Analyzing the performance of top-k retrieval algorithms

Marcus Fontoura
Google, Inc

This talk

- Largely based on the paper
  - Evaluation Strategies for Top-k Queries over Memory-Resident Inverted Indices, VLDB 2011

- No Google-specific data or algorithm!

Goal

- Highlight the parameters used to characterize the performance of retrieval systems

- Analysis of a few top-k algorithms

Outline

- Problem representation
- DAAT approaches
- TAAT approaches
- Hybrid approaches
- Conclusion

Top-k Query Evaluation

- Given a query *Q* and a document corpus *D* return the *k* documents that have the highest score according to some scoring function *score(d, Q)*
- Scoring is based on intersecting the *terms* in the query with the documents
- Query evaluation cost =

  Index access cost +

  Score computation cost

Memory Resident Indices

- Many applications need very low latency and very high throughput
  - Cannot tolerate even a single disk seek
- Disk access kills both latency and throughput
- Caching is not effective in the presence of real time updates
- No previous study on DAAT vs TAAT on memory resident indices

Dot Product Scoring Function

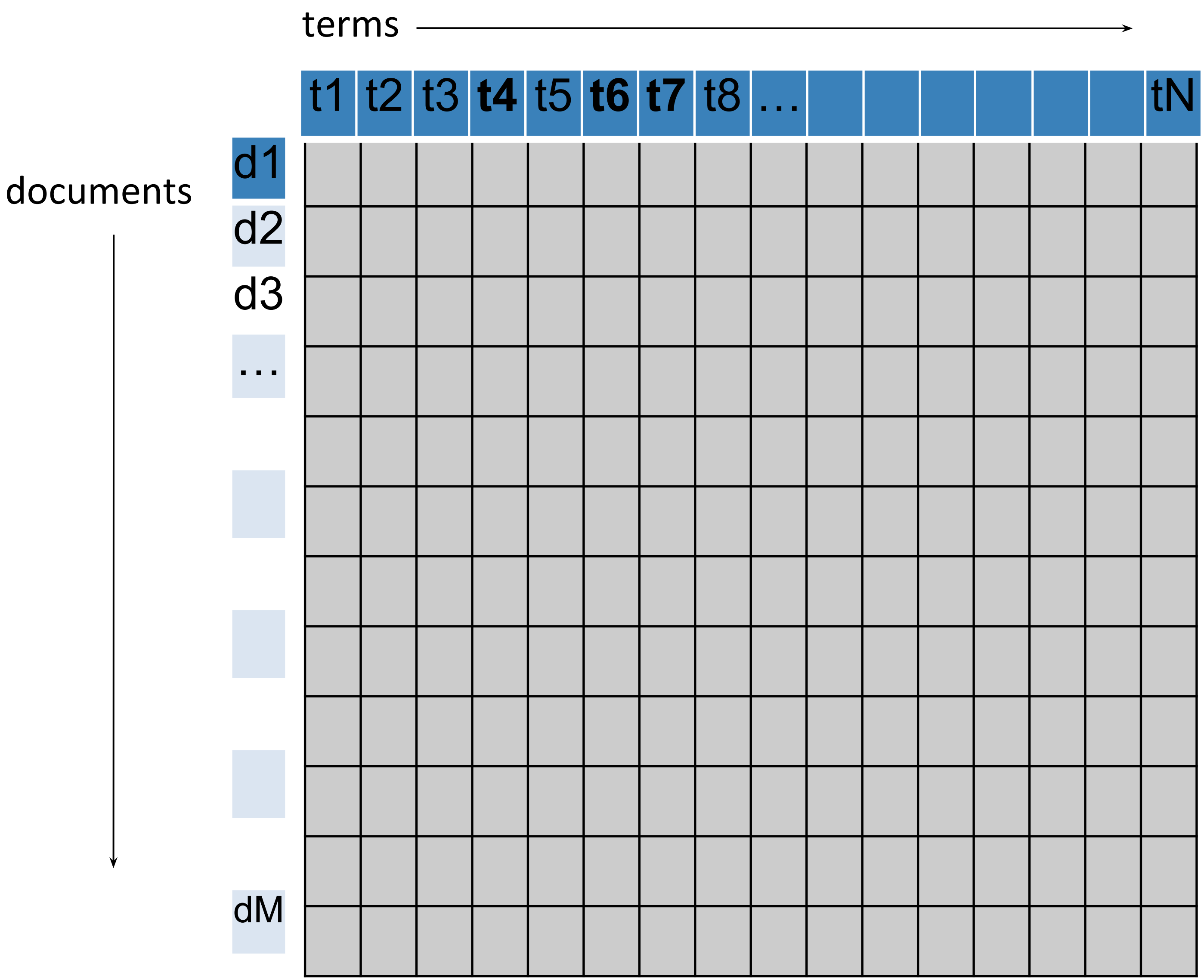
Document $d = \{d_1 \ldots d_N\}$

Query $Q = \{q_1 \ldots q_N\}$

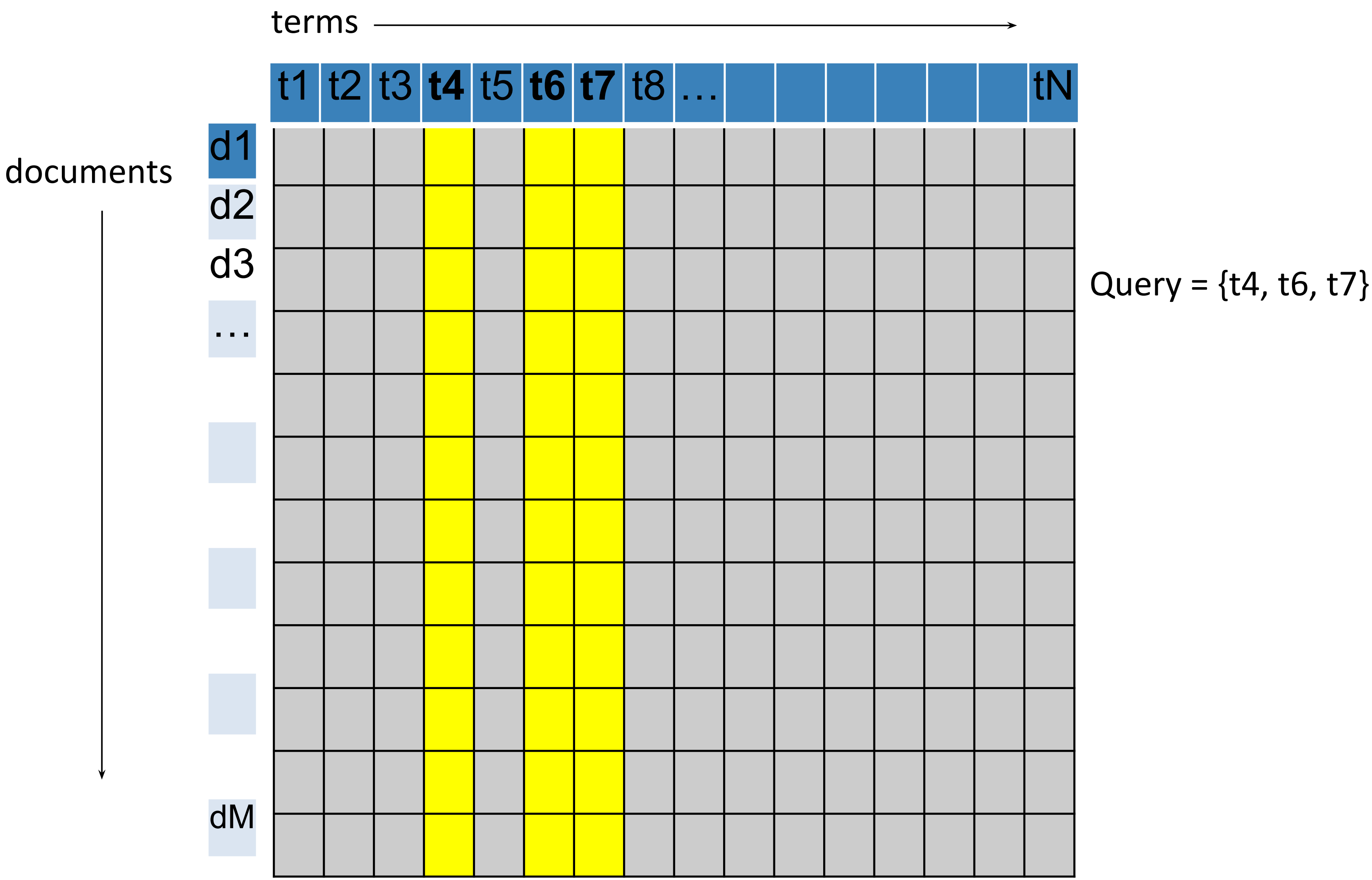
Score $(d, Q) = \sum_{i=1}^{N} (d_i \, q_i)$

The document and query weights could be derived from standard IR techniques, such as TFIDF, language models, etc

