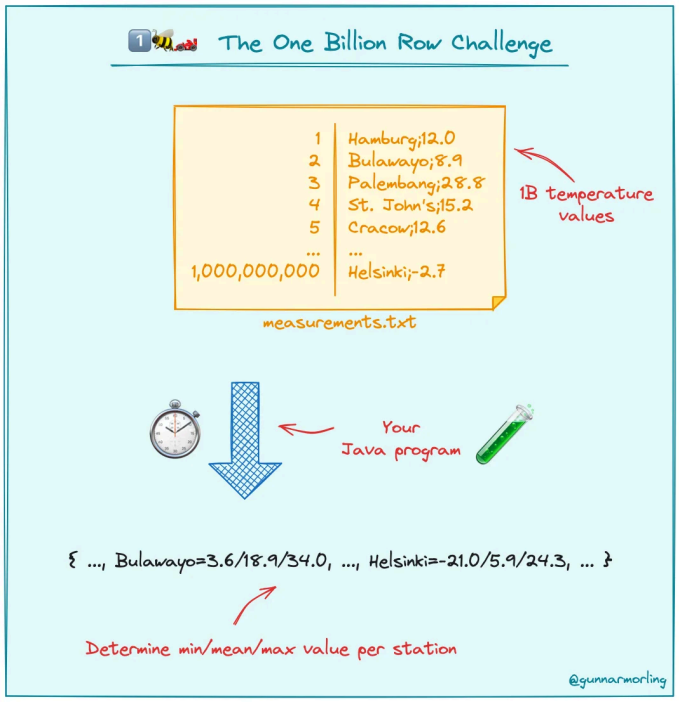
The One Billion Row Challenge Shows That Java Can Process a One Billion Rows File in Two Seconds

This item in japanese

On the first day of 2024, Gunnar Morling, senior staff software engineer at Decodable, launched The One Billion Row Challenge (1BRC) to the Java Community. This ongoing challenge will run until the end of January and aims to find Java code that processes one billion rows in the fastest time. Until now, the podium contained algorithms that finished the processing in under 1.7 seconds.



The challenge's rules are simple: only the SDK features running on any Java distribution may be used. Hence, external libraries or data stores are excluded from the solution. InfoQ reached out to Morling, Eliot Barlas, principal software engineer at GoTo, Roy van Rijn, director at OpenValue Rotterdam, and Thomas Wuerthinger, vice president of software development at Oracle and founder of GraalVM, for a better understanding of this challenge.

InfoQ: This is an exciting challenge. Can you describe it, please? What was the motivation behind it?

> ***Morling:*** *The 1BRC is a coding challenge based on a deceptively simple task: parsing temperature measurement values from a text file and determining the min, max, and mean temperature for each weather station present. The caveat: the file has 1,000,000,000 entries!*
>
> *I wanted to create the opportunity to explore high-performance programming*

> *techniques, new APIs (like Vector API - which leverages CPU SIMD instructions), the capabilities of different Java distributions, and anything else that will prove how fast Java has become.*

InfoQ: How can anybody join the challenge?

> ***Morling:*** *Start with the `README` file and by cloning the repo. Try to implement your solution then see what others tried as well - it's all about the learning in the end.*

InfoQ: Did you encounter anything surprising in the solution space?

> ***Morling:*** *Hackers gonna hack: many solutions optimised for the specific key set (i.e., weather station names). This worked just for this particular dataset. With the help of the community, we clarified the purpose.*
>
> *There were many interesting solutions: the use of SIMD and the new Java native memory API (something I was hoping for) and highly optimized parsing functions including SWAR (SIMD within a register), which I didn't quite expect to that extent. By now, people working on the fastest entries are deep into native optimization territory, counting CPU instructions, evaluating branch mispredictions etc.*

InfoQ: Please describe your solution. Were there particular techniques or technologies you wanted to try?

