Informatics 225
Computer Science 221

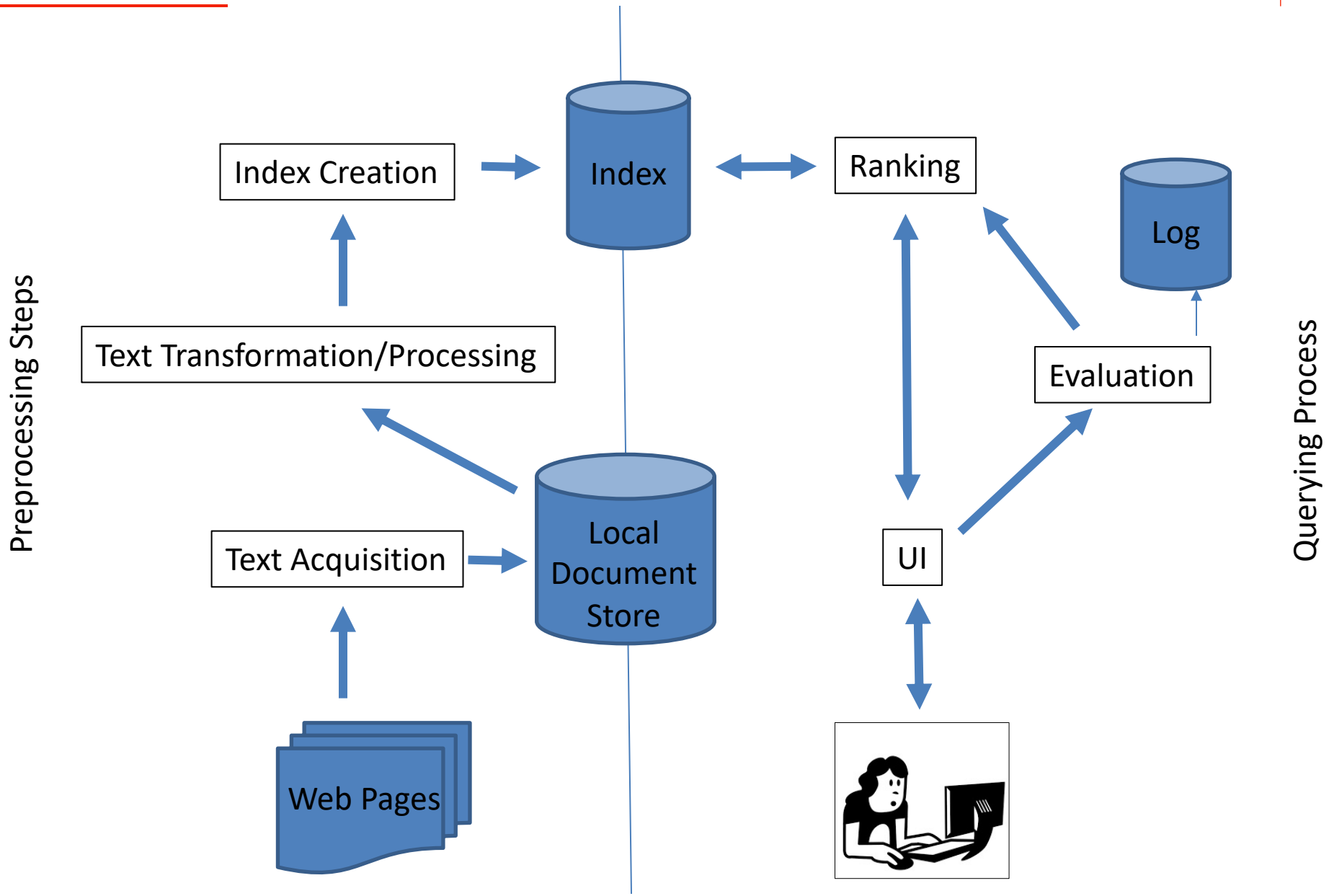
Information Retrieval

Lecture 19

These course materials borrow, with permission, from those of Prof. Cristina Videira Lopes, Addison Wesley 2008, Chris Manning, Pandu Nayak, Hinrich Schütze, Heike Adel, Sascha Rothe, Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie. Powerpoint theme by Prof. André van der Hoek.