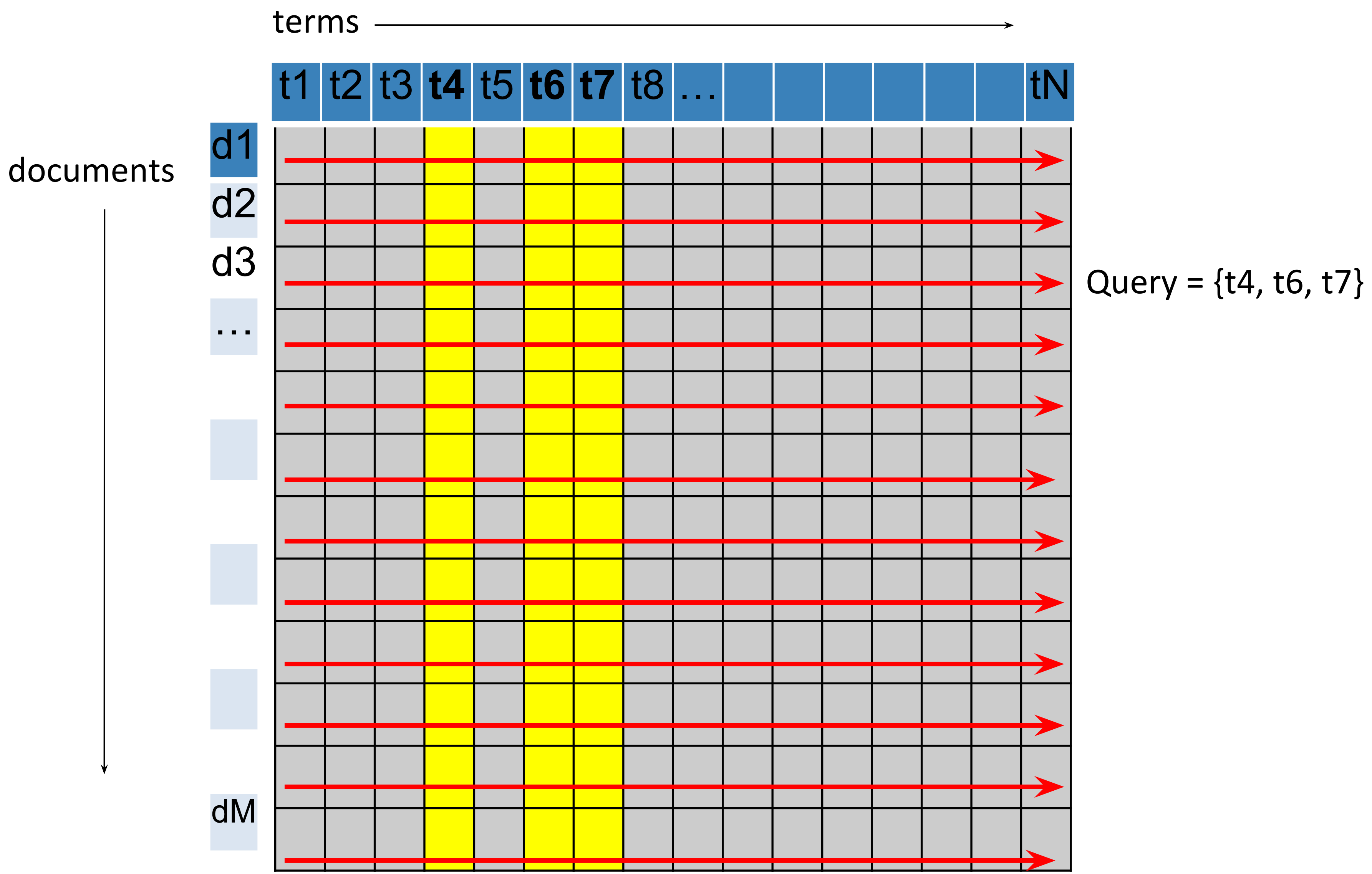
Document Corpus Matrix



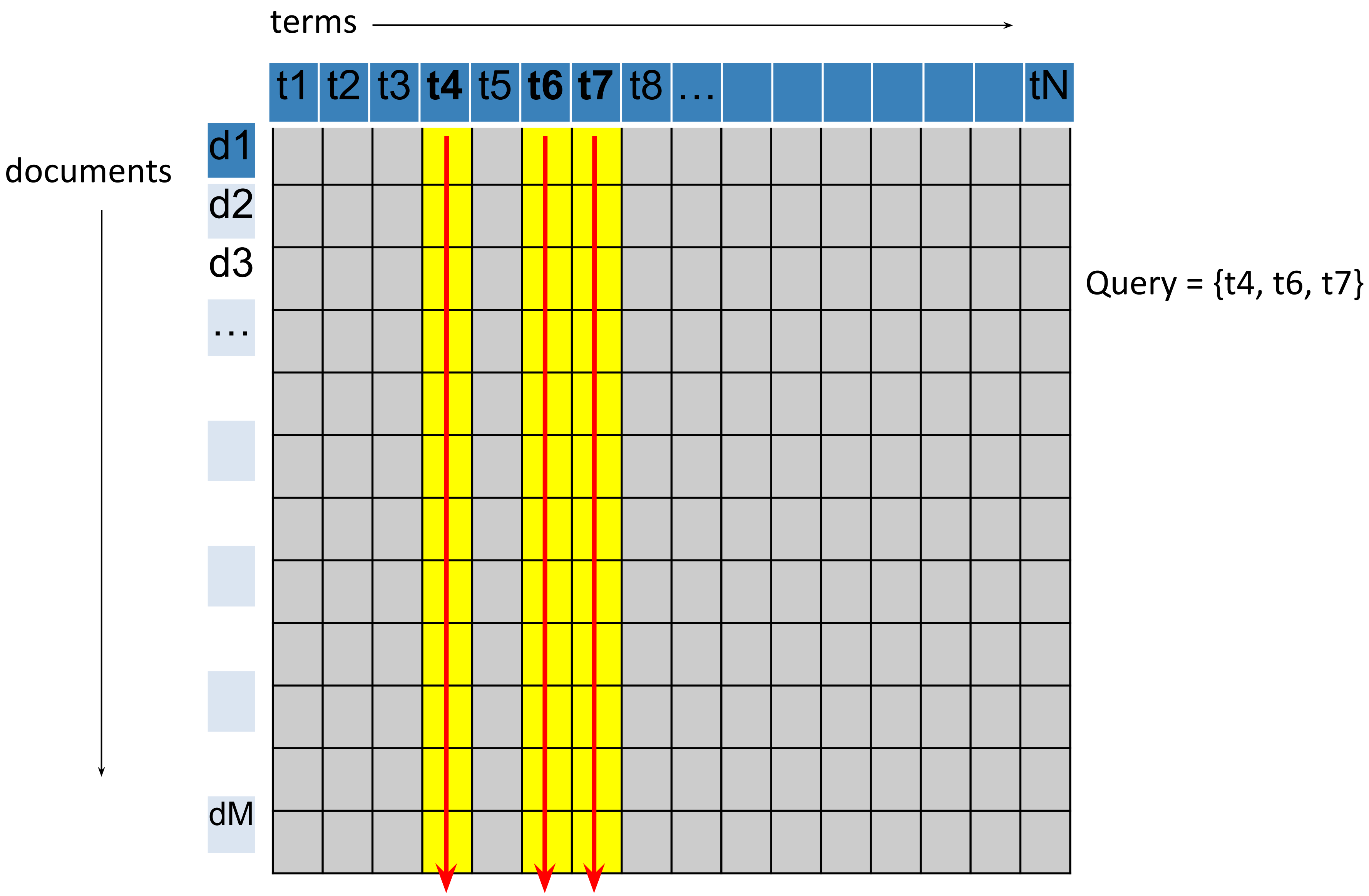
Document Corpus Matrix



DAAT (Document-at-a-time)



TAAT (Term-at-a-time)



Document Corpus Representation

- Document corpus is a sparse matrix representation
- Represent the document corpus matrix using posting lists
- Each term has list of documents and metadata
- Posting List Entry has: <**DocumentID**, WeightOfTermInDocument>

t4	t6	t7
<**1**, 3>	<**1**, 4>	<**1**, 6>
<**2**, 4>	<**2**, 3>	<**2**, 7>
<**10**, 2>	<**7**, 2>	<**5**, 1>
	<**8**, 5>	<**6**, 7>
	<**9**, 2>	<**10**, 1>
	<**11**, 5>	<**11**, 7>

Query = {t4, t6, t7}

Cursor

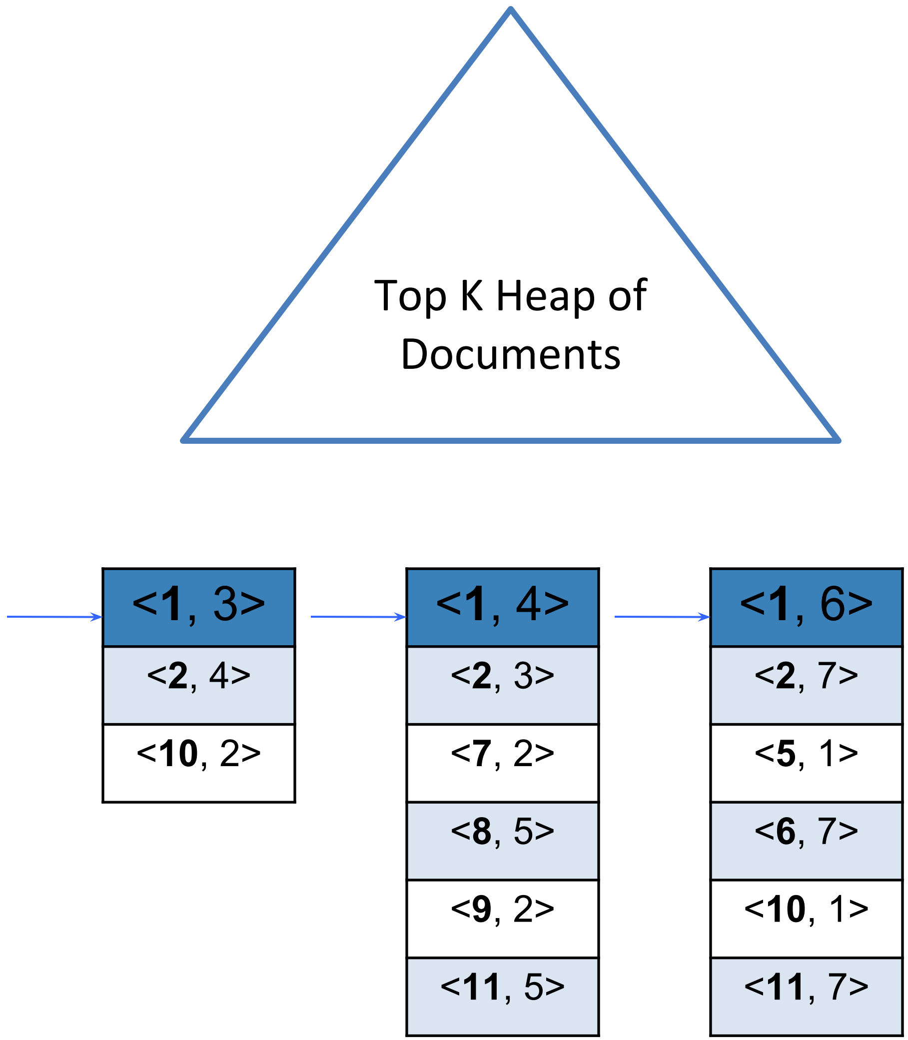
- Cursor a pointer into a posting list

- Important cursor operations
  - C_t*.next()* // *move to next posting*
  - C_t*.fwdBeyond(docid d)* // *move to posting with*
    // *docid >= d*

DAAT Algorithms - Naive

- Use a min-heap maintaining the top k candidates
- Let θ be the min score on heap
- Use N-way merge to compute score of each document and insert it into heap if score > θ
- Every posting for every query term is touched
  - Index access cost is proportional to sum of sizes of postings list of all query terms.
- All documents containing any of the query terms are scored
  - Scoring cost is proportional to the number of documents scored

