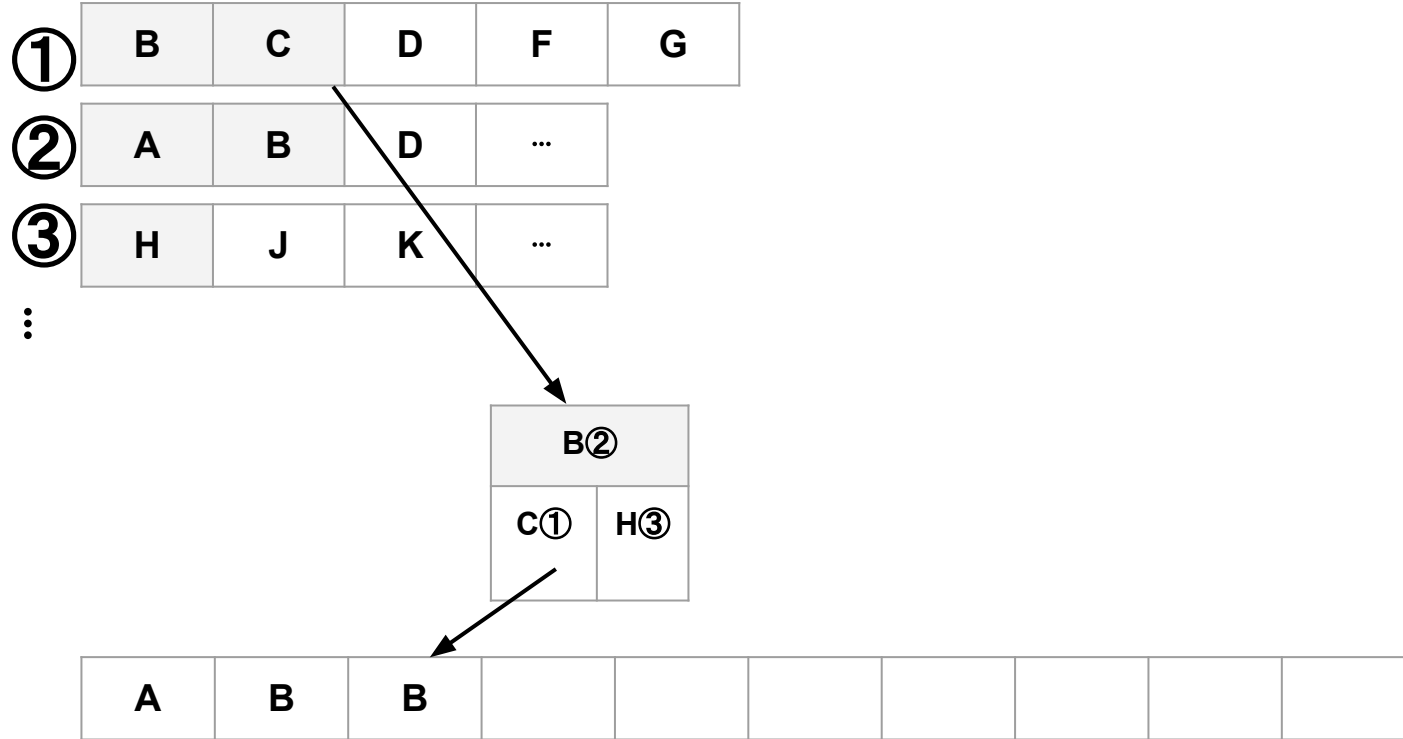


# Replacement Selection: Merge Runs



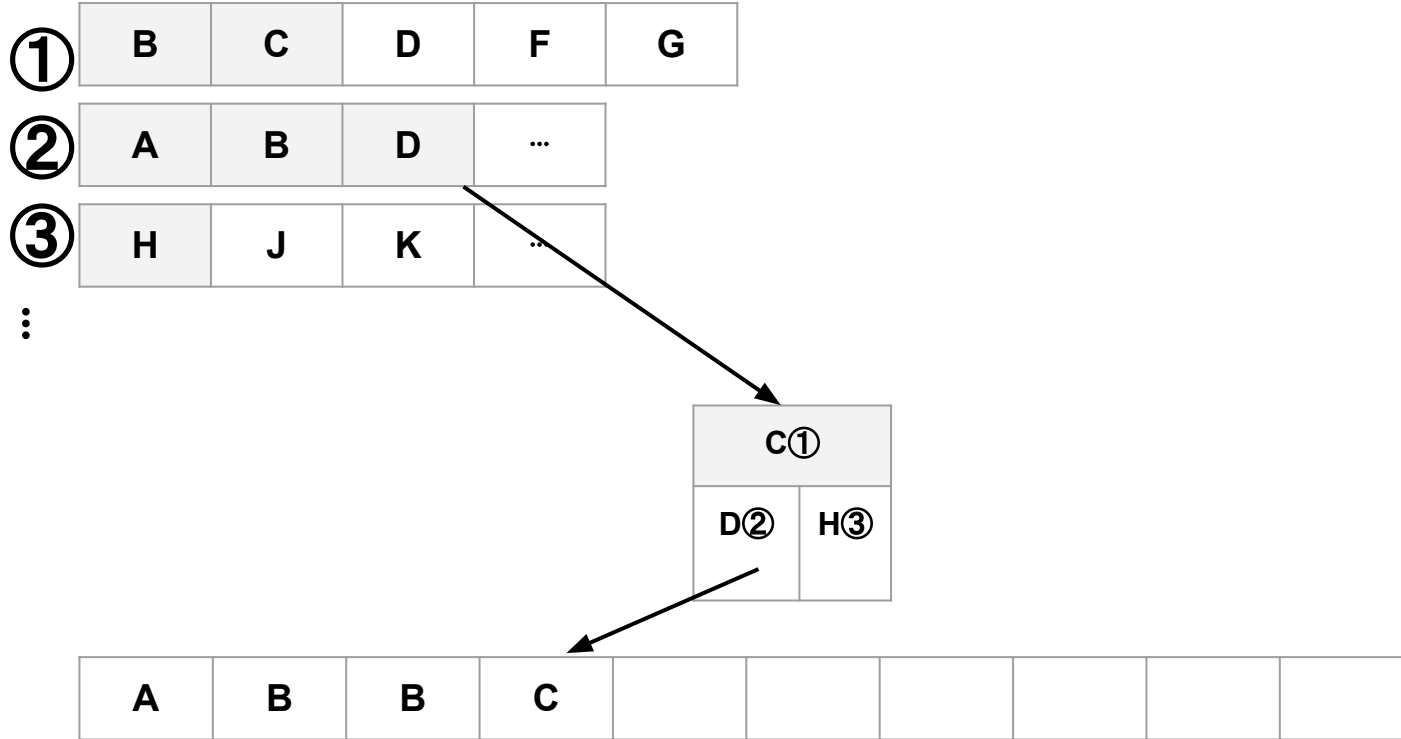
- ❖ Repeat outputting lowest value and replacing with next value from that sorted run