Optimizing Query Evaluation

Architecture



Distributed Query Evaluation

- Will speed up significantly the query processing speed

Distributed Query Evaluation

- Will speed up significantly the query processing speed
- Basic process
  - All queries sent to a *director machine*

Distributed Query Evaluation

- Will speed up significantly the query processing speed

- Basic process
  - All queries sent to a *director machine*
  - Director then sends messages to many *index servers*

Distributed Query Evaluation

- Will speed up significantly the query processing speed

- Basic process
  - All queries sent to a *director machine*
  - Director then sends messages to many *index servers*
  - Each index server does some portion of the query processing

Distributed Query Evaluation

- Will speed up significantly the query processing speed

- Basic process
  - All queries sent to a *director machine*
  - Director then sends messages to many *index servers*
  - Each index server does some portion of the query processing
  - Director organizes the results and returns them to the user

Distributed Query Evaluation

- Will speed up significantly the query processing speed
- Basic process
  - All queries sent to a *director machine*
  - Director then sends messages to many *index servers*
  - Each index server does some portion of the query processing
  - Director organizes the results and returns them to the user
- Two main approaches on how to distribute the query
  - Document distribution
  - Term distribution

Distributed Query Evaluation

- Document distribution
  - each index server acts as a search engine for a small fraction of the total collection

Distributed Query Evaluation

- Document distribution
  - each index server acts as a search engine for a small fraction of the total collection
  - director sends a copy of the query to each of the index servers, each of which returns the top-*k* results

Distributed Query Evaluation

- Document distribution
  - each index server acts as a search engine for a small fraction of the total collection
  - director sends a copy of the query to each of the index servers, each of which returns the top-*k* results
  - results are merged into a single ranked list by the director

Distributed Query Evaluation

- Document distribution
  - each index server acts as a search engine for a small fraction of the total collection
  - director sends a copy of the query to each of the index servers, each of which returns the top-*k* results
  - results are merged into a single ranked list by the director
- Collection statistics should be shared for effective ranking

Distributed Query Evaluation

- Term distribution
  - Single index is built for the whole cluster of machines

Distributed Query Evaluation

- Term distribution
  - Single index is built for the whole cluster of machines
  - Each inverted list in that index is then assigned to one index server
    - in most cases the data to process a query is not stored on a single machine

Distributed Query Evaluation

- Term distribution
  - Single index is built for the whole cluster of machines
  - Each inverted list in that index is then assigned to one index server
    - in most cases the data to process a query is not stored on a single machine
  - One of the index servers is chosen to process the query
    - usually the one holding the longest inverted list

Distributed Query Evaluation

- Term distribution
  - Single index is built for the whole cluster of machines
  - Each inverted list in that index is then assigned to one index server
    - in most cases the data to process a query is not stored on a single machine
  - One of the index servers is chosen to process the query
    - usually the one holding the longest inverted list
  - Other index servers send information to that server