DAAT Algorithms - Naive



DAAT Algorithms - WAND

- Compute upper bound contribution of each query term:

$$UB_t = D_t q_t$$

- Sort the term cursors by its current document and identify a pivot term *p* such that:

$$\sum_{1 \le t \le p} UB_t > \theta$$

- Upper bounds of cursors including this pivot could enter top k

DAAT Algorithms - WAND

- The current document for the pivot term is the next possible candidate to score
- If all the cursors before pivot point to the pivot document, score it otherwise pick a term before pivot and move it beyond pivot document
- After each cursor move the terms are resorted and pivot selection is continued

DAAT Algorithms - WAND

- Compute upper bound contribution of each query term
  $UB_t = D_t q_t$

A	B	C
UB_A = 4	**UB_B = 5**	**UB_c = 7**
<1, 3>	<1, 4>	<1, 6>
<2, **4**>	<2, 3>	<2, 7>
<10, 2>	<7, 2>	<5, 1>
	<8, **5**>	<6, **7**>
	<9, 2>	<10, 1>
	<11, **5**>	<11, 7>

DAAT Algorithms - WAND

- Sort the term cursors by its current document and identify a pivot term *p* such that $\sum_{1 \le t \le p} UB_t > \theta$

A	B	C
$UB_A = 4$	$UB_B = 5$	$UB_c = 7$
<1, 3>	<1, 4>	<1, 6>
<2, **4**>	<2, 3>	<2, 7>
<10, 2>	<7, 2>	<5, 1>
	<8, **5**>	<6, **7**>
	<9, 2>	<10, 1>
	<11, **5**>	<11, 7>

Sorted Cursors

	C	B	A
docid	5	7	10

pivot term

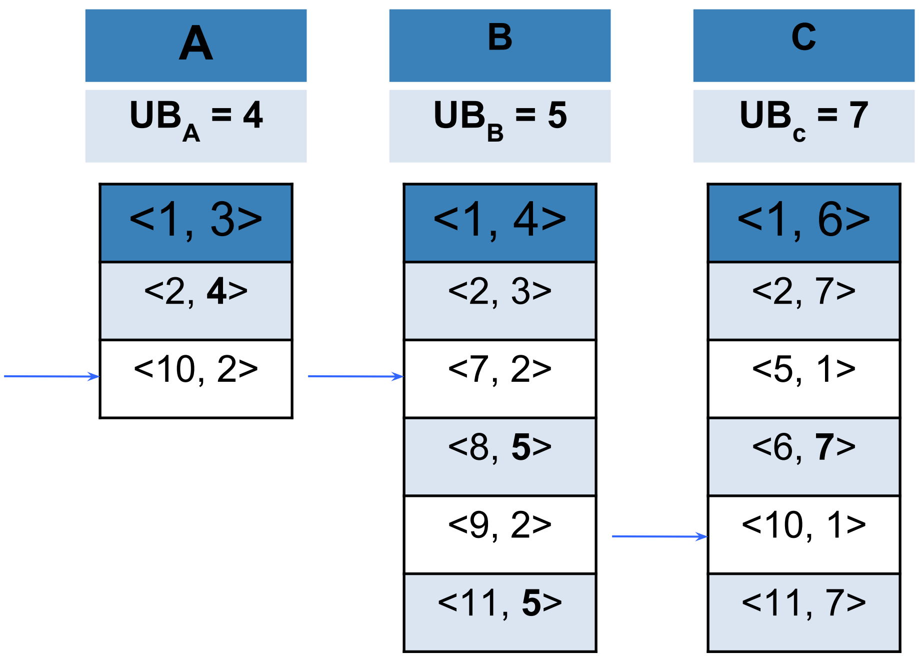
7+5+4 > 13 (θ)

Top K Heap

docid	Score(d, Q)
1	13(θ)
2	14

DAAT Algorithms - WAND

- If all the cursors are before pivot point to the pivot document, score it, otherwise pick a term before pivot and move it beyond pivot document
- After each cursor move the terms are resorted and pivot selection is continued



Sorted Cursors			
	B	**C**	**A**
docid	7	10	10

pivot term

Top K Heap	
docid	**Score(d, Q)**
1	13(θ)
2	14

DAAT Algorithms - mWAND

- Traditional WAND picks one term at a time to move to/ahead of the pivot document
  - This reduces potential disk I/O
  - Optimizes for reducing index access at the expense of doing more pivot selections

- mWAND – for memory resident indices, index access is less significant. Hence we propose a variation to move all terms between *1* and *p* beyond the pivot document.
  - Increases cost of index access
  - Minimize the number of pivot selections

DAAT Algorithms - mWAND

A	B	C
$UB_A = 4$	$UB_B = 5$	$UB_C = 7$
<1, 3>	<1, 4>	<1, 6>
<2, 4>	<2, 3>	<2, 7>
<10, 2>	<7, 2>	<5, 1>
	<8, 5>	<6, 7>
	<9, 2>	<10, 1>
	<11, 5>	<11, 7>

Heap	
docid	**Score(d, Q)**
1	13(θ)
2	14

Sorted Cursors			
	C	**B**	**A**
docid	5	7	10

pivot term

WAND – May pick term B **or** C to move to beyond pivot doc id 10.

Sorted Cursors			
	C	**A**	**B**
docid	5	10	11

mWAND – Moves **both** B **and** C beyond pivot doc id 10.

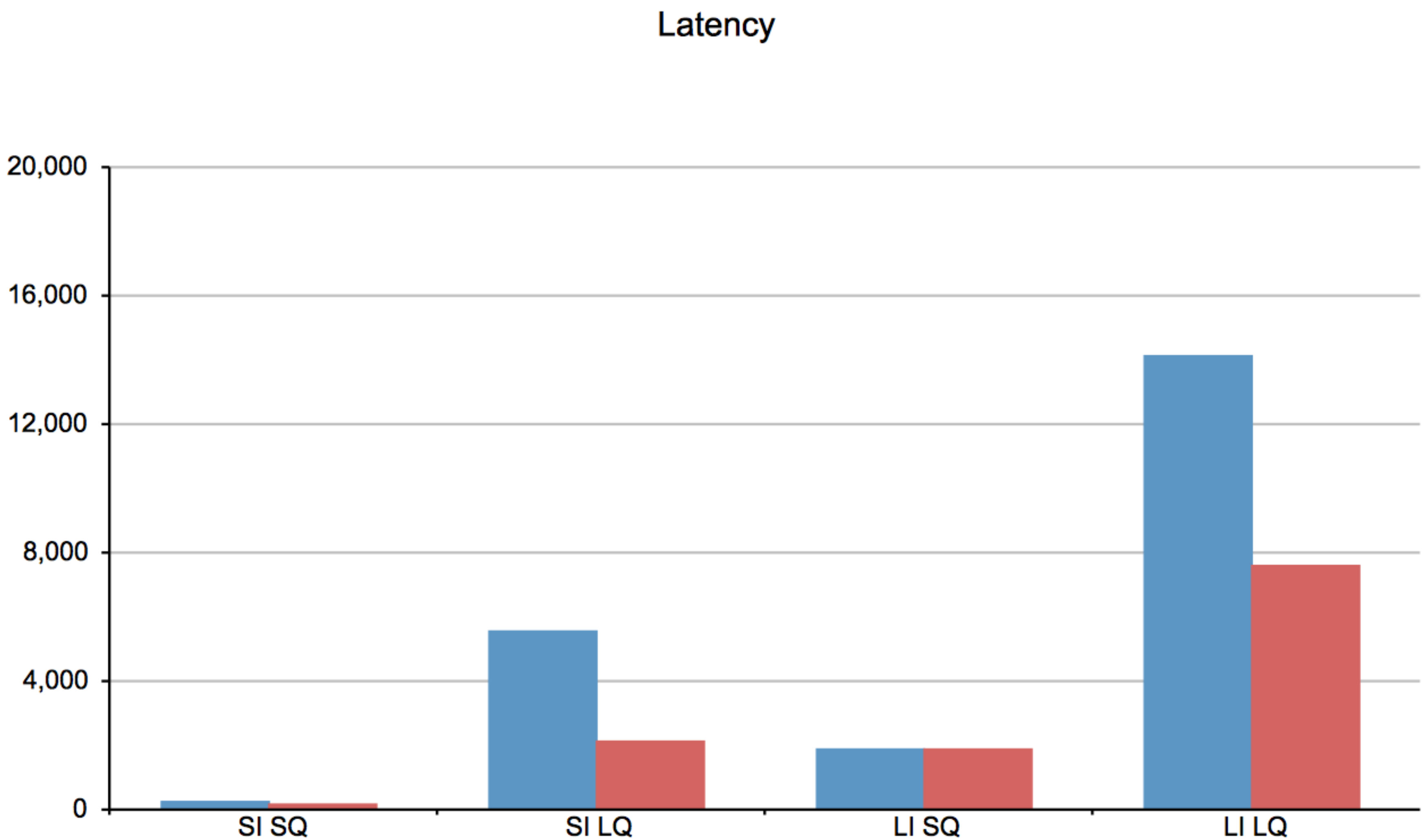
Sorted Cursors			
	C	**A**	**B**
docid	10	10	11

