RMQ Benchmark Example

Florian Kurpicz

January 11, 2021

1 Experimental Setup

In this experimental evaluation of range minimum queries, we compare the following algorithms and data structures.

rmq_scan A naive scanning solution, which we use as baseline,

rmq_nlgn The $O(n \lg n)$ space solution described in the course *Text Indexing and Information Retrieval*

rmq_n The $O(n)$ space solution described in the same course with some practical improvements, and

rmq_ferrada The implementation by Ferrada and Navarro (downloaded from `https://github.com/hferrada/rmq/`).

The first three algorithms and data structures are implemented by us. The source code is available at `Link`. Our experiments were conducted on a computer with four Intel Xeon CPUs E5-4620 v4 (each with 10 cores and 20 threads, as Hyperthreading was enabled, 2.1 GHz base frequency (2.6 GHz maximum turbo frequency), 32 KB L1, 256 KB L2, 25 MB L3 cache) and 252 GB available RAM. Since all algorithms are sequential and use only a single core. The code was compiled with GCC 9.2.0 and compiler flags `-O3`, `-march=native`. All reported results are the average of five runs. We use random generated numbers as inputs. To this end, we generated these numbers using the Standard Template Library random number generators.

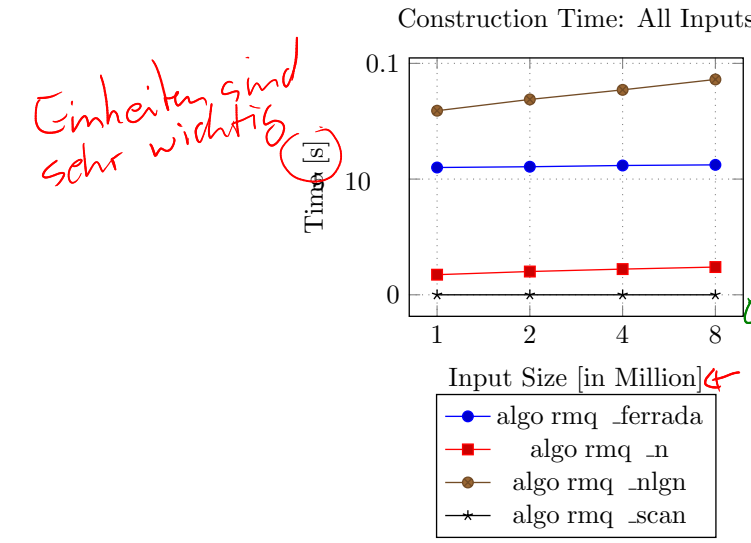
Welche Algorithmen werden getestet

↳ Woher kommt der Algorithmus

→ Woher kommen die Testdaten → Am besten keine Zufallszahlen verwenden

* Welche Hardware → Welche CPU muss dabei stehen, genauere Angaben sind optional
→ Speicherplatz auch angeben
→ Weitere Informationen, die für die Experimente wichtig sind.
↳ z.B. bei External Memory die Art und Anzahl der Festplatten und wie sie angebunden sind

Wie wurde der Code kompiliert, auch mit welchen Flags
↳ O3 → max. Optimierung
↳ O2, O1 weniger Opt.
↳ O0 gar keine Opt.
↳ Os Optimiert den Platz der ausführbaren Datei



Einheiten sind sehr wichtig

Achsenskalierung in diesem Fall logarithmisch

Auch die Größenordnung ist wichtig

Was sehen wir hier

↳ Konstruktionszeiten für die 4 Algorithmen

↳ — " — $\approx$ proportional zur Eingabegröße

↳ rmq-scan hat keine Konstruktionszeit

↳ Von schnell nach langsam: rmq-scan, rmq-n, rmq-ferreda, rmq-nlgn

Interpretation

↳ Dynamische Prog. bei rmq-nlgn dauert sehr lange

↳ rmq-ferreda macht wohl mehr als rmq-n

Was sehen wir

↳ Konstruktionszeit ist unabhängig von der Eingabe

Interpretation

↳ Algorithmen arbeiten Datenunabhängig

Normierte Achse

0

—— Input Size

immer noch mal probelesen

Was sehen wir

↳ Bei normierten Achsen erwarten wir waagerechte Plots

↳ rmg-nlgn benötigt am meisten Platz

↳ rmg-nlgn hat Platzbedarf wie theoretisch erwartet

↳ rmg-ferreda benötigt fast keinen Platz

↳ rmg-n benötig fast 10x so viel Platz wie rmg-ferreda

—— Input Size

Interpretation

↳ rmg-ferreda benötigt länger für die Konstruktion, spart dadurch aber auch Platz

Was sehen wir

- Von schnell nach langsam:
  rmq-nlgn, rmq-n
  rmq-ferreda, rmq-scan

und alle Längen

Interpretation

- Je mehr Platz wir benötigen, desto schneller können wir Queries beantworten
- Es gibt also eine Space-Time-Tradeoff

Was sehen wir

↳ relativ ähnliche Laufzeiten für alle Verteilungen
↳ bei Uniform sind alle Algorithmen etwas langsamer

Interpretation

↳ Auch die Anfragezeit ist (fast) Datenunabhängig

Auch weite Messwerte können interessant sein

Was sehen wir

- Ring-Scan startet wie erwartet an der Länge ab
- Alle anderen Algorithmen nicht

Interpretation

- Auch die Implementierungen haben eine konstante Laufzeit

Schlussfolgerung

- Wir haben einen klaren Space-Time-Tradeoff
- Je mehr Platz wir zur Verfügung haben, desto schneller können wir RMQs beantworten
- Je nach Anwendung muss entschieden werden, welche RMQ-Datenstruktur verwendet werden kann.