V.3 Query Processing

1. **Term-at-a-Time**
2. **Document-at-a-Time**
3. **WAND**
4. **Quit & Continue**
5. **Buckley's Algorithm**
6. **Fagin's Threshold Algorithms**
7. **Query Processing with Importance Scores**
8. **Query Processing with Champion Lists**

Based on **MRS Chapter 7** and **RBY Chapter 9**

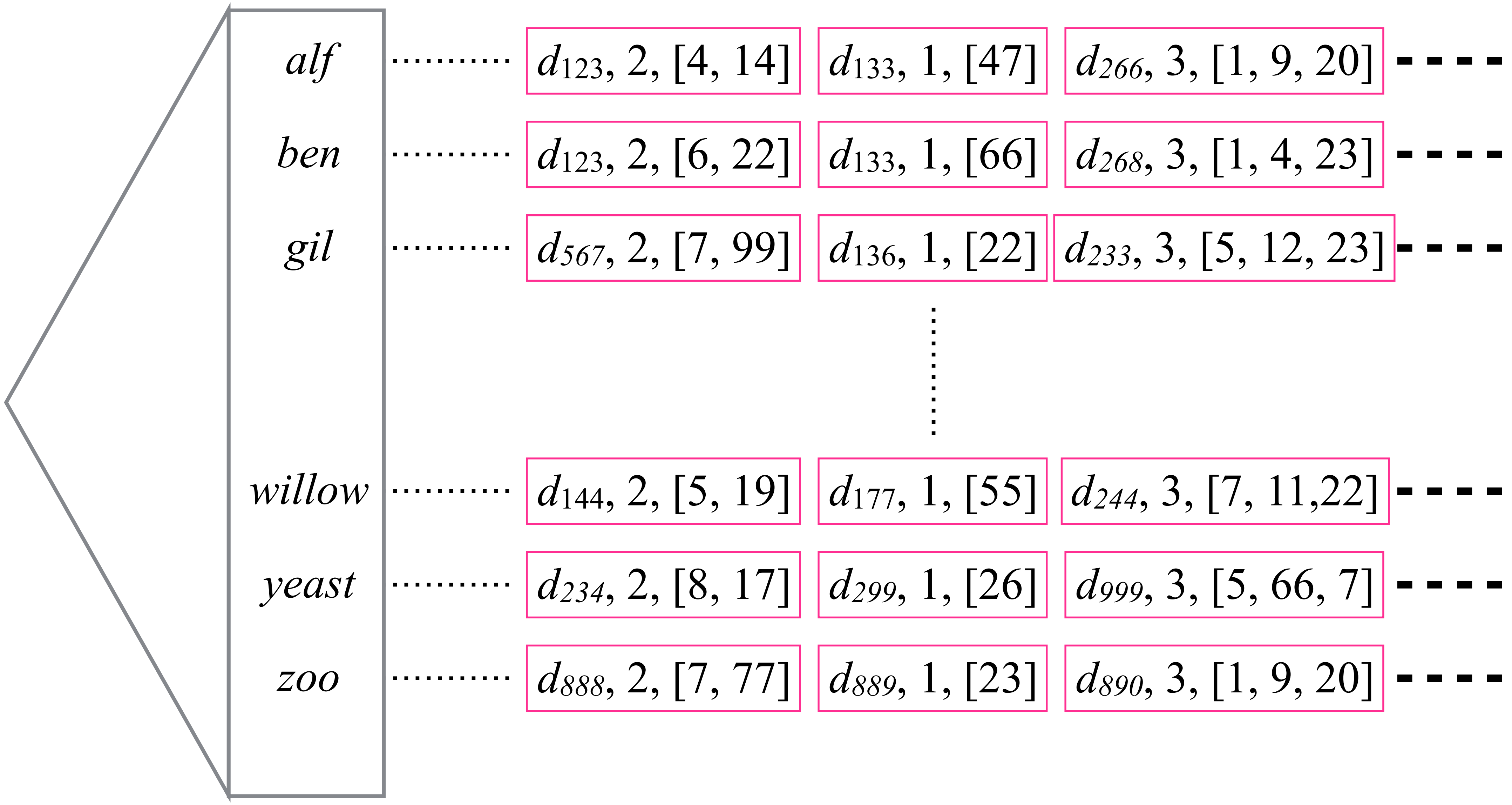
Query Types

- **Conjunctive**
  (i.e., all query terms are required)
- **Disjunctive**
  (i.e., subset of query terms sufficient)
- **Phrase** or **proximity**
  (i.e., query terms must occur in right order or close enough)
- **Mixed-mode** with **negation**
  (e.g., *"harry potter" review +movie -book*)
- Combined with **ranking of result documents** according to