Dataset

- S = Small
- L = Large
- I = Index
- Q = Query

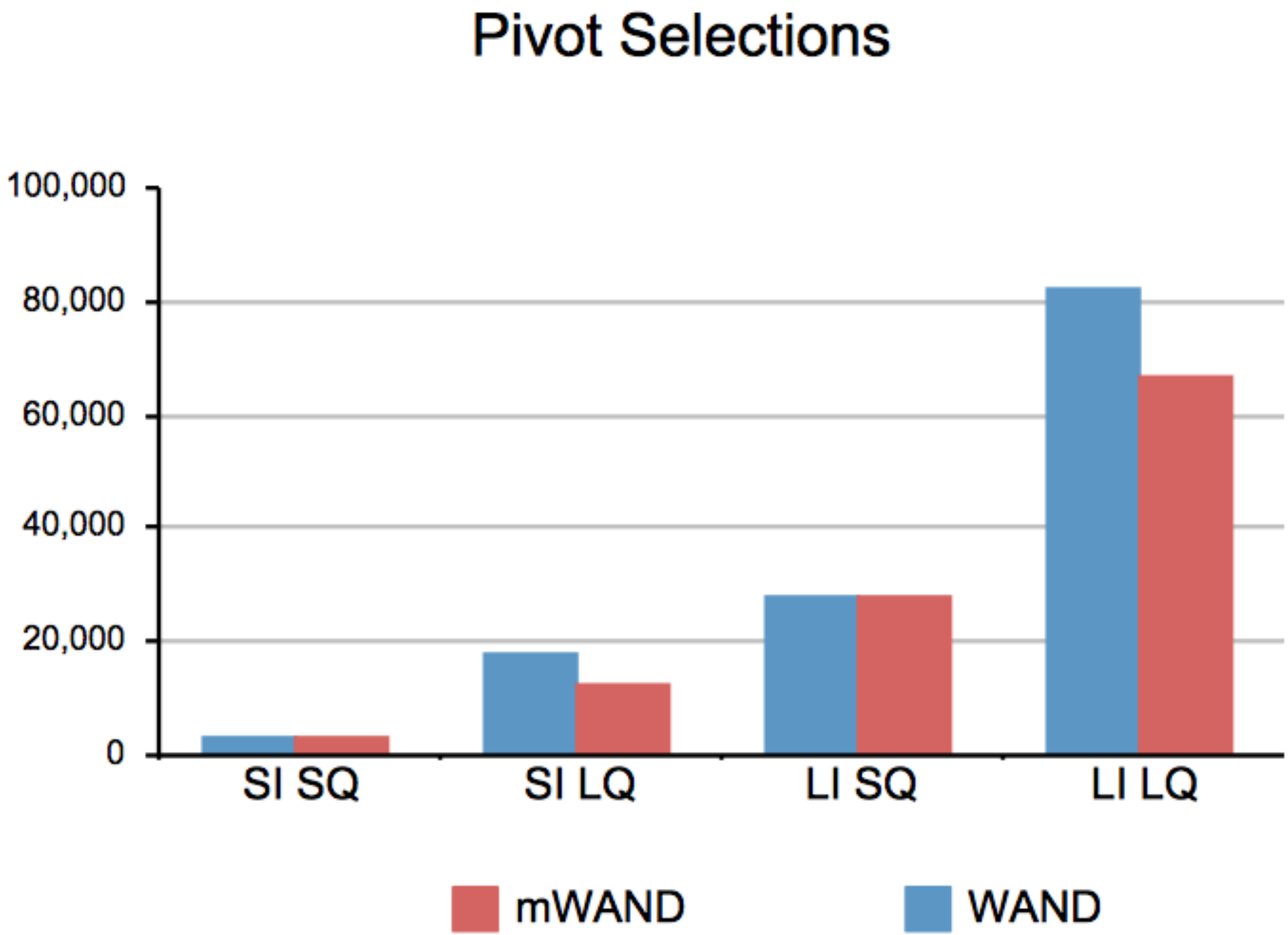
- Example: SI LQ means small index, large (many terms per query) query set
  - Other combinations left as an exercise for the interested reader
- Full description of dataset characteristics in the paper

WAND vs mWAND

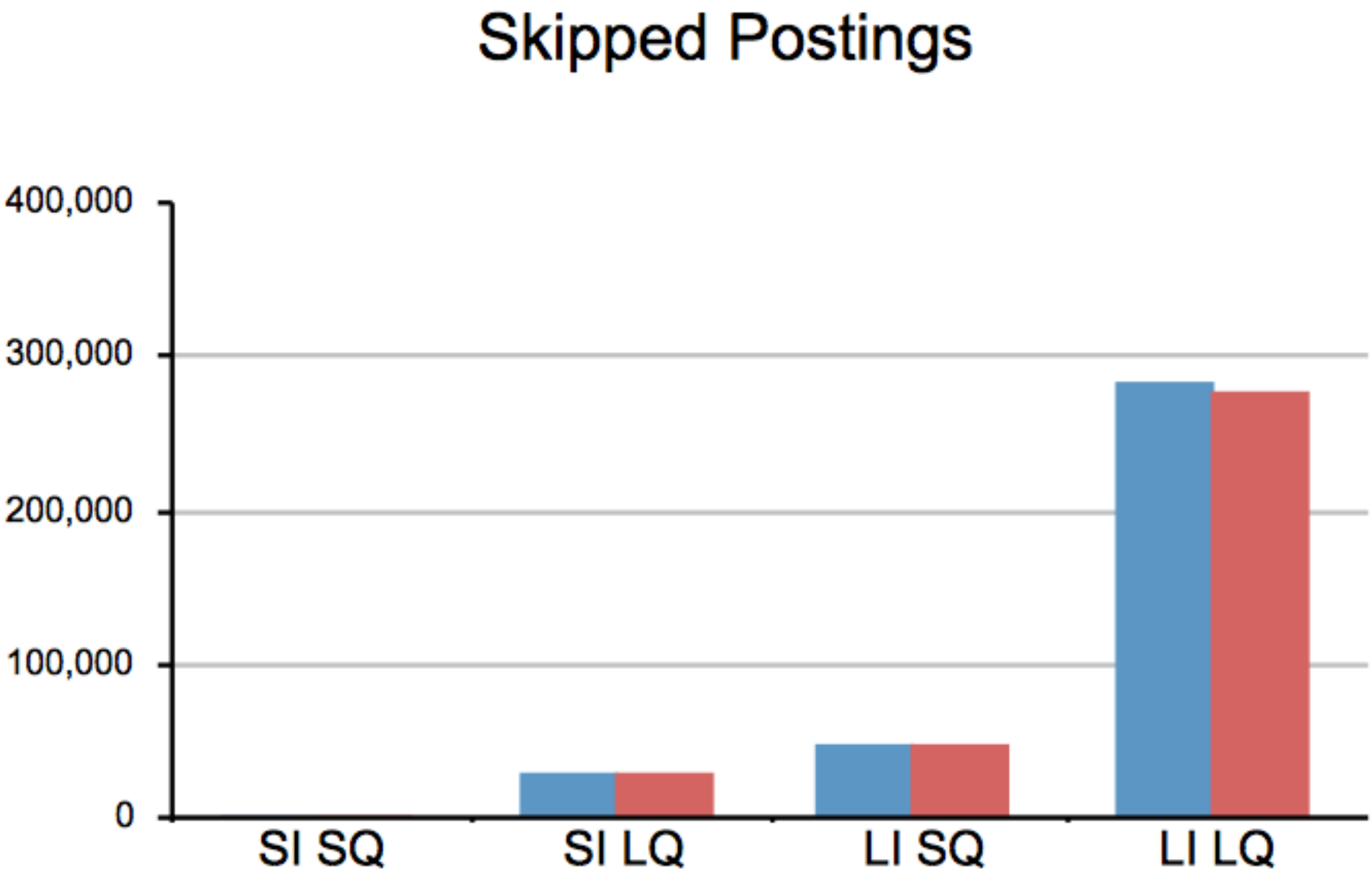


mWAND (red) is 2x faster than WAND (blue)

WAND vs mWAND



WAND vs mWAND



DAAT Algorithms – max_score
(Turtle &Flood)

- Sort the term cursors by the size of their posting list (only once)
- Maintain remaining upper bounds *RUB* for each term such that

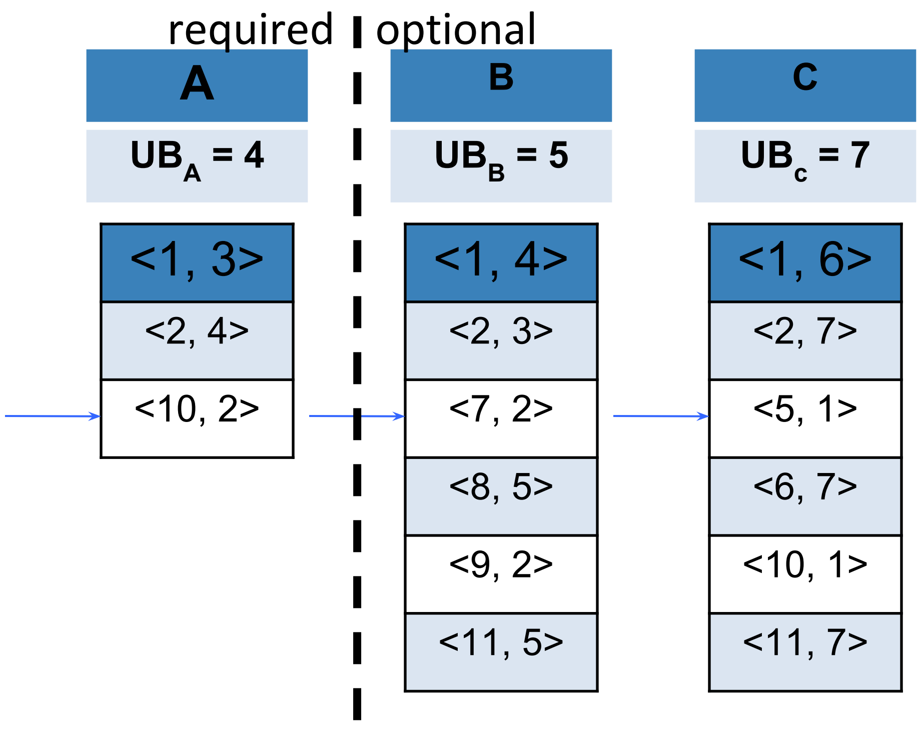
$$RUB_t = \sum_{t < i \leq N} UB_i$$

- Split the terms into two groups required and optional. The optional group is the set of terms from C_k through C_N such that these terms are not enough to allow a document into the top k

$$C_k = argmax_k \sum_{N \geq i > k} UB_i < \theta$$

- Evaluate the terms in required group in a naïve manner, but skip evaluating documents whose current cumulative score after evaluating cursor C_t, having $Score_t + UB_t < \theta$ (infeasible documents)