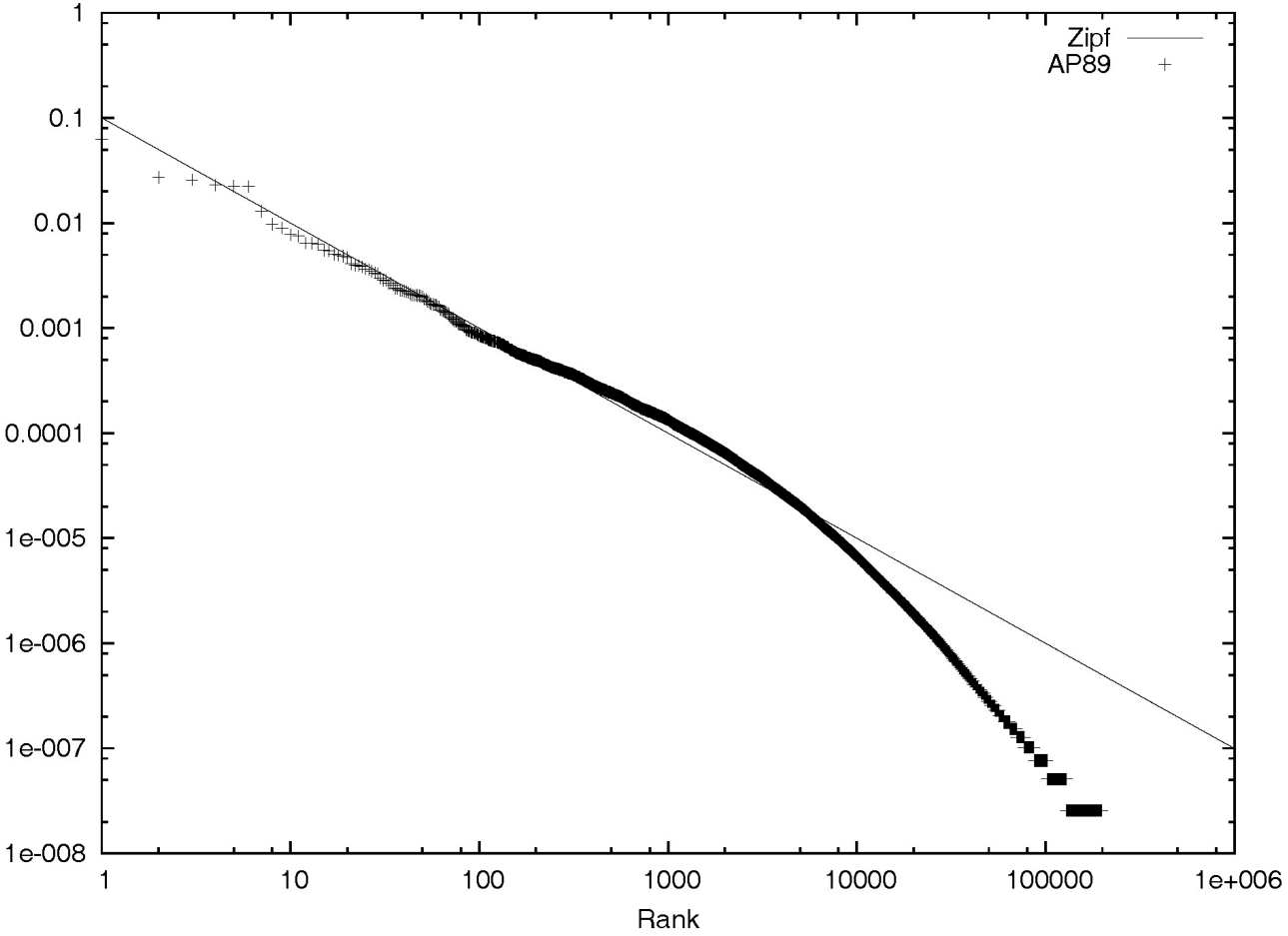
Distributed Query Evaluation

- Term distribution
  - Single index is built for the whole cluster of machines
  - Each inverted list in that index is then assigned to one index server
    - in most cases the data to process a query is not stored on a single machine
  - One of the index servers is chosen to process the query
    - usually the one holding the longest inverted list
  - Other index servers send information to that server
  - Final results sent to director

Query Caching

- Query distributions similar to Zipf
  - About ½ each day are unique, but some are very popular

Reminder: Zipf's Law for AP89



- Note problems at high and low frequencies
- Words that occur once : *Hapax Legomena*.

Query Caching

- Query distributions similar to Zipf
  - About ½ each day are unique, but some are very popular

Query Caching

- Query distributions similar to Zipf
  - About ½ each day are unique, but some are very popular
- Caching can significantly improve effectiveness
  - Cache (perhaps in memory) popular query results
  - Cache in memory common inverted lists