# Replacement Selection: Merge Runs



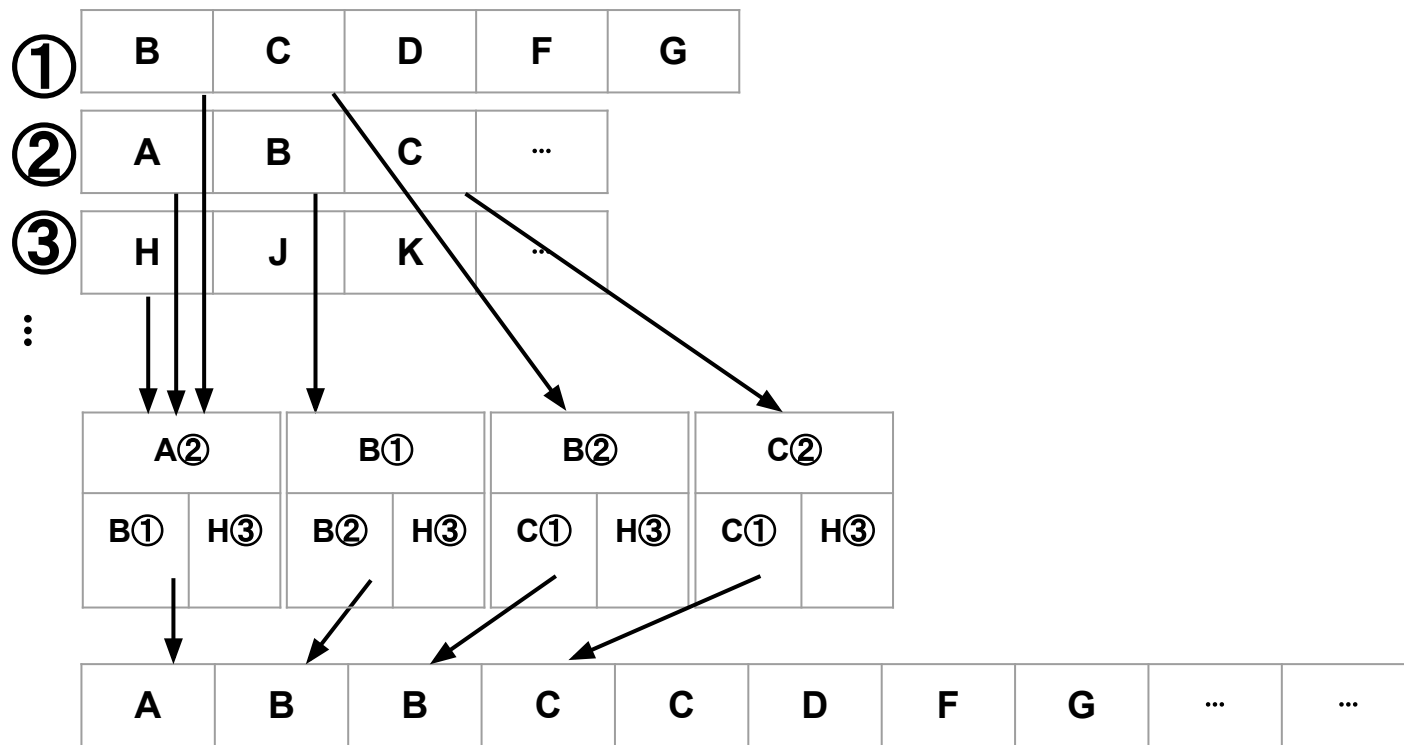
- ❖ Repeat outputting lowest value and replacing with next value from that sorted run

# Replacement Selection: Merge Runs



- ❖ Repeat outputting lowest value and replacing with next value from that sorted run

# Replacement Selection: Merge Runs



- ❖ This generates a longer sorted run containing all the elements from the runs we merged
- ❖ Repeat this process recursively until there's only a single long sorted run remaining with all data

# Replacement Selection: Merge Scheduling

• Our discussions have hitherto assumed that we have direct control over the input/output instructions for tape units and that no complicated operating system keeps us from using tape as efficiently as the tape designers intended. These idealistic assumptions give us insights into the tape merging problem, and may give some insights into the proper design of operating system interfaces, but we should realize that multiprogramming and multiprocessing can make the situation considerably more complicated.

• The issues we have studied in this section were first discussed in print by E. H. Friend [JACM 2 (1956), 134-168, W. Zoberlet [Elektronische Datenverarbeitung 5 (1960), 28-41, and M. A. Goetz [Digital Computer User's Handbook (New York: McGraw-Hill, 1967), 1:292-1:329].

**Summary.** We can sum up what we have learned about the relative efficiencies of different approaches to tape sorting in the following way:

**Theorem A.** It is difficult to decide which merge pattern is best in a given situation. □

The examples we have seen in Chart A show how 100,000 randomly ordered 10-character records (or 1 million 10-character records) might be sorted using six tapes under realistic assumptions. This much data fills about half of a tape, and it can be sorted in about 15 to 19 minutes on the RIT tapes. However, there is considerable variation in available tape equipment, and running times for such a job could vary between about four minutes and about two hours on different machines of the 1970s. In our examples, about 3 minutes of the total time were used for initial distribution of runs and internal sorting; about 41 minutes were used for the final merge and rewinding the output tape; and about 7 to 111 minutes were spent in intermediate stages of merging.