DAAT Algorithms – max_score



Heap

docid	Score(d, Q)
1	13(θ)
2	14

Sorted Cursors

	A	B	C
docid	10	7	5
UB	4	5	7
RUB	12	7	0

Evaluate required: Payload C_A *(2)+* RUB_A *(5+7=12)* $> \theta$ *(13)*

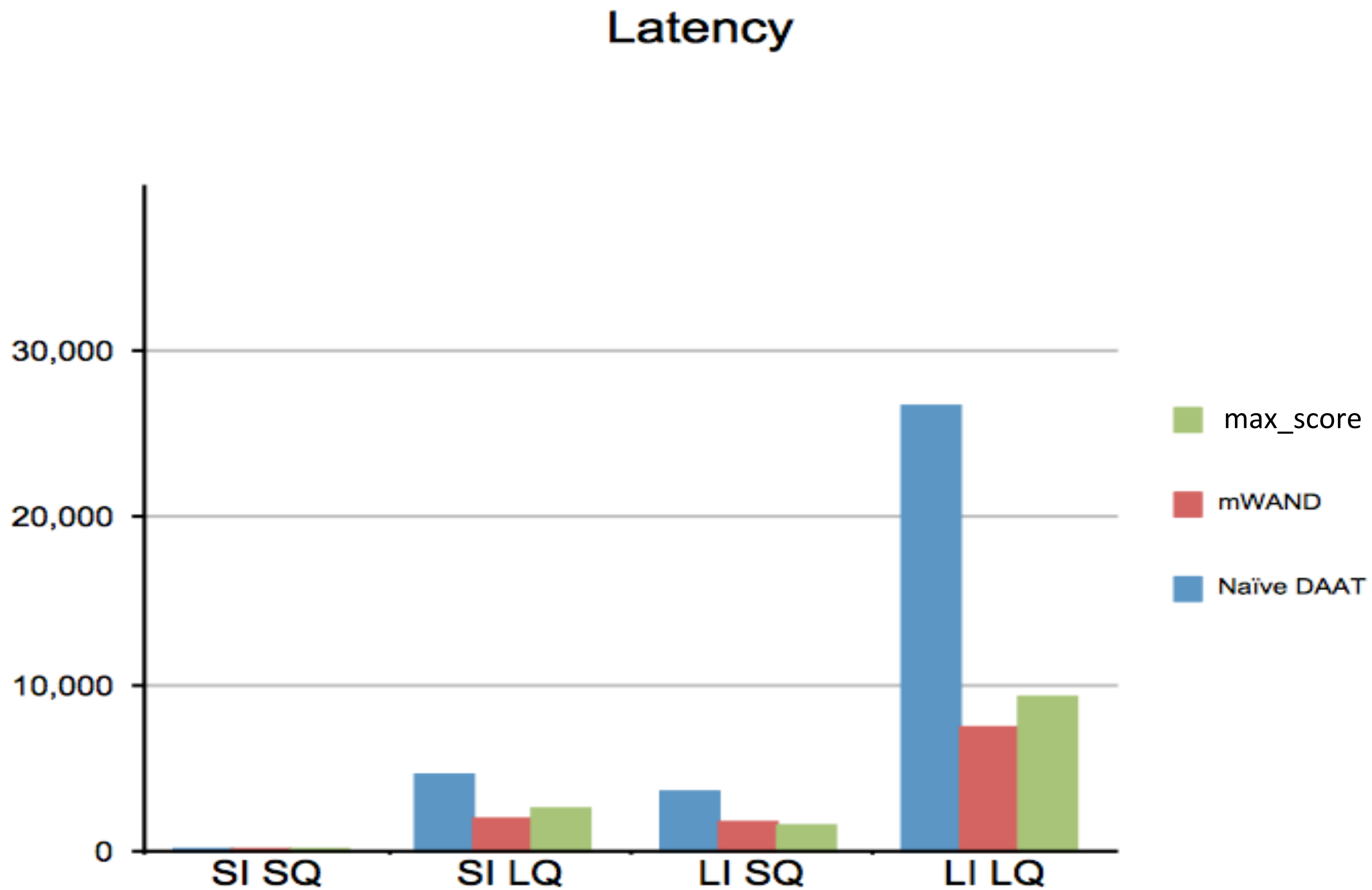
Move optional: Move to doc 10 or beyond on C_B and C_C

and score doc 10.

Split Cursors

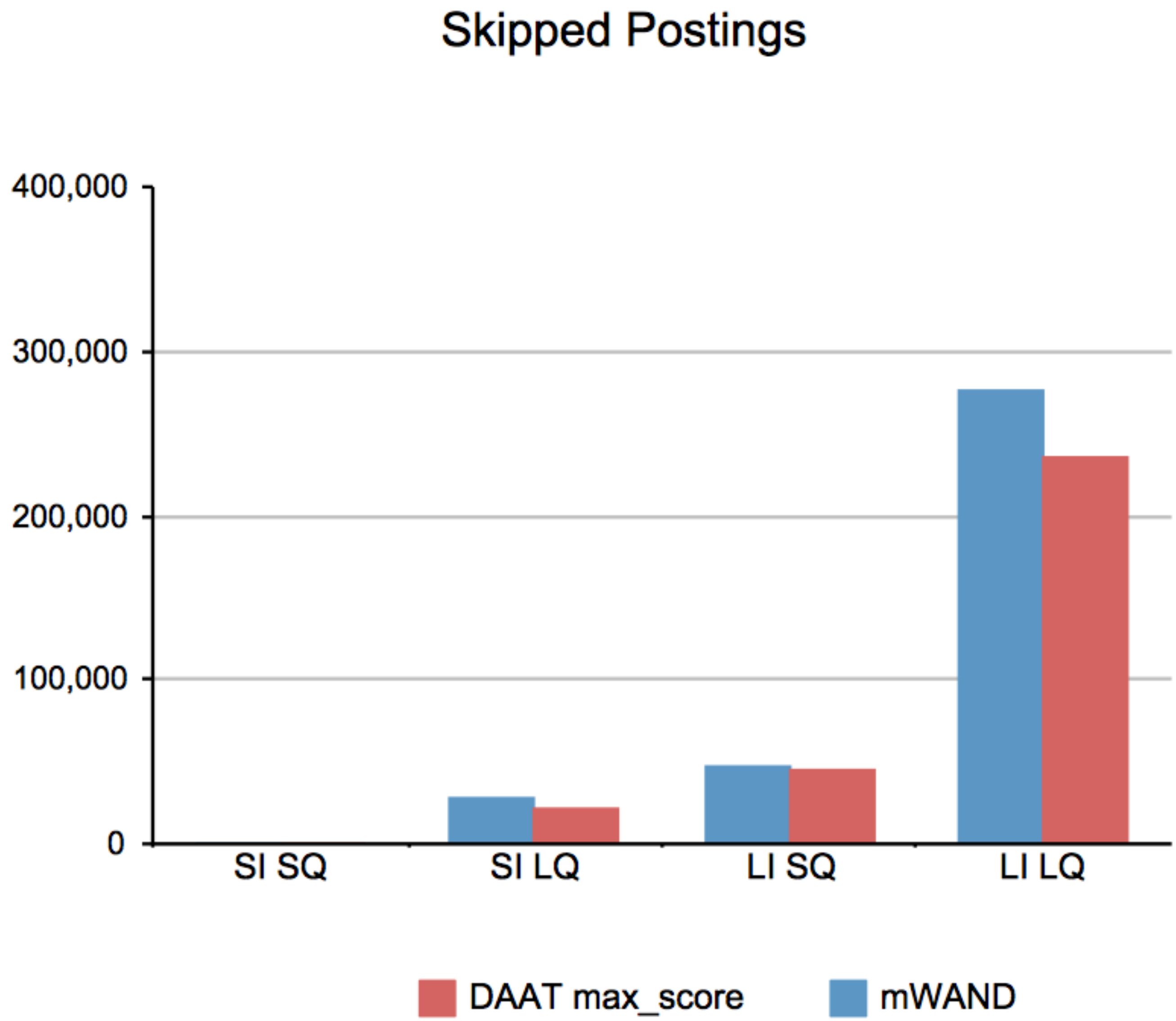
	A	B	C
docid	10	7	5
UB	4	5	7
RUB	12	7	0

Comparison of DAAT Algorithms



- mWAND and DAAT max_score both substantially better than Naïve DAAT
- For LI LQ data, mWAND is 23% faster than DAAT max_score

Comparison of DAAT Algorithms



Comparison of DAAT Algorithms

- mWAND always skips more postings
- For small queries more complex code for finding the pivot does not payoff

TAAT Algorithms - Naive

- Query terms are evaluated one at a time
- An accumulator array A to used to keep track of the partial scores of each document
- Once all terms are evaluated, the top-k documents from the accumulator array are returned
- Every posting for every query term is touched
  - Index access cost is proportional to sum of sizes of postings list of all query terms
- All documents containing any of the query terms are scored
  - Scoring cost is proportional to the number of documents scored

TAAT Algorithms – Buckley & Lewit

- Query terms are evaluated one at a time in decreasing order of upper bounds
- A min heap of size k+1 is maintained having the documents with the highest score so far
- After processing the i^{th} term, the query processing could be terminated if the following condition is met:

$$A[k] \geq A[k+1] + \sum_{t>i} UB_t$$

- If the k^{th} document's score is greater than $k+1^{th}$ document's score by more than sum of the remaining terms' upper bound, then we have found the top-k documents

TAAT Algorithms – Buckley & Lewit

A	B	C
$UB_A = 9$	$UB_B = 7$	$UB_C = 4$
<1, 3>	<1, 5>	<1, 4>
<4, 9>	<2, 1>	<4, 1>
<7, 3>	<4, 7>	<5, 2>
<10, 2>		<6, 2>
		<10, 1>

Accumulator array at each iteration

i	*docid*	A[1]	A[2]	A[4]	A[5]	A[6]	A[7]	A[10]
1	1	3	0	0	0	0	0	0
1	4	3	0	9	0	0	0	0
1	7	3	0	9	0	0	3	0
1	10	3	0	9	0	0	3	2
2	1	8	0	9	0	0	3	2
2	2	8	1	9	0	0	3	2
2	4	8	1	16	0	0	3	2

$$A[1] = 8 \geq A[7] + \sum_{i>2} UB_i = 3 + 4$$

TAAT Algorithms – TAAT max_score (Turtle & Flood)

- Query terms are evaluated one at a time in decreasing order of postings list sizes.
- Phase 1: Continue processing terms until the following condition is met (kth document is better than sum of all unprocessed term upper bounds)

$$A[k] > \sum_{t>i} UB_t$$

- After phase 1, there could be no documents in top-k that are not already present in the accumulator array
- Phase 2: Obtain exact scores by score only documents found in phase 1 for the rest of the terms
  - Need to sort list of documents from phase 1 – *candidate list.*
  - Pruning the candidate list: Document *d* can pruned (if infeasible) during phase 1 if the following holds (its score + all unprocessed terms is less than the kth best)

TAAT Algorithms – TAAT max_score

A	B	C
$UB_A = 9$	$UB_B = 7$	$UB_C = 4$
<1, 3>	<1, 5>	<1, 4>
<4, 9>	<2, 1>	<4, 1>
<7, 3>	<4, 7>	<5, 2>
<10, 2>		<6, 2>
		<10, 1>