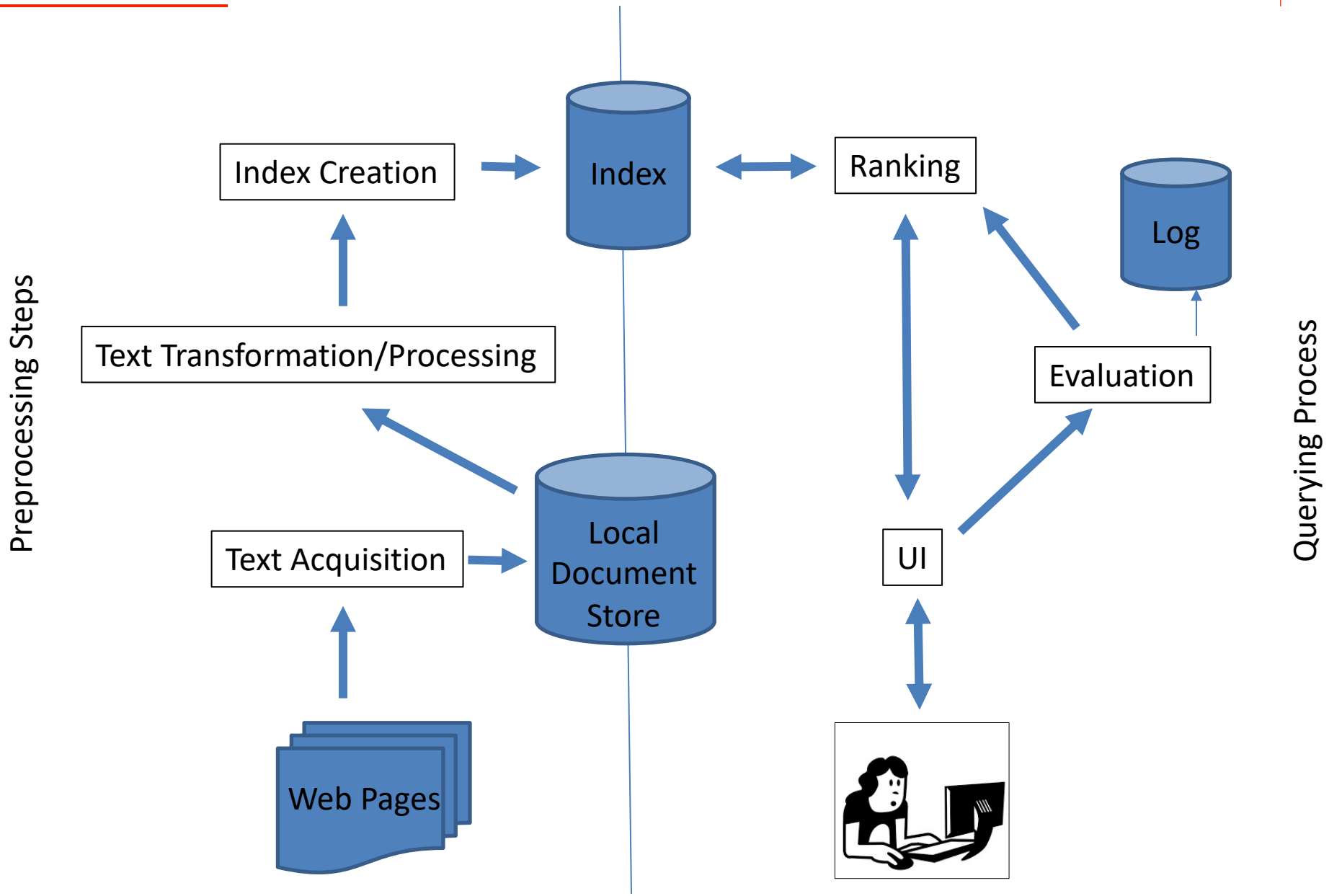
Query Caching

- Query distributions similar to Zipf
  - About ½ each day are unique, but some are very popular
- Caching can significantly improve effectiveness
  - Cache (perhaps in memory) popular query results
  - Cache in memory common inverted lists
- Inverted list caching can help even with unique queries

Query Caching

- Query distributions similar to Zipf
  - About ½ each day are unique, but some are very popular
- Caching can significantly improve effectiveness
  - Cache (perhaps in memory) popular query results
  - Cache in memory common inverted lists
- Inverted list caching can help even with unique queries
- Cache must be refreshed to prevent stale data

Architecture



Query Processing

- Document-at-a-time

- Term-at-a-time

Query Processing

- Document-at-a-time
  - Calculates complete scores for documents by processing all term lists, one document at a time

Query Processing

- Document-at-a-time
  - Calculates complete scores for documents by processing all term lists, one document at a time
- Term-at-a-time
  - Accumulates scores for documents by processing term lists one at a time

Query Processing

- Document-at-a-time
  - Calculates complete scores for documents by processing all term lists, one document at a time
- Term-at-a-time
  - Accumulates scores for documents by processing term lists one at a time
- Both approaches have optimization techniques that significantly reduce time required to generate scores