Given six tapes that cannot read backwards, the best sorting method under our assumptions was the "tape-splitting polyphase merge" (example 4), and for tapes that do allow backward reading, the best method turned out to be read-backward polyphase with a complicated placement of dummy runs (example 7). Oscillating sort (example 9) was a close second. In both cases the cascade merge provided a simpler alternative that was only slightly slower (examples 5 and 8). In the read-forward case, a straightforward balanced merge (example 1) was surprisingly effective, partly by luck in this particular example but partly also because it spends comparatively little time rewinding.

The situation would change somewhat if we had a different number of available tapes.

**Sort generators.** Given the wide variability of data and equipment characteristics, it is almost impossible to write a single external sorting program that is satisfactory in a variety of different applications. And it is also rather difficult to prepare a program that really handles tapes efficiently. Therefore the preparation of sorting software is a particularly challenging job. A sort generator is a program that produces machine code specially tailored to particular sorting applications.



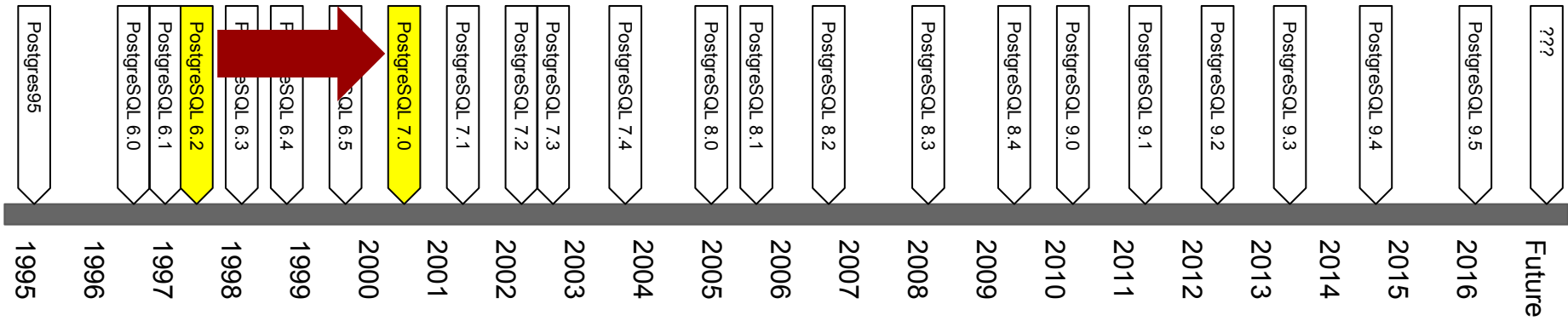
Chart A. Tape merging.

Legend

- Reading in forward direction
- Reading in backward direction
- Writing in forward direction
- Rewinding in backward direction
- Operator changing tapes

- ❖ Knuth dedicated a special pull-out section of his book to various orders in which to merge runs
- ❖ Each of these is a different strategy with different advantages and disadvantages
- ❖ Some depend on being able to read tapes backwards or have an operator change tapes
- ❖ They all assume you have a small fixed number of tape drives

# PostgreSQL 6.2 to 7.0 -- Two years go by



# PostgreSQL 7.0 - Major rewrite of psort. turns it into tuplesort.c



```
commit db3c4c3a2d980dcdd9a19feeddd11230587f0d21
```

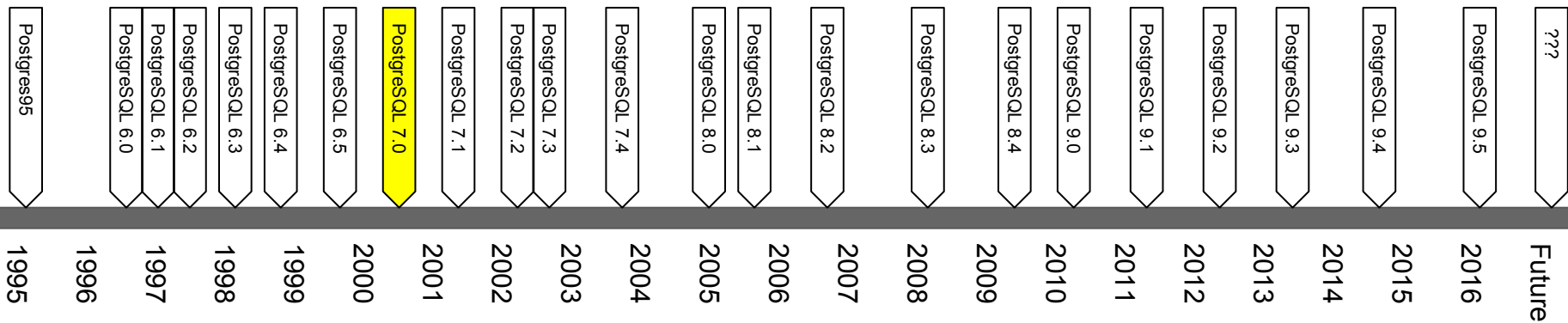
```
Author: Tom Lane <tgl@sss.pgh.pa.us>
```

```
Date: Wed Oct 13 15:02:32 1999 +0000
```

```
Split 'BufFile' routines out of fd.c into a new module, buffile.c. Extend
BufFile so that it handles multi-segment temporary files transparently.
This allows sorts and hashes to work with data exceeding 2Gig (or whatever
the local limit on file size is). Change psort.c to use relative seeks
instead of absolute seeks for backwards scanning, so that it won't fail
when the data volume exceeds 2Gig.
```

Tom Lane

[© David Wheeler \(CC BY-NC-SA 2.0\)](https://creativecommons.org/licenses/by-nc-sa/2.0/)