$$score(q, d) = \sum_{t \in q} score(t, d)$$

with *score*(*t*, *d*) depending on retrieval model (e.g., $tf.idf_{t,d}$)

Inverted Index



- **Document-ordered** or **score-ordered** posting lists
- Posting lists with **skip pointers** allow for **faster traversal**

Overview of Query Processing Methods

- **Holistic** query processing methods determine whole query result
  - Term-at-a-Time
  - Document-at-a-Time
- **Top-*k*** query processing methods determine top-*k* query result
  - WAND
  - Quit & Continue
  - Fagin's Threshold Algorithms
- Opportunities for optimization over **naïve merge & sort baseline**
  - **skipping** in document-ordered posting lists
  - **early termination** of query processing for score-ordered posting lists

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms ⟨ $t_1, \ldots, t_{|q|}$ ⟩ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	0.0
d_4	:	0.0
d_7	:	0.0
d_8	:	0.0
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms ⟨ $t_1, \ldots, t_{|q|}$ ⟩ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	0.0
d_4	:	0.0
d_7	:	0.0
d_8	:	0.0
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	0.0
d_7	:	0.0
d_8	:	0.0
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d) \text{ after the first } j \text{ posting lists have been read}$$

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	0.0
d_7	:	0.0
d_8	:	0.0
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	2.0
d_7	:	0.0
d_8	:	0.0
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	2.0
d_7	:	0.0
d_8	:	0.0
d_9	:	0.0

- required **memory** depends on the **number of accumulators** maintained
- **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms ⟨ $t_1, \ldots, t_{|q|}$ ⟩ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	2.0
d_7	:	0.2
d_8	:	0.0
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	2.0
d_7	:	0.2
d_8	:	0.0
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first *j* posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	2.0
d_7	:	0.2
d_8	:	0.1
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	2.0
d_7	:	0.2
d_8	:	0.1
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	3.0
d_7	:	0.2
d_8	:	0.1
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-k results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	3.0
d_7	:	0.2
d_8	:	0.1
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	3.0
d_7	:	2.2
d_8	:	0.1
d_9	:	0.0

- required **memory** depends on the **number of accumulators** maintained
- **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \le j} score(t_i, d) \text{ after the first } j \text{ posting lists have been read}$$

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	3.0
d_7	:	2.2
d_8	:	0.1
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first *j* posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	3.0
d_7	:	2.2
d_8	:	0.3
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first *j* posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	3.0
d_7	:	2.2
d_8	:	0.3
d_9	:	0.0

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	3.0
d_7	:	2.2
d_8	:	0.3
d_9	:	0.1

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first *j* posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	3.0
d_7	:	2.2
d_8	:	0.3
d_9	:	0.1

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	6.0
d_7	:	2.2
d_8	:	0.3
d_9	:	0.1

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	6.0
d_7	:	2.2
d_8	:	0.3
d_9	:	0.1

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Accumulators

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2
d_8	:	0.3
d_9	:	0.1

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

1. Term-at-a-Time Query Processing

- **Term-at-a-Time** (TAAT) query processing
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively**
  - maintains an **accumulator** for each result document with value

$$acc(d) = \sum_{i \leq j} score(t_i, d)$$ after the first j posting lists have been read

a ………… d_1, 1.0 | d_4, 2.0 | d_7, 0.2 | d_8, 0.1

b ………… d_4, 1.0 | d_7, 2.0 | d_8, 0.2 | d_9, 0.1

c ………… d_4, 3.0 | d_7, 1.0

Accumulators

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2
d_8	:	0.3
d_9	:	0.1

  - required **memory** depends on the **number of accumulators** maintained
  - **top-*k* results** can be determined by **sorting accumulators** at the end