Document-At-A-Time

Query : salt water tropical

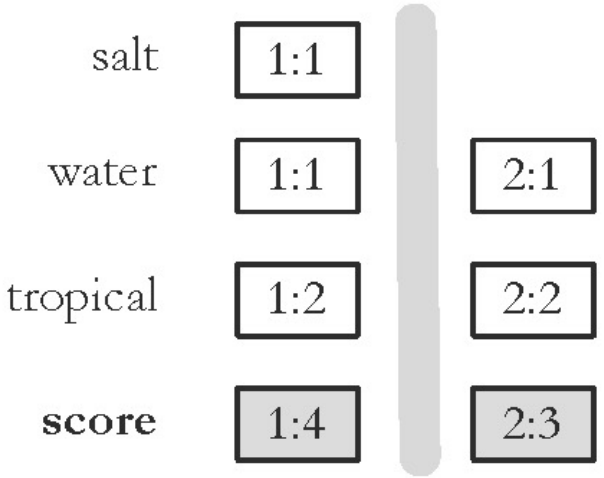
Postings (x:y) – document number and word count

salt	1:1
water	1:1
tropical	1:2
score	1:4

Document-At-A-Time

Query : salt water tropical

Postings (x:y) – document number and word count

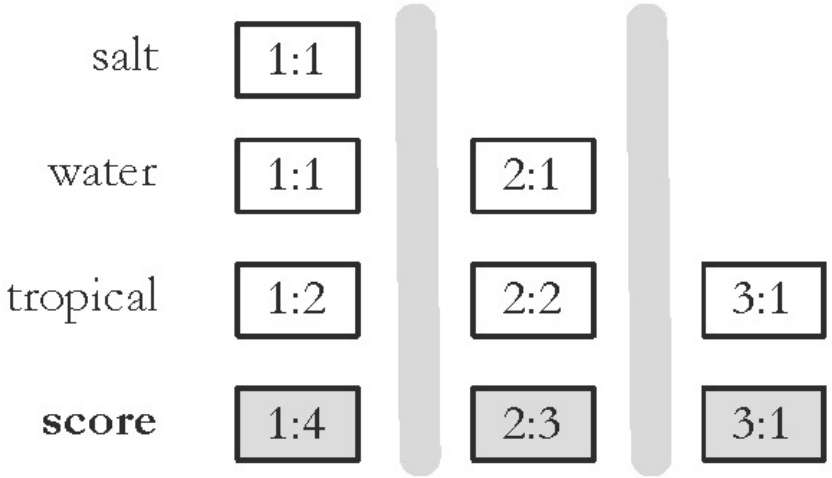


Gray lines : retrieval steps

Document-At-A-Time

Query : salt water tropical

Postings (x:y) – document number and word count

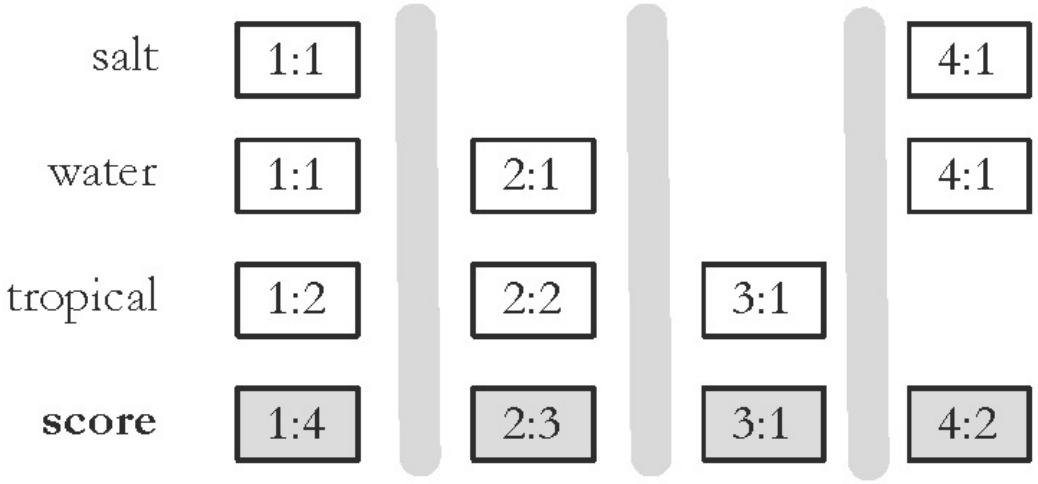


Gray lines : retrieval steps

Document-At-A-Time

Query : salt water tropical

Postings (x:y) – document number and word count



Gray lines : retrieval steps

Document-At-A-Time

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)          ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