# PostgreSQL 7.0 - Reuse space aggressively



commit 957146dcec9cc2fb31220c26c51802623df04e5e

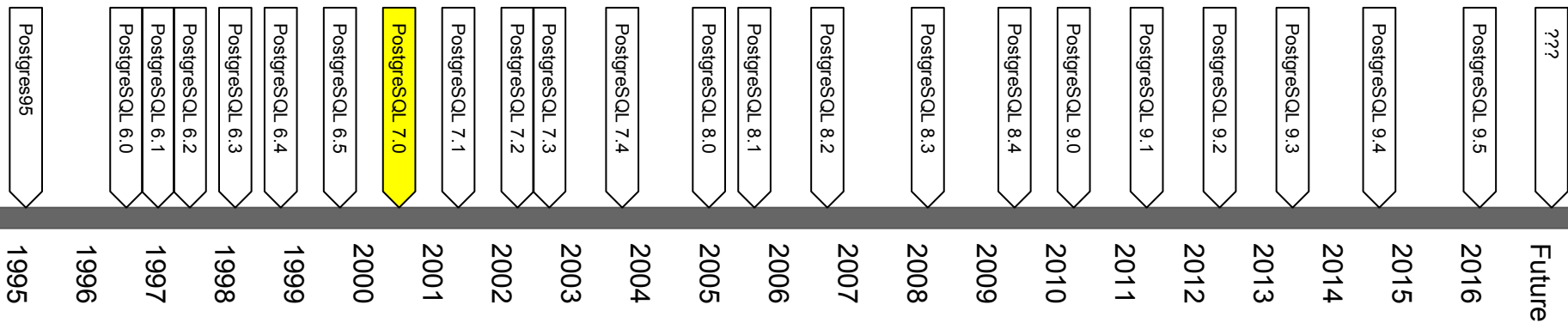
Author: Tom Lane <tgl@sss.pgh.pa.us>

Date: Sat Oct 16 19:49:28 1999 +0000

Second phase of psort reconstruction project: add bookkeeping logic to recycle storage within sort temp file on a block-by-block basis. This reduces peak disk usage to essentially just the volume of data being sorted, whereas it had been about 4x the data volume before.

Tom Lane

[© David Wheeler \(CC BY-NC-SA 2.0\)](#)





# PostgreSQL 7.0 - General purpose module for queries and index builds



commit 26c48b5e8cffafaf3b8acf345ca9fd8a1e408a54

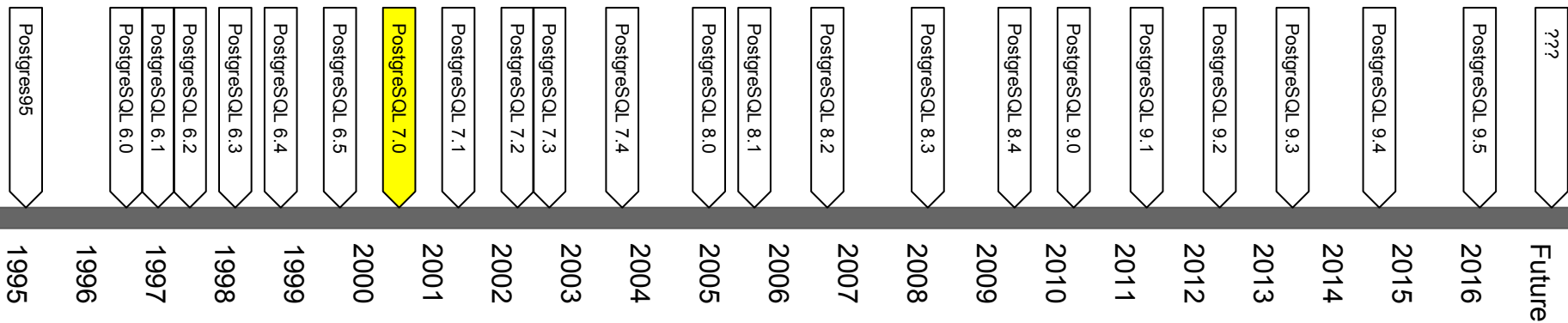
Author: Tom Lane <tgl@sss.pgh.pa.us>

Date: Sun Oct 17 22:15:09 1999 +0000

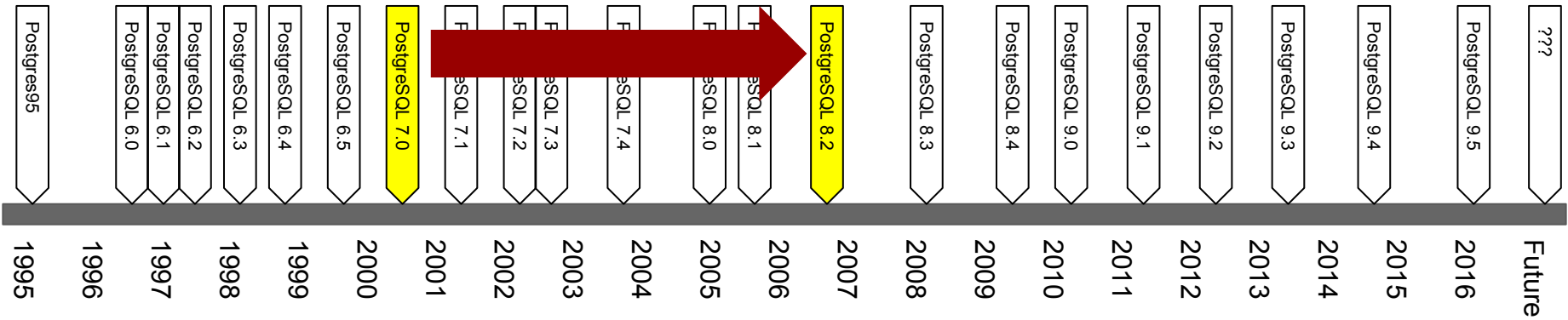
Final stage of psort reconstruction work: replace psort.c with a generalized module 'tuplesort.c' that can sort either HeapTuples or IndexTuples, and is not tied to execution of a Sort node. Clean up memory leakages in sorting, and replace nbtSORT.c's private implementation of mergesorting with calls to tuplesort.c.