Term-at-a-Time Query Processing

- Optimizations for **conjunctive queries**
  - process query terms in **ascending order of their document frequency** to keep the number of accumulators and thus required memory low
  - for document-ordered posting lists, **keep accumulators sorted** to make use of **skip pointers** when read posting lists

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms ⟨ $t_1, \ldots, t_{|q|}$ ⟩ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2
d_8	:	0.3

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2
d_8	:	0.3

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2
d_8	:	0.3

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms $\langle\, t_1, \ldots, t_{|q|} \,\rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2
d_8	:	0.3
d_9	:	0.1

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

2. Document-at-a-Time Query Processing

- **Document-at-a-Time** (DAAT) query processing
  - assumes **document-ordered posting lists**
  - reads posting lists for query terms ⟨ $t_1, \ldots, t_{|q|}$ ⟩ **concurrently**
  - computes score when **same document** is seen in one or more posting lists

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_1	:	1.0
d_4	:	6.0
d_7	:	3.2
d_8	:	0.3
d_9	:	0.1

  - always advances posting list with **lowest current document identifier**
  - required main memory depends on the **number of results** to be reported
  - **top-*k* results** can be determined by keeping results in **priority queue**

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

  where *cdid*(*i*) is the current document identifier in the *i*-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

  where $cdid(i)$ is the current document identifier in the i-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

where $cdid(i)$ is the current document identifier in the i-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_4 : 6.0

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

where *cdid*(*i*) is the current document identifier in the *i*-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_4	:	6.0

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

where *cdid*(*i*) is the current document identifier in the *i*-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_4 :	6.0

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

where $cdid(i)$ is the current document identifier in the i-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_4 :	6.0

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

where *cdid*(*i*) is the current document identifier in the *i*-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_4	:	6.0
d_7	:	3.2

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

where *cdid*(*i*) is the current document identifier in the *i*-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_4	:	6.0
d_7	:	3.2

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

where *cdid*(*i*) is the current document identifier in the *i*-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_4	:	6.0
d_7	:	3.2

Document-at-a-Time Query Processing

- Optimization for **conjunctive queries** using **skip pointers**
  - when advancing posting list with lowest current document identifier, advance to first posting having document identifier **larger or equal to**

$$\max_i cdid(i)$$

where $cdid(i)$ is the current document identifier in the i-th posting list

a	d_1, 1.0	d_4, 2.0	d_7, 0.2	d_8, 0.1
b	d_4, 1.0	d_7, 2.0	d_8, 0.2	d_9, 0.1
c	d_4, 3.0	d_7, 1.0		

d_4	:	6.0
d_7	:	3.2

3. WAND

- **Weak AND** (WAND) query processing
  - assumes **document-ordered posting lists** with known maximum score *maxscore*(i) of any posting in the i-th posting list
  - reads posting lists for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **concurrently**
  - computes score when **same document** is seen in one or more posting lists
  - always advances posting list with **lowest current document identifier** up to **pivot document identifier** computed from **current top-*k* result**

- Computation of **pivot document identifier**
  - let min_k denote the **lowest score** in current top-k results
  - sort posting lists in **ascending order** of *cdid*(i)
  - pivot is *cdid*(j) of **minimal *j*** such that $min_k < \sum_{i \leq j} maxscore(i)$

WAND

- Computation of **pivot document identifier**
  - let min_k denote the **lowest score** in current top-k results
  - sort posting lists in **ascending order** of $cdid(i)$
  - pivot is $cdid(j)$ of **minimal *j*** such that $min_k < \sum_{i \leq j} maxscore(i)$

a	…………	d_2, 0.5	d_7, 0.1	d_8, 0.2	d_9, 0.6	……… d_{99}, 1.0	………
b	…………	d_2, 0.5	d_9, 0.3	d_{11}, 0.2	d_{13}, 0.1	……… d_{33}, 1.0	………
c	…………	d_2, 0.5	d_3, 0.4	d_4, 0.2	d_5, 0.1	……… d_{57}, 1.0	………