Accumulator array at each iteration

i	*docid*	A[1]	A[2]	A[4]	A[5]	A[6]	A[7]	A[10]
1	1	5	0	0	0	0	0	0
1	2	5	1	0	0	0	0	0
1	4	5	1	7	0	0	0	0
1	1	8	1	7	0	0	0	0
2	4	8	1	16	0	0	0	0
2	7	8	1	16	0	0	3	0
2	10	8	1	16	0	0	3	2

$$A[1] = 8 > \sum_{i>2} UB_i = 4$$

Candidate list: 1, 2, 4, 7, 10

Pruned Candidate list: 1, 4

$$A[2] + \sum_{i>2} UB_i = 1 + 4 = 5 < (A[k] = A[1] = 8)$$

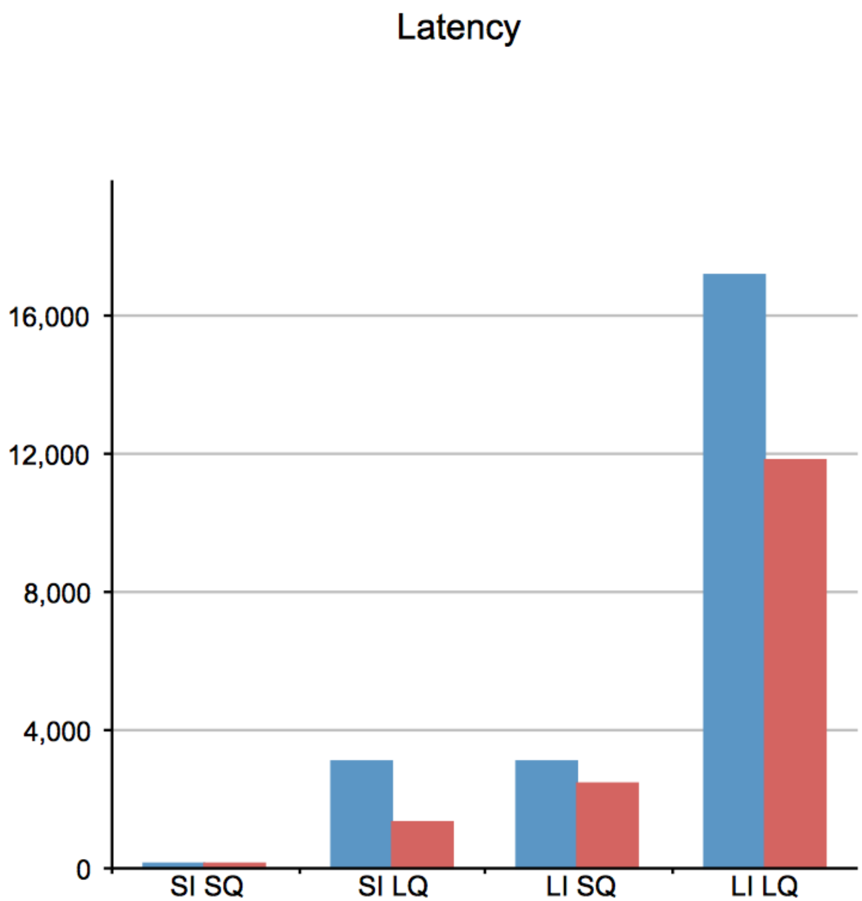
$$A[7] + \sum_{i>2} UB_i = 3 + 4 = 7 < (A[k] = A[1] = 8)$$

$$A[10] + \sum_{i>2} UB_i = 2 + 4 = 6 < (A[k] = A[1] = 8)$$

mTAATmax_score

- Traditional TAAT max_score designed to reduce disk I/O
  - Minimize cursor movements in 2nd phase using the candidate list to help skipping documents
  - Candidate list in phase 1 has to be sorted.
  - Pruning the candidate list to reduce the number of documents to sort.
- Index access is not significantly expensive in memory resident indices.
  - **In many cases sequential read and filter is faster than sort and skip**
  - Hardware prefetching makes **sequential scans very fast**
- Pruning the candidate list requires additional computation and branching instructions.
  - Branch mis-predictions are very expensive in pipelined architectures.
- mTAAT max_score – same as TAAT max_score except:
  - No candidate pruning
  - Phase 2 – no sorting of phase 1 docs: do sequential scan of nonzero phase 1 documents to drive scoring on remaining terms

TAAT max_score vs mTAAT max_score

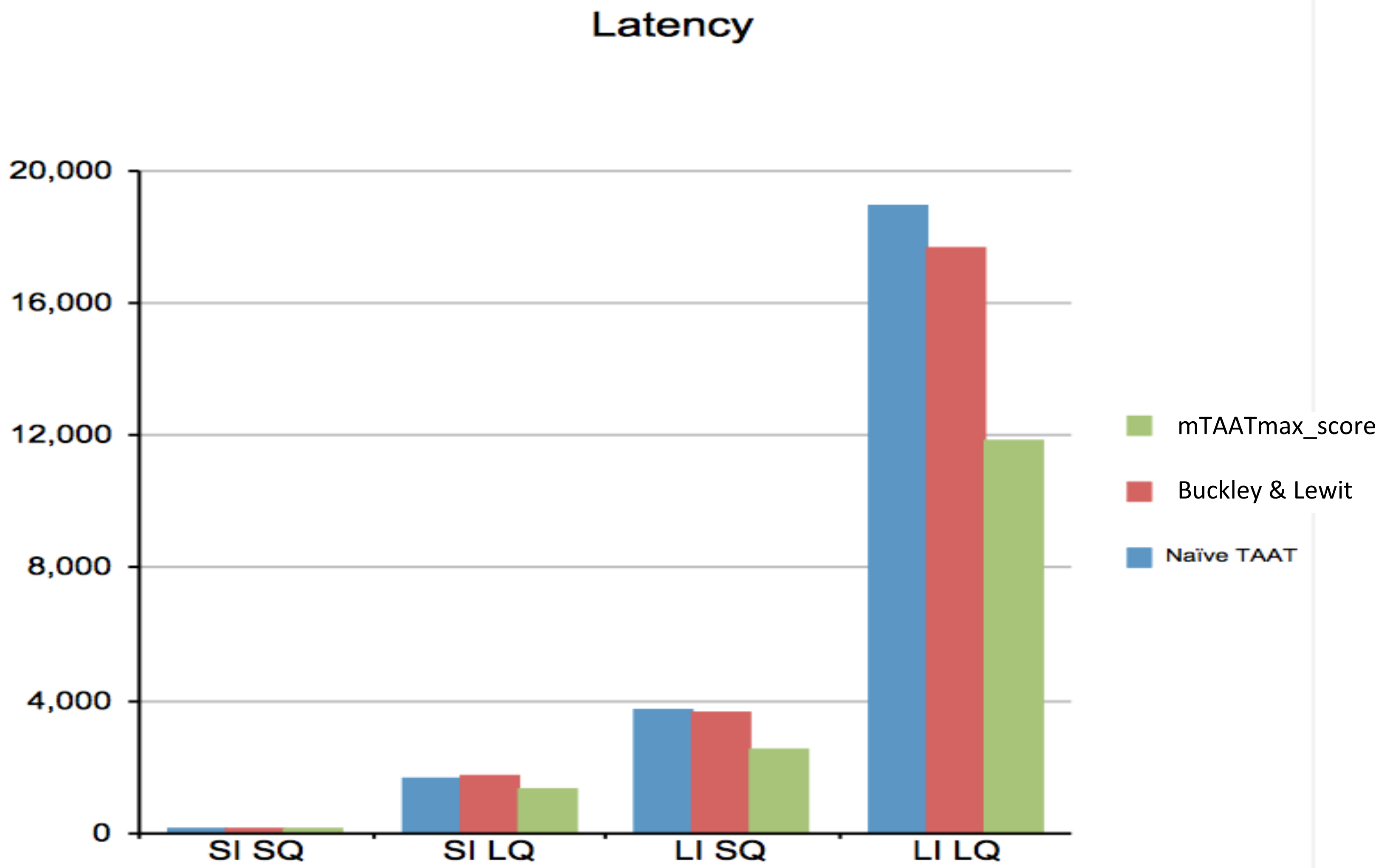


#terms to evaluate in 2nd phase		
	SQ	LQ
SI	0.13	3.44
LI	0.48	3.66

- The number of terms to evaluate in 2nd phase is too little to justify the overhead of maintaining a sorted candidate list.

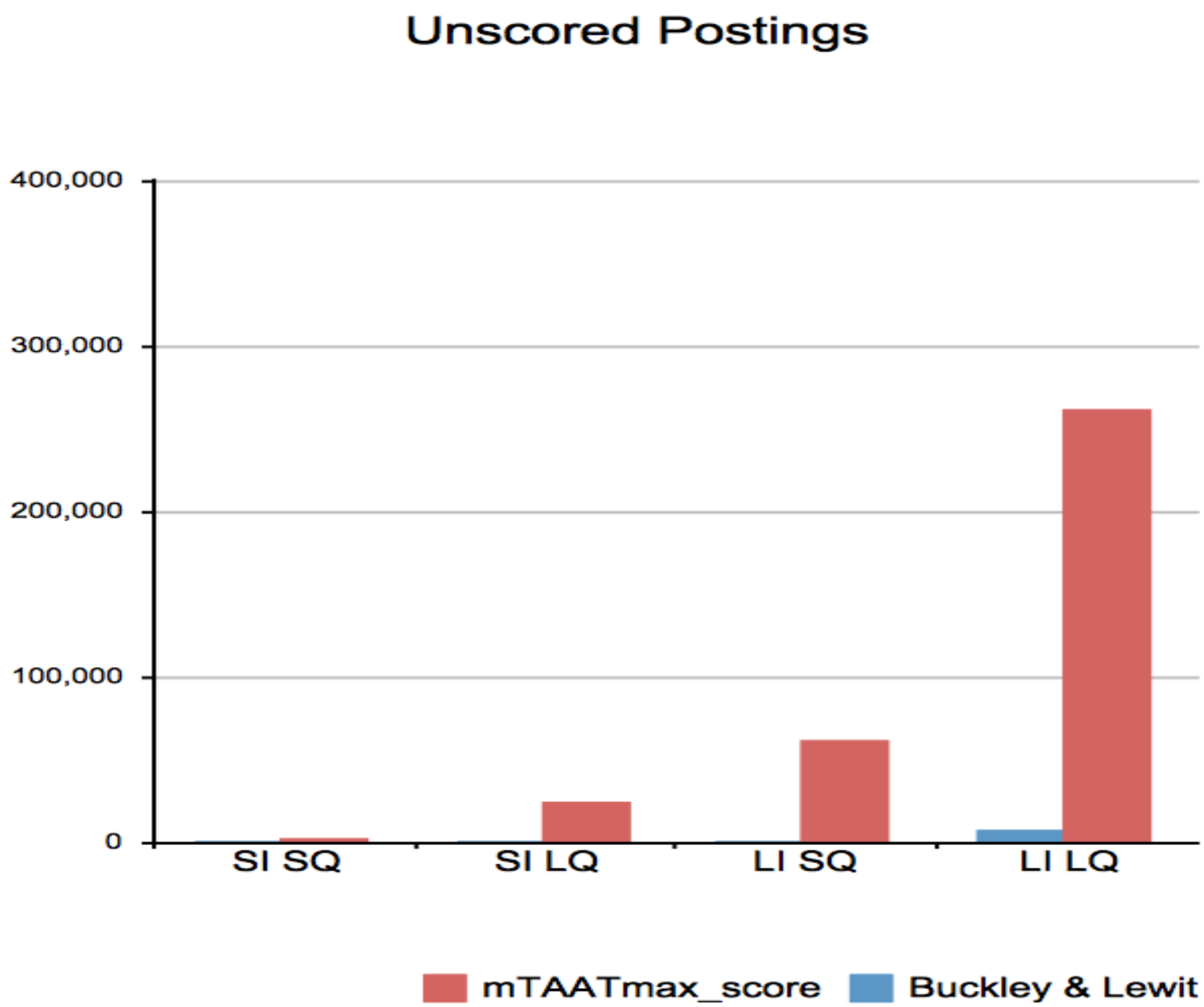
- mTAATmax_score (red) is 46% faster for LI LQ test