Tom Lane

© David Wheeler (CC BY-NC-SA 2.0)



# PostgreSQL 7.0 to 8.2 -- Six years go by!



# PostgreSQL 8.2 - Virtual Tape Drives Are Cheaper than Real Drives



```
commit df700e6b40195d28dc764e0c694ac8cef90d4638
```

```
Author: Tom Lane <tgl@sss.pgh.pa.us>
```

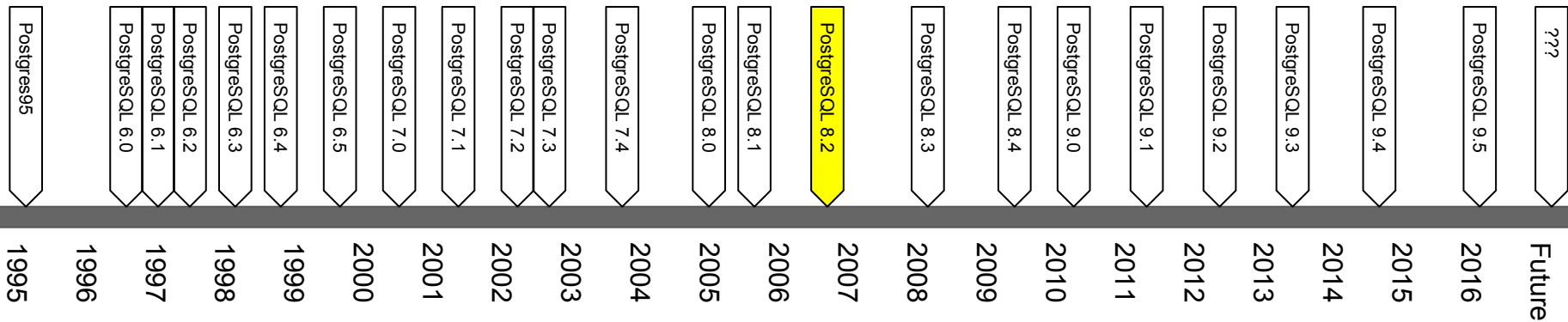
```
Date: Sun Feb 19 05:54:06 2006 +0000
```

Improve tuplesort.c to support variable merge order. The original coding with fixed merge order (fixed number of "tapes") was based on obsolete assumptions, namely that tape drives are expensive. Since our "tapes" are really just a couple of buffers, we can have a lot of them given adequate workspace. This allows reduction of the number of merge passes with consequent savings of I/O during large sorts.

Simon Riggs with some rework by Tom Lane

Simon Riggs

[© Alvaro Herrera \(CC BY-NC-SA 2.0\)](https://creativecommons.org/licenses/by-nc-sa/2.0/)

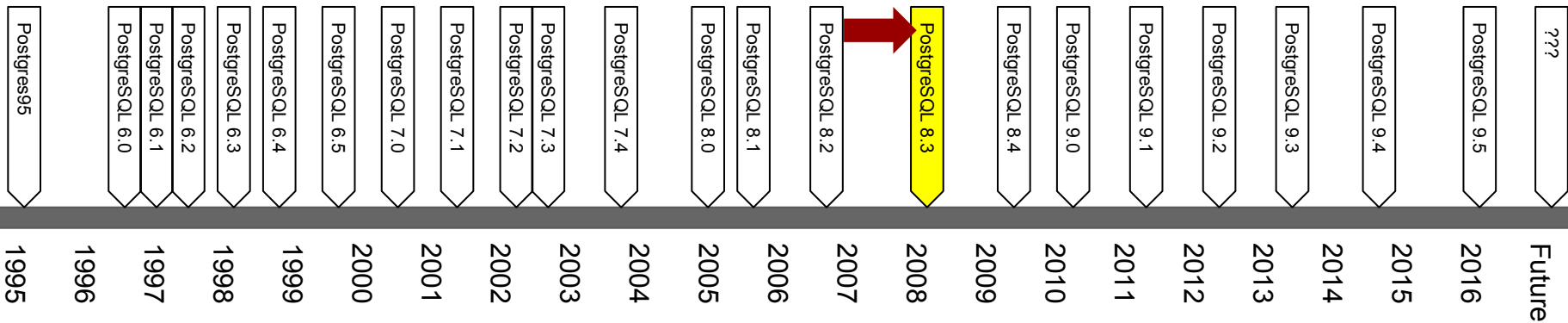


# PostgreSQL 8.3 - "External Merge" saves a round trip to disk



Gregory Stark

© Oleg Bartunov



```
commit 2415ad983174164ff30ce487c0e6b4b53321b83a
```

```
Author: Tom Lane <tgl@sss.pgh.pa.us>
```

```
Date: Mon May 21 17:57:35 2007 +0000
```

Teach tuplestore.c to throw away data before the "mark" point when the caller is using mark/restore but not rewind or backward-scan capability. Insert a materialize plan node between a mergejoin and its inner child if the inner child is a sort that is expected to spill to disk. The materialize shields the sort from the need to do mark/restore and thereby allows it to perform its final merge pass on-the-fly; while the materialize itself is normally cheap since it won't spill to disk unless the number of tuples with equal key values exceeds work\_mem.

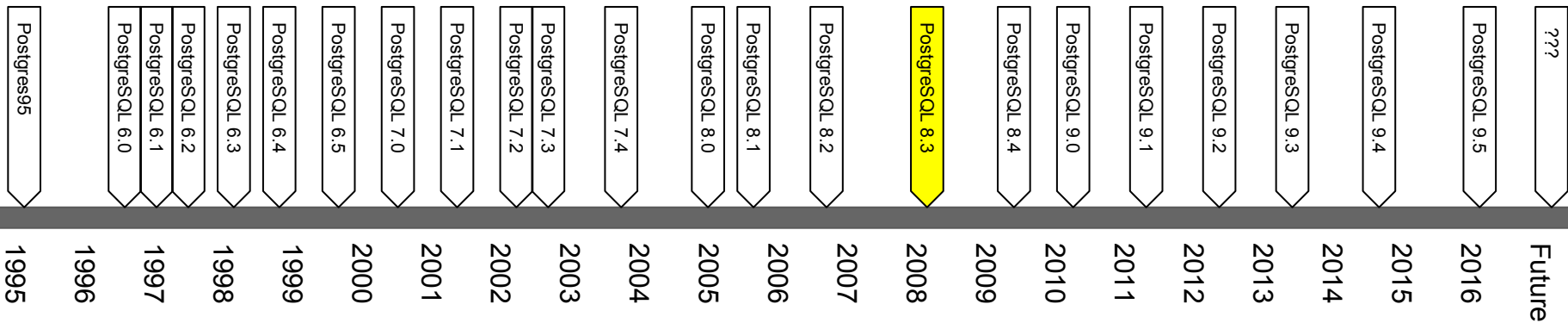
Greg Stark, with some kibitzing from Tom Lane.