Top-1

d_2	:	1.5

Pivot Computation

d_3, 0.4	1.0	$maxscore(i) = 1.0$
d_7, 0.1	**2.0**	**d_7** is pivot
d_9, 0.3	3.0	

WAND

- Intuition: No document with an identifier smaller than the pivot can have a **score large enough** to make it into the top-*k* result

- Observation: As the value of ***min*$_k$ can only increase** over time, WAND **skips more and more postings** as time progresses

- WAND can be made an **approximate top-*k* query processing method** by computing the pivot such that

$$F \times min_k < \sum_{i \leq j} maxscore(i)$$

  with **tunable parameter** *F* controlling fidelity of results

- Full details: [Broder et al. ’03]

4. Quit & Continue

- **Quit & Continue** query processing
  - reads **score-ordered posting lists** for query terms $\langle t_1, \ldots, t_{|q|} \rangle$ **successively** in descending order of $idf(t_i)$
- **Quit heuristics**
  - **ignore** posting lists for terms t_i with $idf(t_i)$ below threshold
  - **stop scanning** posting list for t_i if $tf(t_i, d_j) * idf(t_i)$ drops below threshold
  - **stop scanning** posting list when the number of accumulators is too high
- **Continue heuristics**
  - upon reaching accumulator limit, continue reading remaining posting lists, update existing accumulators but do not create new accumulators
- Full details: [Moffat and Zobel '96]

5. Buckley's Algorithm

- **Buckley's** query processing method
  - reads **score-ordered posting lists concurrently** in round-robin manner
  - maintains **partial scores** of documents and keeps track of *k*-th best score
  - computes **upper bound** for any unseen document based on current scores

$$ub = \sum_i cscore(i)$$

    with $cscore(i)$ as the **current score** in the i-th posting list

  - **stops** if upper bound ub is less than k-th best partial score

a …………	d_2, 0.5	d_1, 0.4	d_5, 0.3	d_9, 0.2	………
b …………	d_2, 0.5	d_3, 0.5	d_{61}, 0.4	d_{13}, 0.1	………
c …………	d_3, 0.4	d_5, 0.3	d_7, 0.2	d_4, 0.1	………

Top-1

d_2	:	1.0

$$ub = 0.9$$

Buckley's Algorithm

- Note: This is a simplified version of Buckley's algorithm. The original algorithm maintains an upper bound for the $(k + 1)$-th best document. If implemented correctly, this gives us the **first exact top-*k* query processing method** described in the literature, which is only based on sequential accesses.

- Full details: [Buckley and Lewitt '85]

6. Fagin's Threshold Algorithms

- **Threshold Algorithm** (TA)
  - original version, often used as synonym for entire family of algorithms
  - requires eager random access to candidate objects
  - worst-case memory consumption: $O(k)$
- **No-Random-Accesses** (NRA)
  - no random access required, may have to scan large parts of the lists
  - worst-case memory consumption: $O(m*n + k)$
- **Combined Algorithm** (CA)
  - cost-model for scheduling random accesses to candidate objects
  - algorithmic skeleton very similar to NRA, but typically terminates faster
  - worst-case memory consumption: $O(m*n + k)$

Fagin's Threshold Algorithms

- Assume **score-ordered posting lists** and **additional index** for score look-ups by document identifier
- Scan posting lists using **inexpensive sequential accesses** (SA) in round-robin manner
- Perform **expensive random accesses** (RA) to look up scores for a specific document when beneficial
- Support **monotone score aggregation function**

$$aggr : \mathbb{R}^m \rightarrow \mathbb{R} \; : \; \forall x_i \geq x'_i \Rightarrow aggr(x_1, \ldots, x_m) \geq aggr(x'_1, \ldots, x'_m)$$

- Compute **aggregate scores** incrementally in **candidate queue**
- Compute **score bounds** for candidate results and stop when **threshold test** guarantees correct top-*k* result

Threshold Algorithm (TA)

- **Sequential accesses** (SA) mixed with eager **random accesses** (RA)
- Worst-case memory consumption $O(k)$

<u>Threshold Algorithm</u> (TA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$

if $d \notin$ top-k **then** // compute $score(d)$
  look up $score(t_j, d)$ for all $j \neq i$
  $score(d) = aggr\{\ score(t_j, d) \mid j = 1 \ldots |q|\ \}$

if $score(d) >$ min-k **then** // update top-k
  add d to top-k and remove min-score d'
  $\min_k = min\{\ score(d') \mid d' \in$ top-$k\ \}$