Pseudocode Function Descriptions

- getCurrentDocument()
  - Returns the document number of the current posting of the inverted list.
- skipForwardToDocument(d)
  - Moves forward in the inverted list until getCurrentDocument() <= d. This function may read to the end of the list.
- movePastDocument(d)
  - Moves forward in the inverted list until getCurrentDocument() < d.
- moveToNextDocument()
  - Moves to the next document in the list. Equivalent to movePastDocument(getCurrentDocument()).
- getNextAccumulator(d)
  - returns the first document number d' >= d that has already has an accumulator.
- removeAccumulatorsBetween(a, b)
  - Removes all accumulators for documents numbers between a and b. A_d will be removed iff a < d < b.

Term-At-A-Time

Query : salt water tropical

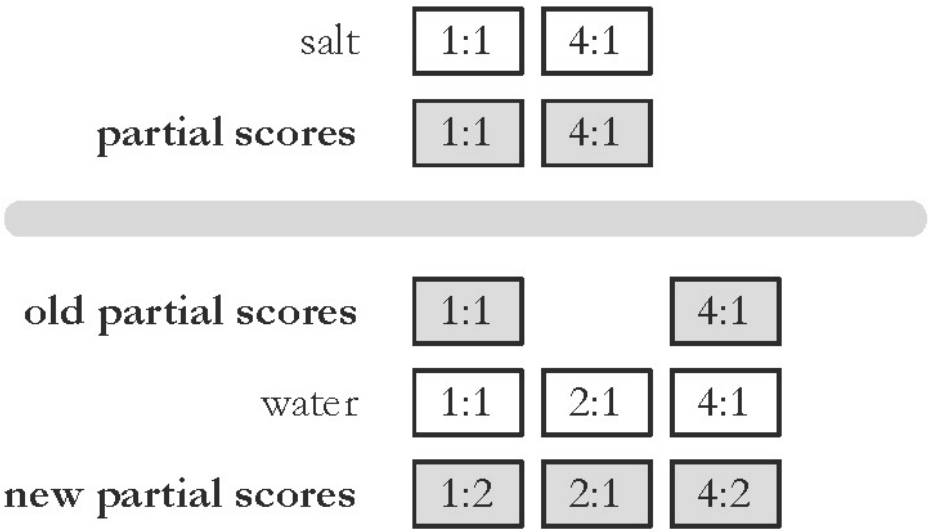
Postings (x:y) – document number and word count