Comparison of TAAT Algorithms

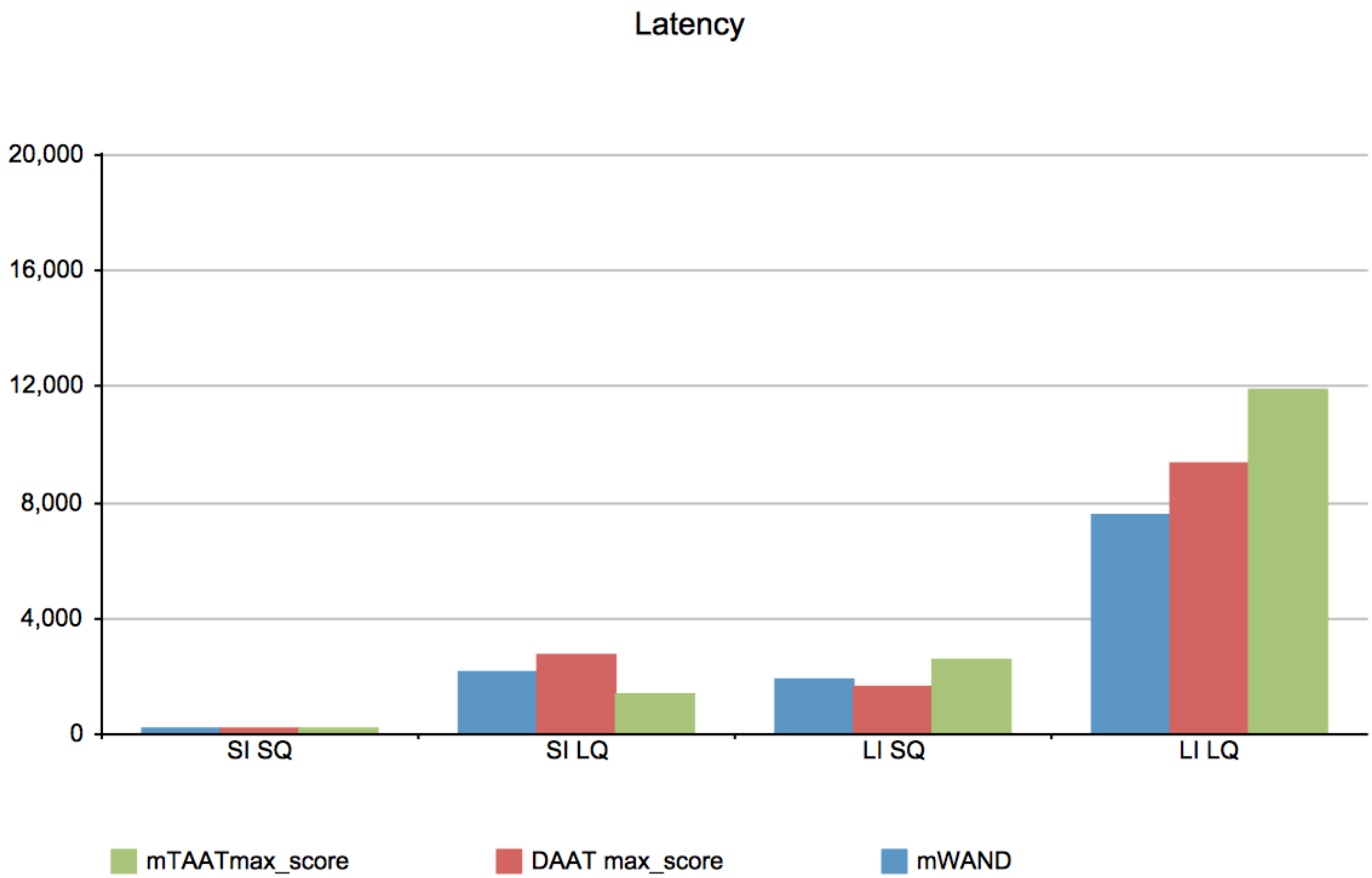


- mTAATmax_score 49% faster than Buckley & Lewit for LI LQ test

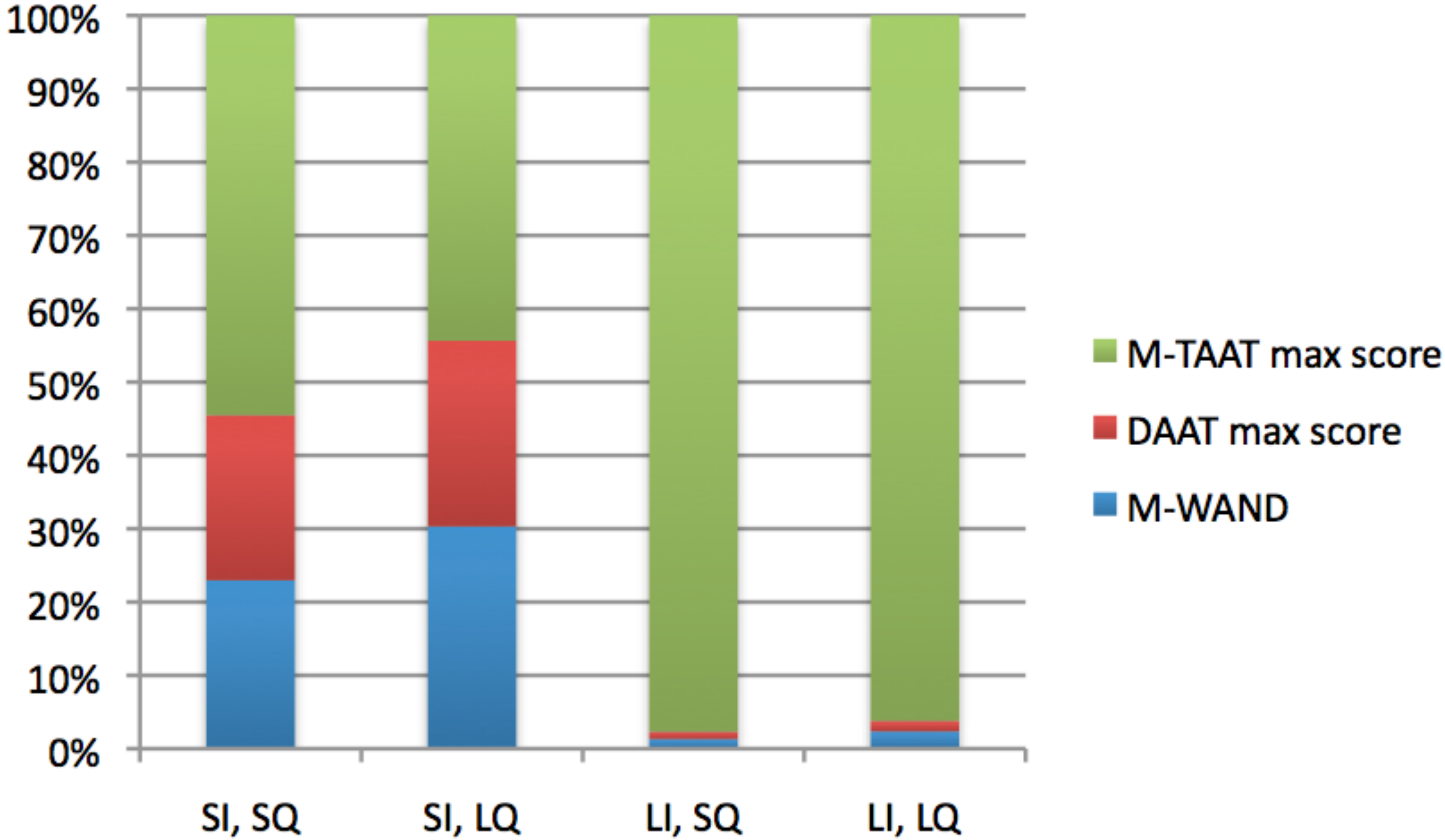
Comparison of TAAT Algorithms



Comparing DAAT and TAAT Algorithms



Cache misses



Hybrid Algorithms

Intuition: It's very fast to process small posting lists and groups of small posting lists. Use this for better lower bounds on θ (min score for candidate docs)

- Split the query terms into two groups – short, and long based on number of postings for each query term and a configurable threshold

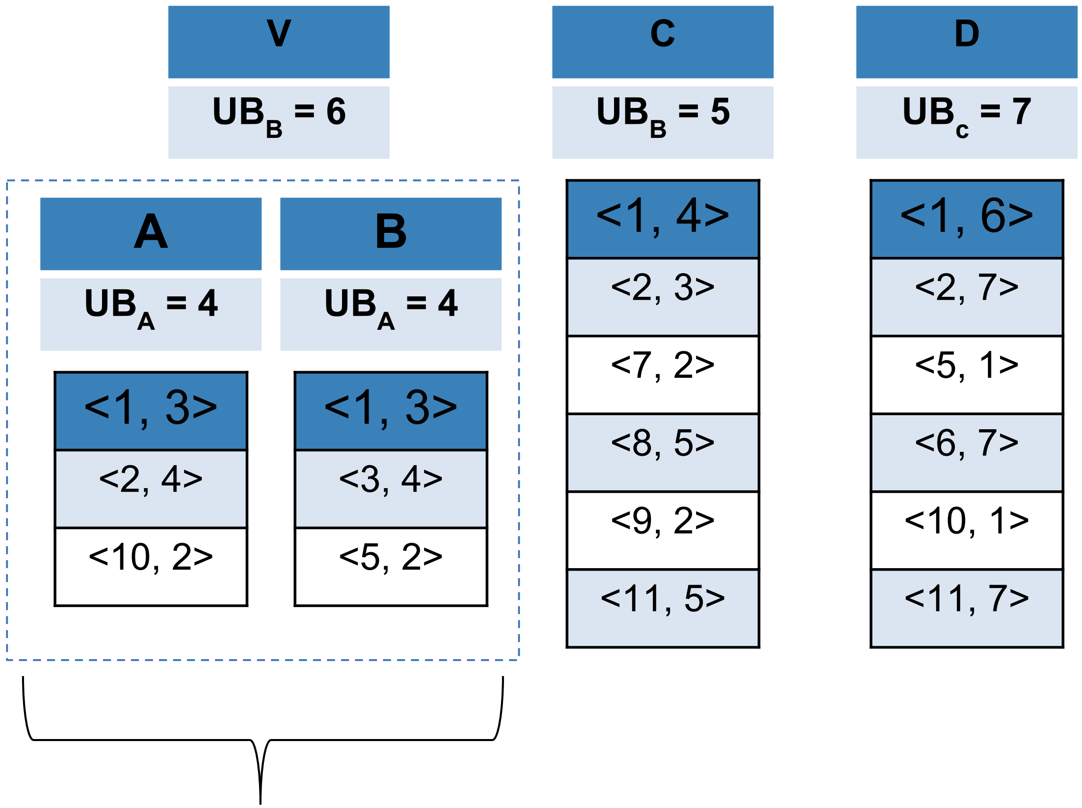
$$Q = Q_{t \leq T} \cup Q_{t > T}$$

- Evaluate $Q_{t \leq T}$ group using any of the TAAT or DAAT algorithms
- Use the partial score of the k^{th} element as the lower bound θ when processing the $Q_{t>T}$ group
- A new virtual or real posting list is created which has all the documents evaluated for $Q_{t \leq T}$ group – call it $\{cl\}$ which stands for candidate list
- A DAAT algorithm is used to evaluate the new query

