Better than next()

• What’s the worst case for sequential merge-based intersection?

• \{52, 1\} ➞ move \(k_2\)’s cursor
  – To the position whose id is at least 52 ➞ **skipTo(52)**
  – Essentially, asking the first \(i\), such that \(K_2[i] \geq 52\) (\(K_2\)’s list is sorted).
  – Takes many sequential call of next()
  – Could use binary search in the rest of the list
  – Cost: \(\left\lceil \log_2(N_{\text{remainder}}) \right\rceil\)

\[
\begin{array}{cccccccc}
K_2: & 1 & 3 & 5 & \cdots & \cdots & \cdots & \cdots & 79 \\
K_1: & 52 & 54 & 56 & 58
\end{array}
\]
skipTo(id)

- Galloping search (gambler’s strategy)
  - [Stage 1] Doubling the search range until you overshoot
  - [Stage 2] Perform binary search in the last range

- Performance analysis (worst case)
  - Let the destination position be $n$ positions away.
  - $\approx \log_2 n$ probes in Stage 1 + $\approx \log_2 n$ probes in Stage 2
  - Total = $2 \left\lceil \log_2 (n+1) \right\rceil = O(\log_2 n)$
Total Cost

- Galloping search (gambler’s strategy)
  - Cost of the i-th probe: $\approx 2 \log_2(n_i)$
  - Total cost of $K_1$ probes: $\approx 2 \log_2(\prod_{i=1}^{K_1} n_i) \leq 2 \log_2(\frac{(\sum_{i=1}^{K_2} n_i)}{|K_1|})^{\frac{|K_1|}{K_2}} \leq 2|K_1|*\log_2(|K_2|/|K_1|)$

- Asymptotically, resembles linear merge when $|K_2|/|K_1| = O(1)$, resembles binary search when $|K_1| = O(1)$

What about list intersection using binary search?
Multiple Term Conjunctive Queries

• $K_1$ AND $K_2$ AND … AND $K_n$
• SvS does not perform well if none of the associated lists are short
• In addition, it is blocking
• Can you design non-blocking multiple sorted array intersection algorithm?
Generalization

• Generalize the 2-way intersection algorithm
• 2-way:
  – \{1, 2\} \rightarrow move k_1’s cursor
  – skipTo(2)
• 3-way:
  – \{1, 2, 3\} \rightarrow move k_1,k_2’s cursor
  – skipTo(3)

\[
\begin{array}{c}
K_1: \begin{pmatrix} 1 \\ 3 \end{pmatrix} \\
K_2: \begin{pmatrix} 2 & 4 & 6 \end{pmatrix} \\
K_3: \begin{pmatrix} 3 & 9 & 27 & 81 \end{pmatrix}
\end{array}
\]

\[\text{eliminator} = \max_{1 \leq i \leq n}(k_i \cdot \text{cursor})\]
Optimization

- Mismatch found even before accessing $K_3$’s cursor
- Choice 1: continue to get cursors of other list
- Choice 2: settle the dispute within the first two lists $\Rightarrow$ max algorithm [Culpepper & Moffat, 2010]
  - Better locality of access $\Rightarrow$ fewer cache misses
  - Similar to SvS
Pseudo-Code for the Max Algorithm (Wrong)

- Input: $K_1, K_2, \ldots, K_n$ in increasing size

1. $x := K_1[1]$; $startAt := 2$ \hspace{1cm} // $x$ is the eliminator
2. while $x$ is defined do
3.   for $i = startAt$ to $n$ do
4.     $y := K_i$.skipTo($x$)
5.     if $y > x$ then \hspace{1cm} // mismatch
6.       $x := K_1$.next() \hspace{1cm} // restart_1 \hspace{1cm} // restart_2
7.     if $y > x$ then $startAt := 1$; $x := y$ else $startAt := 2$ end if
8.     break \hspace{1cm} // match in all lists
9.   elsif $i = n$ then \hspace{1cm} // $y = x$
10.  Output $x$
11.  $x := K_1$.next()
12. end if
13. end for
14. end while
1. Aligned on AB
2. Mismatch on C
3. (L6) Try A.next()
4. $6 < 8 \Rightarrow \text{restart}_1$
   • $x = 8$
   • Align from A, by A.skipTo(x)
1. Aligned on AB
2. Mismatch on C
3. (L6) Try A.next()
4. $9 < 8 \Rightarrow \text{restart}_2$
   - $x = 9$
   - Align from B, by B.skipTo(x)
Pseudo-Code for the **Max** Algorithm (Fixed)

- Input: $K_1, K_2, \ldots, K_n$ in increasing size

(1) $x := K_1[1]; \ startAt := 2$

(2) while $x$ is defined do

(3) for $i = startAt$ to $n$ do

(4) $y := K_i$.skipTo($x$)

(5) if $y > x$ then

(6) $x := K_1$.next()

(4.1) if $i = 1$ then

(4.2) if $y > x$ then

(4.3) $x := y$

(4.4) break

(4.5) end if

(4.6) end if

(7) if $y > x$ then $startAt := 1; x := y$ else $startAt := 2$ end if

(8) break

(9) elseif $i = n$ then

(10) Output $x$

(11) $x := K_1$.next()

(12) end if

(13) end for

(14) end while
References


• Stefan Buettcher, Charles L. A. Clarke, Gordon V. Cormack, Information Retrieval: Implementing and Evaluating Search Engines, 2010 [Chapter 5]