# PostgreSQL 8.3 - Top-N sorting



Gregory Stark

© Oleg Bartunov



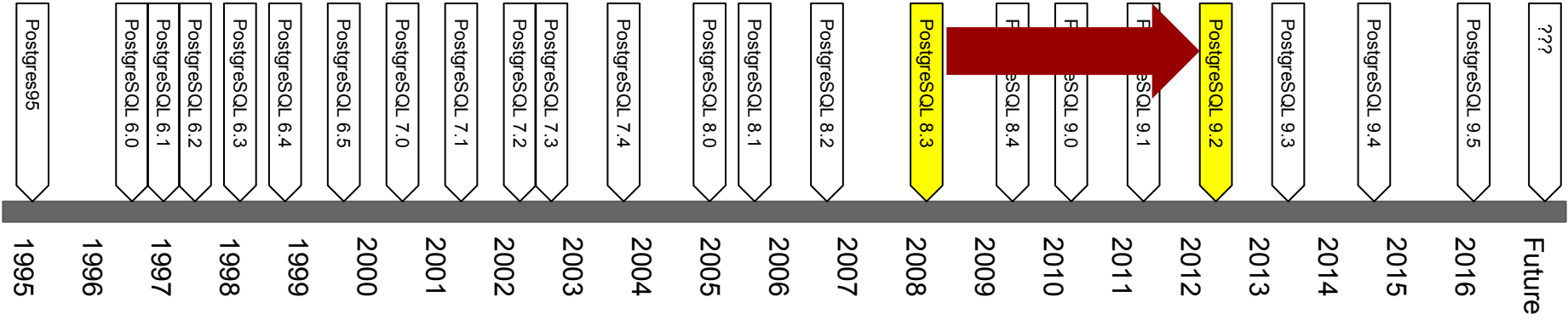
```
commit d26559dbf356736923b26704ce76ca820ff3a2b0
```

```
Author: Tom Lane <tgl@sss.pgh.pa.us>
```

```
Date: Fri May 4 01:13:45 2007 +0000
```

Teach tuplesort.c about "top N" sorting, in which only the first N tuples need be returned. We keep a heap of the current best N tuples and sift-up new tuples into it as we scan the input. For M input tuples this means only about  $M \cdot \log(N)$  comparisons instead of  $M \cdot \log(M)$ , not to mention a lot less workspace when N is small --- avoiding spill-to-disk for large M is actually the most attractive thing about it. Patch includes planner and executor support for invoking this facility in ORDER BY ... LIMIT queries. Greg Stark, with some editorialization by moi.

# PostgreSQL 8.3 to 9.1 -- Four years go by...



# PostgreSQL 9.2 - SortSupport



Peter Geoghegan

© Oleg Bartunov

```
commit c6e3ac11b60ac4a8942ab964252d51c1c0bd8845
```

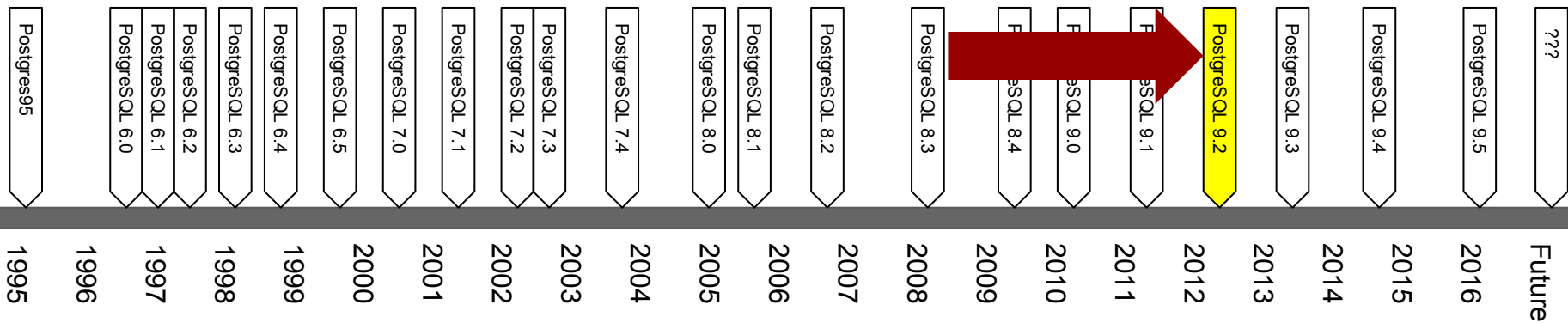
```
Author: Tom Lane <tgl@sss.pgh.pa.us>
```

```
Date: Wed Dec 7 00:18:38 2011 -0500
```

Create a "sort support" interface API for faster sorting.

This patch creates an API whereby a btree index opclass can optionally provide non-SQL-callable support functions for sorting. In the initial patch, we only use this to provide a directly-callable comparator function, which can be invoked with a bit less overhead than the traditional SQL-callable comparator. While that should be of value in itself, the real reason for doing this is to provide a datatype-extensible framework for more aggressive optimizations, as in Peter Geoghegan's recent work.