$ub = \text{aggr}\{high(i) \mid i = 1 \ldots |q|\}$ // update upper bound
if $ub \leq min_k$ **then**
  exit

a	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2
b	d_{64}, 0.9	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1
c	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1

Top-2

SA RA

Threshold Algorithm (TA)

- **Sequential accesses** (SA) mixed with eager **random accesses** (RA)
- Worst-case memory consumption $O(k)$

<u>Threshold Algorithm</u> (TA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$

if $d \notin$ top-k **then** // compute $score(d)$
  look up $score(t_j, d)$ for all $j \neq i$
  $score(d) = aggr\{\, score(t_j, d) \mid j = 1 \ldots |q| \,\}$

if $score(d) >$ min-k **then** // update top-k
  add d to top-k and remove min-score d'
  $\min_k = min\{\, score(d') \mid d' \in$ top-$k\,\}$

$ub = \text{aggr}\{high(i) \mid i = 1 \ldots |q|\}$ // update upper bound
if $ub \leq min_k$ **then**
  exit

a	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2	…
b	d_{64}, 0.9	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1	…
c	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1	…

$ub = 2.5$

Top-2

d_{10}	:	2.1
d_{78}	:	1.5

SA RA

Threshold Algorithm (TA)

- **Sequential accesses** (SA) mixed with eager **random accesses** (RA)
- Worst-case memory consumption $O(k)$

<u>Threshold Algorithm</u> (TA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$

if $d \notin$ top-k **then** // compute $score(d)$
  look up $score(t_j, d)$ for all $j \neq i$
  $score(d) = aggr\{\ score(t_j, d) \mid j = 1 \ldots |q|\ \}$

if $score(d) >$ min-k **then** // update top-k
  add d to top-k and remove min-score d'
  $\min_k = min\{\ score(d') \mid d' \in$ top-k $\}$

$ub = \text{aggr}\{high(i) \mid i = 1 \ldots |q|\}$ // update upper bound
if $ub \leq min_k$ **then**
  exit

a	……… d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2	………
b	……… d_{64}, 0.9	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1	………
c	……… d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1	………

$ub = 1.9$

Top-2

d_{10}	:	2.1
d_{78}	:	1.5

SA RA

Threshold Algorithm (TA)

- **Sequential accesses** (SA) mixed with eager **random accesses** (RA)
- Worst-case memory consumption $O(k)$

<u>Threshold Algorithm</u> (TA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$

if $d \notin$ top-k **then** // compute $score(d)$
  look up $score(t_j, d)$ for all $j \neq i$
  $score(d) = aggr\{\ score(t_j, d) \mid j = 1 \ldots |q|\ \}$

if $score(d) >$ min-k **then** // update top-k
  add d to top-k and remove min-score d'
  $\min_k = min\{\ score(d') \mid d' \in$ top-k $\}$

$ub = \text{aggr}\{high(i) \mid i = 1 \ldots |q|\}$ // update upper bound
if $ub \leq min_k$ **then**
  exit

a	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2
b	d_{64}, 0.9	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1
c	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1

$ub = 1.7$

Top-2

d_{10}	:	2.1
d_{78}	:	1.5

SA RA

Threshold Algorithm (TA)

- **Sequential accesses** (SA) mixed with eager **random accesses** (RA)
- Worst-case memory consumption $O(k)$

<u>Threshold Algorithm</u> (TA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$

if $d \notin$ top-k **then** // compute $score(d)$
    look up $score(t_j, d)$ for all $j \neq i$
    $score(d) = aggr\{\ score(t_j, d) \mid j = 1 \ldots |q|\ \}$

if $score(d) >$ min-k **then** // update top-k
    add d to top-k and remove min-score d'
    $\min_k = min\{\ score(d') \mid d' \in \text{top-}k\ \}$

$ub = \text{aggr}\{high(i) \mid i = 1 \ldots |q|\}$ // update upper bound
if $ub \leq min_k$ **then**
    exit

a	………	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_1, 0.7	d_{88}, 0.2	………
b	………	d_{64}, 0.9	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1	………
c	………	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1	………