$$Q_{DAAT} = Q_{t > T} \cup \{cl\}$$

- Seeding the DAAT algorithm with an initial good lower bound θ enables more skipping

Diagram of Hybrid Method



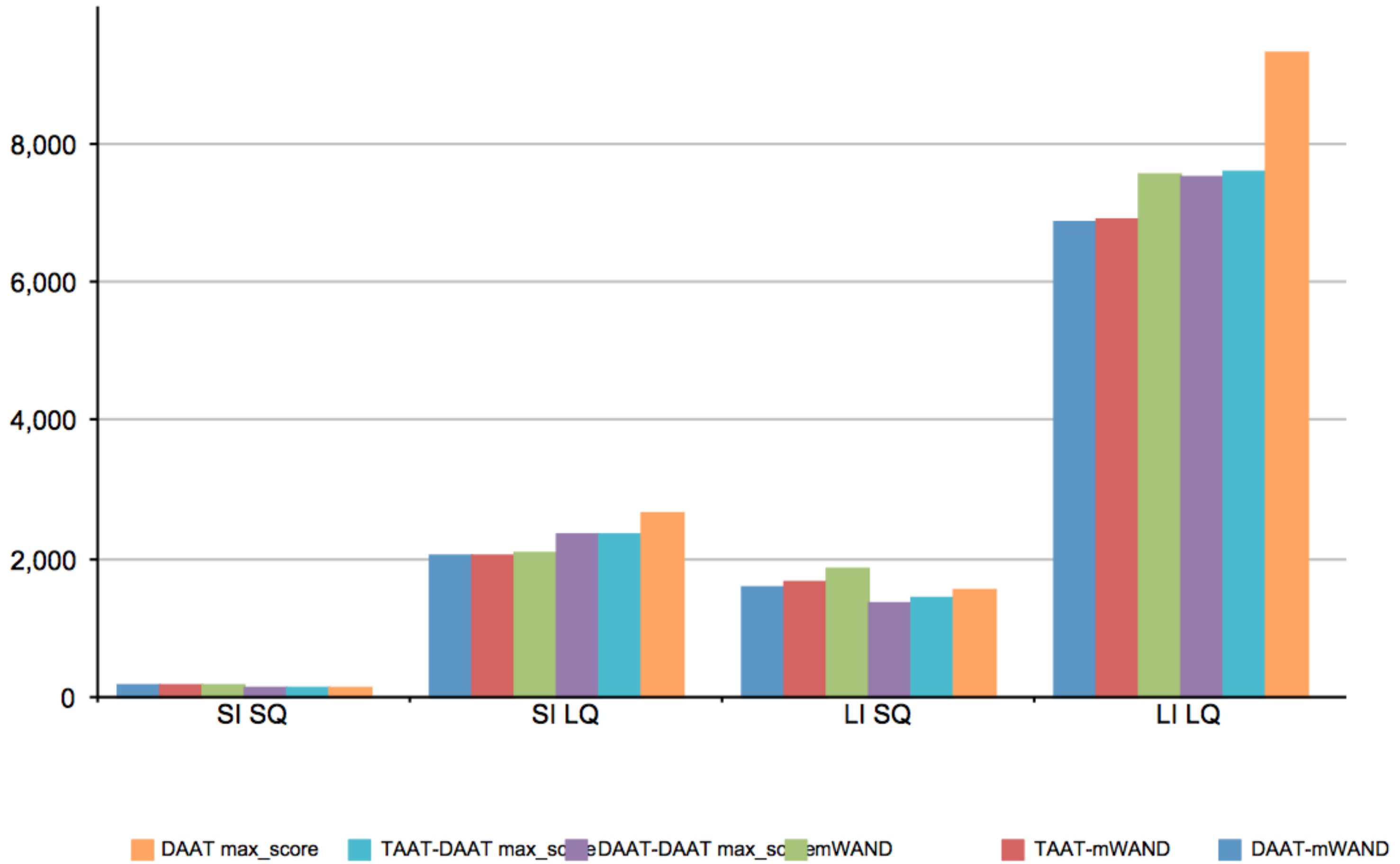
Short posting lists evaluated to calc **θ** and
used to create one virtual or merged posting list V

Then use V (along with the long posting lists) with a DAAT algorithm using **θ** LB

Optimizing DAAT – Hybrid Algorithms

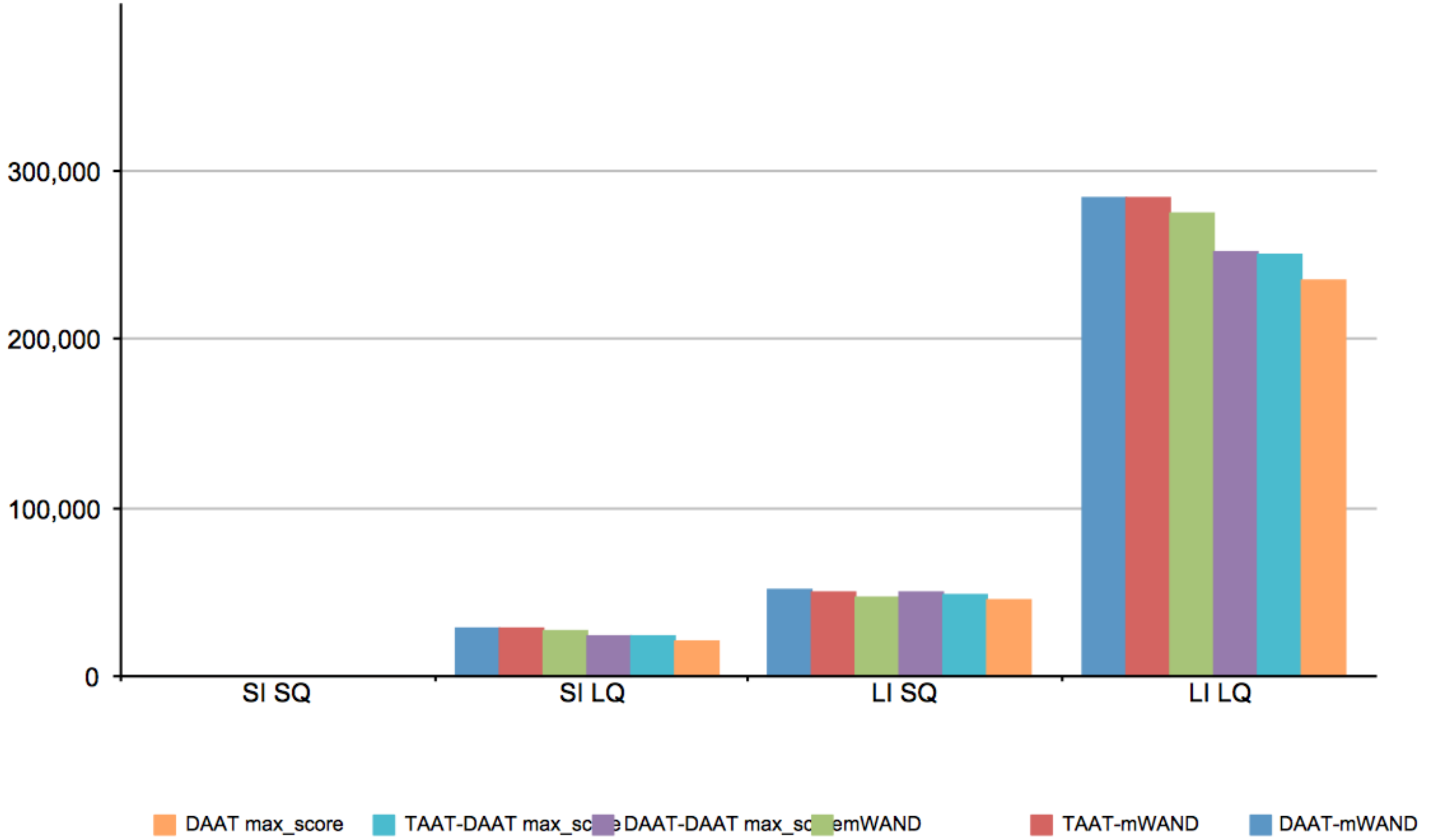
- DAAT-mWAND – uses naïve DAAT for $Q_{t \leq T}$ and mWAND for Q_{DAAT}
- TAAT-mWAND – uses naïve TAAT for $Q_{t \leq T}$ and mWAND for Q_{DAAT}
- DAAT-DAAT max_score – uses naïve DAAT for $Q_{t \leq T}$ and DAAT max_score for Q_{DAAT}
- TAAT-DAAT max_score – uses naïve TAAT for $Q_{t \leq T}$ and DAAT max_score for Q_{DAAT}

Hybrid Algorithms - Latency



- For LI LQ test, DAAT-mWAND 10.7% faster than mWAND and 35.8% faster than DAAT max_score

Hybrid Algorithms – Skipped Postings



Conclusion

- Evaluated traditional DAAT and TAAT algorithms in an in-memory index production setting
- Proposed adaptations to the existing algorithms that are better suited for index accesses over memory
- Achieved 60% latency improvements over traditional algorithms
- Proposed new hybrid technique to speed up DAAT algorithms by segmenting query terms
  - Achieves 20% incremental latency gains