Robert Haas and Tom Lane



# PostgreSQL 9.2 - Specialized quicksorts with inlined comparators



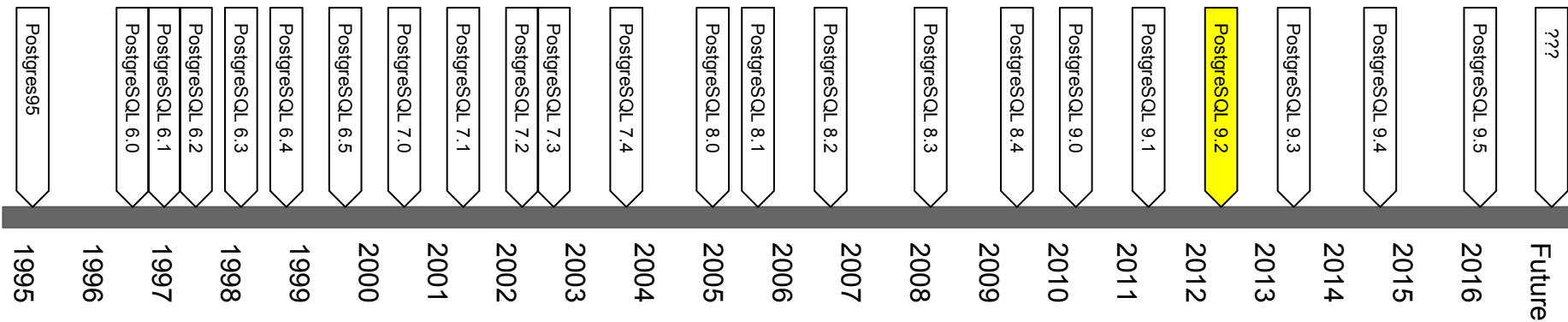
Peter Geoghegan

© Oleg Bartunov

```
commit 337b6f5ecf05b21b5e997986884d097d60e4e3d0
Author: Robert Haas <rhaas@postgresql.org>
Date:   Wed Feb 15 12:13:32 2012 -0500
```

Speed up in-memory tuplesorting.

Per recent work by Peter Geoghegan, it's significantly faster to tuplesort on a single sortkey if ApplySortComparator is inlined into quicksort rather than reached via a function pointer. It's also faster in general to have a version of quicksort which is specialized for sorting SortTuple objects rather than objects of arbitrary size and type. This requires a couple of additional copies of the quicksort logic, which in this patch are generated using a Perl script. There might be some benefit in adding further specializations here too, but thus far it's not clear that those gains are worth their weight in code footprint.





# PostgreSQL 9.3 - Use all of work\_mem for large sorts

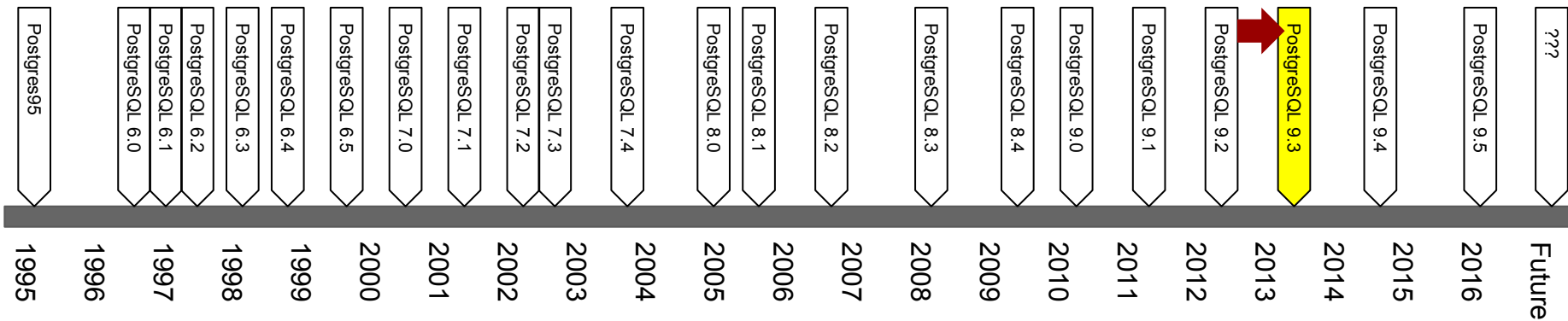
commit 8ae35e91807508872cabd3b0e8db35fc78e194ac

Author: Tom Lane <tgl@sss.pgh.pa.us>

Date: Thu Jan 17 13:12:14 2013 -0500

Improve memory space management in tuplesort and tuplestore.

The code originally just doubled the size of the tuple-pointer array so long as that would fit in allowedMem. This could result in failing to use as much as half of allowedMem, if (as is typical) the last doubling attempt didn't quite fit. Worse, we might double the array size but be unable to use most of the added slots, because there was no room left within the allowedMem limit for tuples the slots should point to. To fix, double only so long as we've used less than half of allowedMem in total. Then do one more array enlargement, but scale it based on total memory consumption so far. This will work nicely as long as the average tuple size is reasonably stable, and in any case should be better than the old method.



# PostgreSQL 9.3 - Allow using more than 2GB for sorts

commit 263865a48973767ce8ed7b7788059a38a24a9f37

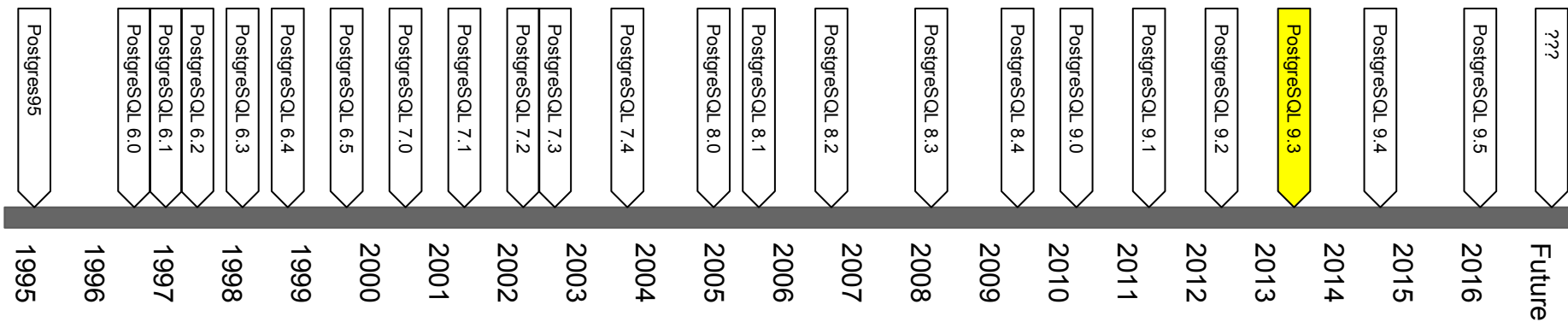
Author: Noah Misch <noah@leadboat.com>

Date: Thu Jun 27 14:53:57 2013 -0400

Permit super-`MaxAllocSize` allocations with `MemoryContextAllocHuge()`.