$ub = 1.1$

Top-2

d_{10}	:	2.1
d_{78}	:	1.5

SA RA

Threshold Algorithm (TA)

- **Sequential accesses** (SA) mixed with eager **random accesses** (RA)
- Worst-case memory consumption $O(k)$

<u>Threshold Algorithm</u> (TA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$

if $d \notin$ top-k **then** // compute $score(d)$
  look up $score(t_j, d)$ for all $j \neq i$
  $score(d) = aggr\{\ score(t_j, d) \mid j = 1 \ldots |q|\ \}$

if $score(d) >$ min-k **then** // update top-k
  add d to top-k and remove min-score d'
  $\min_k = min\{\ score(d') \mid d' \in$ top-k $\}$

$ub = \text{aggr}\{high(i) \mid i = 1 \ldots |q|\}$ // update upper bound
if $ub \leq min_k$ **then**
  exit

a	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2
b	d_{64}, 0.9	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1
c	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1

$ub = 1.1$

Top-2

d_{10}	:	2.1
d_{78}	:	1.5

SA RA

No-Random-Accesses Algorithm (NRA)

- **Sequential accesses** (SA) only
- Worst-case memory consumption $O(m*n + k)$

No-Random-Accesses Algorithm (NRA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$
$eval(d) = eval(d) \cup \{i\}$ // where have we seen d?

$worst(d) = aggr\{\ score(t_j, d) \mid j \in eval(d)\ \}$
$best(d) = aggr\{\ worst(d), aggr\{\ high(j) \mid j \notin eval(d)\ \}\ \}$

if $worst(d) > min_k$ **then** // good enough for top-k?
    add d top top-k
    $min_k = min\{\ worst(d') \mid d' \in$ top-$k\ \}$
else if $best(d) > min_k$ **then** // good enough for cand?
    $cand = cand \cup \{\ d\ \}$
$ub = max\{\ best(d') \mid d' \in cand\ \}$
if $ub \leq min_k$ **then**
    exit

a	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2
b	d_{64}, 0.8	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1
c	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1

SA RA

No-Random-Accesses Algorithm (NRA)

- **Sequential accesses** (SA) only
- Worst-case memory consumption $O(m*n + k)$

No-Random-Accesses Algorithm (NRA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$
$eval(d) = eval(d) \cup \{i\}$ // where have we seen d?

$worst(d) = aggr\{\ score(t_j, d) \mid j \in eval(d)\ \}$
$best(d) = aggr\{\ worst(d), aggr\{\ high(j) \mid j \notin eval(d)\ \}\ \}$

if $worst(d) > min_k$ **then** // good enough for top-k?
  add d top top-k
  $min_k = min\{\ worst(d') \mid d' \in$ top-$k\ \}$
else if $best(d) > min_k$ **then** // good enough for cand?
  $cand = cand \cup \{\ d\ \}$
$ub = max\{\ best(d') \mid d' \in cand\ \}$
if $ub \leq min_k$ **then**
  exit

a	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2
b	d_{64}, 0.8	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1
c	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1

$ub = 2.4$

	worst	best	
d_{78}	0.9	2.4	Top-*1*
d_{64}	0.8	2.4	
d_{10}	0.7	2.4	

SA RA

No-Random-Accesses Algorithm (NRA)

- **Sequential accesses** (SA) only
- Worst-case memory consumption $O(m*n + k)$

<u>No-Random-Accesses Algorithm</u> (NRA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$
$eval(d) = eval(d) \cup \{i\}$ // where have we seen d?

$worst(d) = aggr\{\ score(t_j, d) \mid j \in eval(d)\ \}$
$best(d) = aggr\{\ worst(d), aggr\{\ high(j) \mid j \notin eval(d)\ \}\ \}$

if $worst(d) > min_k$ **then** // good enough for top-k?
    add d top top-k
    $min_k = min\{\ worst(d') \mid d' \in$ top-k $\}$
else if $best(d) > min_k$ **then** // good enough for cand?
    $cand = cand \cup \{\ d\ \}$
$ub = max\{\ best(d') \mid d' \in cand\ \}$
if $ub \leq min_k$ **then**
    exit

a	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2
b	d_{64}, 0.8	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1
c	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1

$ub = 2.1$

	worst	best	
d_{78}	1.4	2.0	Top-*1*
d_{23}	1.4	1.9	
d_{64}	0.8	2.1	
d_{10}	0.7	2.1	

SA RA

No-Random-Accesses Algorithm (NRA)

- **Sequential accesses** (SA) only
- Worst-case memory consumption $O(m*n + k)$

No-Random-Accesses Algorithm (NRA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$
$eval(d) = eval(d) \cup \{i\}$ // where have we seen d?