> ***Eliot Barlas:***
>
> *My solution is to partition the file into ranges equal to the number of available processors. For each partition, there is a task that computes statistics for each weather station on separate threads. When these tasks are finished, the final results are aggregated into a final table of statistics.*
>
> *The data in each partition is memory-mapped and accessed via a **`MappedByteBuffer`** that covers the entire partition byte range. The tasks move through the data in the partition, one byte or int at a time, using **`ByteBuffer`**. The weather station names are extracted and stored using **`sun.misc.Unsafe`** as sequences of integers.*

> ***Roy van Rijn:*** *The solution is an evolutionary one that started by using plain data structures and APIs offered by the SDK (like* ***`BufferedInputStream`*** *or* ***`HashMap`****). Step by step, it evolved into using* ***Unsafe*** *for direct access to the memory. Parallelism, branchless code and implementing a SWAR (****SIMD*** *as a Register) were among the decisions that made my solution one of the top contenders of the challenge to date. For storage, I implemented my own "very simple" hashmap backed by an array based on the linear probing concept.*

> ***Thomas Wuerthinger:*** *The first part of the solution splits the workload into the number of cores available on the target processor for parallelization. It uses Java's capabilities to memory map the input file for the most efficient direct memory access. The innermost loop of parsing the data is performed with a technique that tries to create code without branches and instead performs a few complex arithmetic and bit operations. For this specific problem, the branches are often mispredicted by the processor due to the random nature of the input and therefore avoiding branches is key for maximizing performance.*

InfoQ: Would it be possible to improve your solution further?

> ***Barlas:*** *I've been following Project Panama from afar, but the 1BRC offered a chance to explore foreign memory capabilities in an applied way. [...] I have not succeeded in achieving speed-up using the Project Panama Vector API. For example, early on I attempted to use the* ***`ByteVector`*** *API to perform fast weather station name comparisons. I'd like to revisit that using other types of vectors or in conjunction with the* ***`MemorySegment`*** *interface.*

> ***Wuerthinger:*** *The possible improvements now depend a lot on the target hardware. Specifically, one can trade off the aspects of memory bandwidth, compute bandwidth, and branch prediction reliance.*

> ***Roy van Rijn:*** *At the top, the approaches are similar. The concept I'm currently trying to explore is "mechanical sympathy," trying to improve the instructions that need to be executed in a way that best fits the machine we're testing on.*

InfoQ: What is your conclusion about this interesting beginning-of-the-year challenge?

> ***Morling:*** *For sure Java, its ecosystem, and its community are thriving more than ever! It is just so encouraging to see how many joined the challenge, including some very well-known developers. Everybody learned: either by coding or by reading the code. The help of the community was truly humbling and made the challenge grow this much.*

The challenge was welcomed by the coders' community, with Morling describing its success as "*This all went way beyond what I'd have expected.*" Even though the podium seems to be taken by solutions running on GraalVM, there were submissions using OpenJDK builds, Amazon Corretto or Eclipse Temurin. Morling further commented, "*Graal is a great fit for the task at hand, providing a few extra per cent of performance for free."*

The challenge extended beyond the boundaries of the Java ecosystem solutions being written in Rust, Go, C++, or even SQL and Shell. As for the Java ecosystem, Thomas Wuerthinger, Quan Anh Mai, and Alfonso Peterssen shared the gold medal. The silver and bronze go to Artsiom Korzun and Jaromir Hamala, respectively.

Morling extended his appreciation to the community and Decodable, which provided the evaluation machine.

This news item was updated on February 4th, 2024, to incorporate the results of the challenge.

About the Author

Olimpiu Pop

Tech Executive and Engineer Focused on a Holistic Approach and using technology to provide solutions to real problems with minimal impact on the environment. He has experience in developing real-time applications ranging from financial software to IAM. Passionate about tooling and optimising development flows with or without AI. Led and shaped technical organisations of hundreds of developers (from support engineers to Architects). Tech community builder: Transylvania JUG facilitator, member of the program committee for Voxxed Romania and Devoxx UK, conference speaker and podcaster on cybersecurity and open-source topics for 505updates.com. Main editor and troublemaker of JavaAdventCalendar.