The `MaxAllocSize` guard is convenient for most callers, because it reduces the need for careful attention to overflow, data type selection, and the `SET_VARSIZE()` limit. A handful of callers are happy to navigate those hazards in exchange for the ability to allocate a larger chunk. Introduce `MemoryContextAllocHuge()` and `realloc_huge()`. Use this in `tuplesort.c` and `tuplestore.c`, enabling internal sorts of up to `INT_MAX` tuples, a factor-of-48 increase. In particular, B-tree index builds can now benefit from much-larger `maintenance_work_mem` settings.

Reviewed by Stephen Frost, Simon Riggs and Jeff Janes.



# PostgreSQL 9.5 - Quicksort specializations with inlined comparators



Peter Geoghegan

© Oleg Bartunov

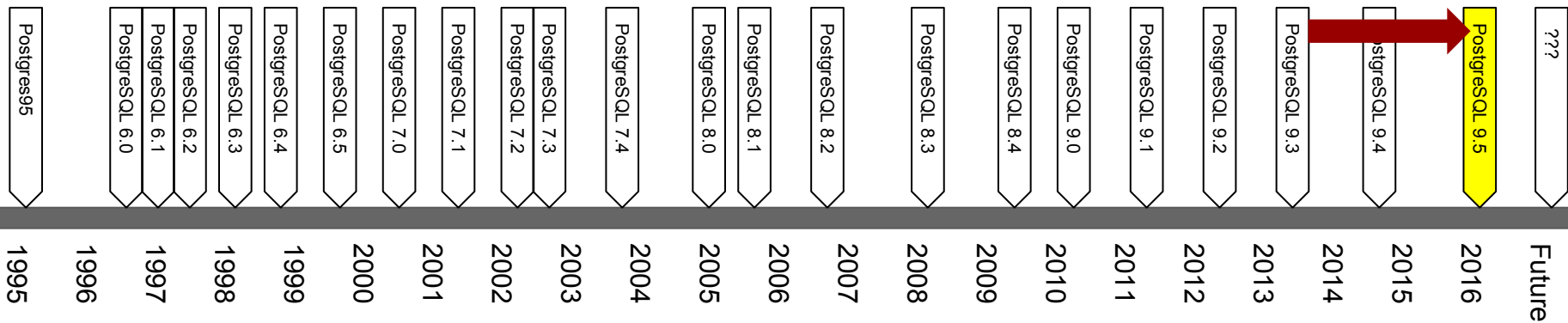
```
commit 4ea51cdfe85ceef8afabceb03c446574daa0ac23
```

```
Author: Robert Haas <rhaas@postgresql.org>
```

```
Date: Mon Jan 19 15:20:31 2015 -0500
```

Use abbreviated keys for faster sorting of text datums.

This commit extends the SortSupport infrastructure to allow operator classes the option to provide abbreviated representations of Datums; in the case of text, we abbreviate by taking the first few characters of the strxfrm() blob. If the abbreviated comparison is insufficient to resolve the comparison, we fall back on the normal comparator. This can be much faster than the old way of doing sorting if the first few bytes of the string are usually sufficient to resolve the comparison.



# Abbreviated Keys For Text Data

```
strcoll("Олег Бартунов", "Магнус Хагандер") == 3
```

```
=> "Олег Бартунов" > "Магнус Хагандер"
```

- ❖ strcoll can be very slow because comparison rules can be very complicated
- ❖ Even when comparing ASCII it is much slower than integer comparisons

```
strxfrm("Олег Бартунов")
```

```
=> C393C38EC382C2BEC2BCC2BVC395C397C39AC391C393C2BD01...
```

```
strxfrm("Магнус Хагандер")
```

```
=> C390C2BVC2BEC391C39AC396C39DC2BVC2BEC2BVC391C2BFC3...
```

- ❖ Result from strxfrm can be compared quickly using memcmp
- ❖ But result can be very large which requires more memory and disk

## Abbreviated Keys For Text Data

- ❖ Use first 8 bytes (4 bytes on 32-bit platforms) as integer for fast comparisons

```
strxfrm("Олег Бартунов")
```

```
=> C393C38EC382C2BEC2BCC2BVC395C397C39AC391C393C2BD01...
```

```
strxfrm("Магнус Хагандер")
```

```
=> C390C2BVC2BEC391C39AC396C39DC2BVC2BEC2BVC391C2BFC3...
```

```
0xC393C38EC382C2BE > 0xC390C2BVC2BEC391
```

```
=> TRUE
```

- ❖ When two strings start with similar prefixes then the integers may be equal
- ❖ Call strcoll to disambiguate these cases

# What lies ahead? Hardware evolution driving major changes in algorithm choices:

- Quicksort to generate runs instead of Replacement Selection
  - Quicksort is cache oblivious and CPU cache is much more important in modern CPUs
  - As main memory grows the number of tapes we can handle increases proportionally and the number of runs decreases proportionally. So the number of merges decreases quadratically. It's less important to use a heap to maximize the run length.
- Parallel Sort
  - Modern hardware is scaling by adding CPUs faster than by increasing clock speeds.
  - Infrastructure for parallel query is already committed and can be used for sorting.
- Using SIMD instructions (MMX/SSE/AVX)
  - Most operators limited to floats but recently more general purpose (integer) vector operations have been supported
  - Registers keep increasing in size -- the next generation of CPUs will have 512 byte registers which may be sufficient
- Using a GPU to sort using OpenCL or CUDA
  - Quicksort unsuitable -- Radixsort or Bitonic sort would be needed
  - <https://wiki.postgresql.org/wiki/PGStrom>