$worst(d) = aggr\{\ score(t_j, d) \mid j \in eval(d)\ \}$
$best(d) = aggr\{\ worst(d),\ aggr\{\ high(j) \mid j \notin eval(d)\ \}\ \}$

if $worst(d) > min_k$ **then** // good enough for top-k?
  add d top top-k
  $min_k = min\{\ worst(d') \mid d' \in \text{top-}k\ \}$
else if $best(d) > min_k$ **then** // good enough for cand?
  $cand = cand \cup \{\ d\ \}$
$ub = max\{\ best(d') \mid d' \in cand\ \}$
if $ub \leq min_k$ **then**
  exit

a	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2
b	d_{64}, 0.8	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1
c	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1

$ub = 2.0$

	worst	best	
d_{10}	2.1	2.1	Top-*1*
d_{78}	1.4	2.0	
d_{23}	1.4	1.7	
d_{64}	1.1	1.9	

SA RA

No-Random-Accesses Algorithm (NRA)

- **Sequential accesses** (SA) only
- Worst-case memory consumption $O(m*n + k)$

<u>No-Random-Accesses Algorithm</u> (NRA):
scan index lists (e.g., round-robin)
consider $d = cdid(i)$ in posting list for t_i
$high(i) = cscore(i)$
$eval(d) = eval(d) \cup \{i\}$ // where have we seen d?

$worst(d) = aggr\{\ score(t_j, d) \mid j \in eval(d)\ \}$
$best(d) = aggr\{\ worst(d), aggr\{\ high(j) \mid j \notin eval(d)\ \}\ \}$

if $worst(d) > min_k$ **then** // good enough for top-k?
  add d top top-k
  $min_k = min\{\ worst(d') \mid d' \in$ top-k $\}$
else if $best(d) > min_k$ **then** // good enough for cand?
  $cand = cand \cup \{\ d\ \}$
$ub = max\{\ best(d') \mid d' \in cand\ \}$
if $ub \leq min_k$ **then**
  exit

a	d_{78}, 0.9	d_{23}, 0.8	d_{10}, 0.8	d_{1}, 0.7	d_{88}, 0.2
b	d_{64}, 0.8	d_{23}, 0.6	d_{10}, 0.6	d_{12}, 0.2	d_{78}, 0.1
c	d_{10}, 0.7	d_{78}, 0.5	d_{64}, 0.3	d_{99}, 0.2	d_{34}, 0.1

$ub = 2.0$

STOP!

	worst	best	
d_{10}	2.1	2.1	Top-*1*
d_{78}	1.4	2.0	
d_{23}	1.4	1.7	
d_{64}	1.1	1.9	

SA RA

Combined Algorithm (CA)

- **Balanced SA/RA Scheduling**:
  - define **cost ratio** $r = C_{SA}/C_{RA}$ (e.g., based on statistics for execution environment, typical values $C_{RA}/C_{SA} \sim 100 - 10{,}000$ for hard disks)
  - run **NRA** (using SA only) but **perform one RA every r rounds** (i.e., $m*r$ SAs) to look up the unknown scores of the **best candidate** that is not in the current top-k
- **Cost competitiveness** w.r.t. "optimal schedule" (scan until $aggr\{\ high(i)\ \} \leq min\{\ best(d) \mid d \in \text{final top-}k\ \}$, then perform RAs for all d' with $best(d') > min_k$): **4*m + k**

TA / NRA / CA Instance Optimality

- Definition: For class of algorithms **A** and class of datasets **D**, algorithm A ∈ **A** is **instance optimal** over **A** and **D** if

$$\forall A' \in \mathbf{A}\ \forall D \in \mathbf{D}\ :\ cost(A, D) \leq c \cdot cost(A', D) + c'$$

$$\text{(i.e., } cost(A, D) \in \mathcal{O}(cost(A', D)))$$

- **TA is instance optimal** over all top-*k* algorithms based on random and sequential accesses to *m* lists (no “wild guesses”)

- **NRA is instance optimal** over all top-*k* algorithms based on only sequential accesses

- **CA is instance optimal** over all top-*k* algorithms based on **random and sequential accesses** and given **cost ratio** C_{RA}/C_{SA}

- Full details: [Fagin et al. ’03]

Implementation Issues for Threshold Algorithms

- Limitation of **asymptotic complexity**
  - m (# lists), n (# documents), k (# results) are **important parameters**
- Priority queues
  - straightforward use of **heap** (even Fibonacci) has high overhead
  - better: **periodic rebuilding** of queue with partial sort $O(n \log k)$
- Memory management
  - **peak memory usage** as important for performance as scan depth
  - aim for **early candidate pruning** even if scan depth stays the same

7. Query Processing with Importance Scores

- Focus on score combining **textual relevance** (*rel*) (e.g., TF*IDF) and **global importance** (*imp*) (e.g., PageRank)

$$score(q, d) = imp(d) + rel(q, d)$$

  with **normalization** $imp(d) \leq a$ and $rel(q, d) \leq b$ and $a + b \leq 1$

- Keep posting lists in descending order of **global importance**

$high(i) = imp(cdid(i)) + b$ // upper bound for document from i-th list $high = max\{ high(i) \mid i = 1 \ldots \lvert q \rvert \} + b$ // global upper bound Stop scanning i-th posting list when high(i) < min_k (i.e., minimal score in top-k) Terminate when $high < min_k$

  effective when combined score is dominated by $imp(d)$

- **First-*k*' heuristic**: Scan all posting lists until $k' \geq k$ documents have been seen in all lists, so that their combined score is known

- Full details: [Long and Suel '03]

8. Query Processing with Champion Lists

- Idea: In addition to full posting lists L_i sorted by *imp*(*d*), keep short **"champion lists"** sorted (aka. "fancy lists") F_i that contain docs *d* with the highest values of score(t_i, *d*) and sort these lists by *imp*(*d*)

- **Champions First-*k*' heuristic**:

```
Compute total score for all docs in ∩ Fi (i = 1 … |q|) and keep top-k results
cand = ∪ Fi - ∩ Fi
for each d ∈ cand do
    compute partial score of d

scan full posting lists Li (i = 1 … |q|)
    if cdid(i) ∈ cand then
        add score(ti, cdid(i)) to partial score of cdid(i)
    else
        add cdid(i) to cand and set its partial score to score(ti, cdid(i))
    terminate the scan when we have k' documents with complete scores
```

- Full details: [Brin and Page '98]

Summary of V.3

- **Query Type**
  determines usefulness of optimizations (e.g., skip pointers)
- **Term-at-a-Time** and **Document-at-a-Time**
  for holistic query processing
- **WAND**
  for top-*k* query processing on document-ordered posting lists
- **Buckley's Algorithm**
  for top-*k* query processing on scored-ordered posting lists
- **Fagin's Threshold Algorithms**
  top-*k* query processing with, without, or with some RAs

Additional Literature for V.3

- **S. Brin and L. Page**: *The anatomy of a large-scale hypertextual Web search engine*, Computer Networks 30:107-117, 1998
- **A. Broder, D. Carmel, M. Herscovici, A. Soffer, J. Zien**: *Efficient query evaluation using a two-level retrieval process*, CIKM 2003
- **C. Buckley and A. Lewit**: *Optimization of Inverted Vector Searches*, SIGIR 1985
- **R. Fagin, A. Lotem, and M. Naor**: *Optimal Aggregation Algorithms for Middleware*, Journal of Computer and System Sciences 2003
- **X. Long and T. Suel**: *Optimized Query Execution in Large Search Engines with Global Page Ordering*, VLDB 2003
- **J. Zobel and A. Moffat**: *Self-Indexing Inverted Files for Fast Text Retrieval*, ACM TOIS 14(4):349-379, 1996