Gray lines :
retrieval steps

Term-At-A-Time

Query : salt water tropical

Postings (x:y) – document number and word count

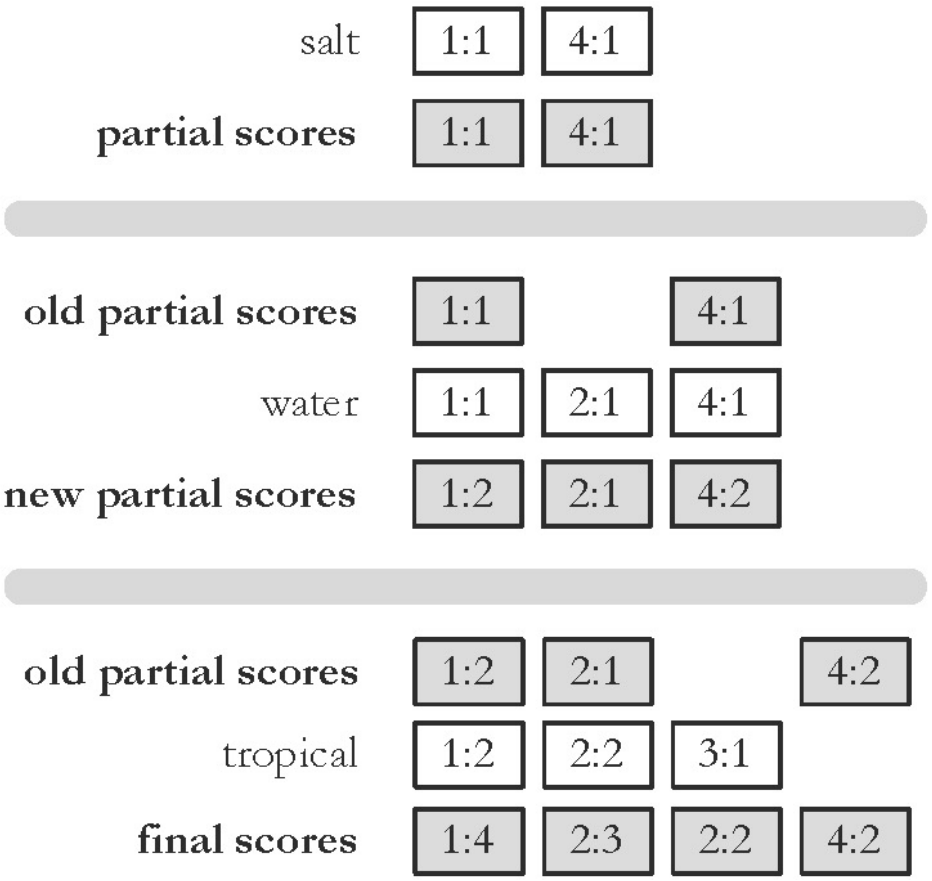


Gray lines : retrieval steps

Term-At-A-Time

Query : salt water tropical

Postings (x:y) – document number and word count



Gray lines : retrieval steps

Term-At-A-Time

```
procedure TermAtATimeRetrieval(Q, I, f, g k)
    A ← HashTable()
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all lists l_i ∈ L do
        while l_i is not finished do
            d ← l_i.getCurrentDocument()
            A_d ← A_d + g_i(Q) f(l_i)
            l_i.moveToNextDocument()
        end while
    end for
    for all accumulators A_d in A do
        s_d ← A_d                    ▷ Accumulator contains the document score
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

Optimization Techniques

- Term-at-a-time uses more memory for accumulators, but accesses disk more efficiently

Optimization Techniques

- Term-at-a-time uses more memory for accumulators, but accesses disk more efficiently

- Two classes of optimization
  - Read less data from inverted lists
    - e.g., skip lists (already covered on previous lectures. e.g. skip pointers), index the index
    - better for simple feature functions

Optimization Techniques

- Term-at-a-time uses more memory for accumulators, but accesses disk more efficiently

- Two classes of optimization
  - Read less data from inverted lists
    - e.g., skip lists (already covered on previous lectures. e.g. skip pointers), index the index
    - better for simple feature functions
  - Calculate scores for fewer documents
    - E.g., conjunctive processing
    - better for complex feature functions

Conjunctive processing

- Among the simplest types of query optimization.
- Default mode of many engines.

Conjunctive processing

- Among the simplest types of query optimization.
- Default mode of many engines.
- Every returned document must contain all query terms.

Conjunctive processing

- Among the simplest types of query optimization.
- Default mode of many engines.
- Every returned document must contain all query terms.
- Works best when one of the query terms is rare.
  - By first selecting the documents containing the rare term(s), you can skip a large fraction of the documents

Conjunctive processing

- Among the simplest types of query optimization.
- Default mode of many engines.
- Every returned document must contain all query terms.
- Works best when one of the query terms is rare.
  - By first selecting the documents containing the rare term(s), you can skip a large fraction of the documents
- Can be used in Term-at-a-time and Document-at-a-time approaches

```
1: procedure TermAtATimeRetrieval(Q, I, f, g, k)
2:     A ← Map()
3:     L ← Array()
4:     R ← PriorityQueue(k)
5:     for all terms w_i in Q do
6:         l_i ← InvertedList(w_i, I)
7:         L.add( l_i )
8:     end for
9:     for all lists l_i ∈ L do
10:        d_0 ← −1
11:        while l_i is not finished do
12:            if i = 0 then
13:                d ← l_i.getCurrentDocument()
14:                A_d ← A_d + g_i(Q) f(l_i)
15:                l_i.moveToNextDocument()
16:            else
17:                d ← l_i.getCurrentDocument()
18:                d′ ← A.getNextAccumulator(d)
19:                A.removeAccumulatorsBetween(d_0, d′)
20:                if d = d′ then
21:                    A_d ← A_d + g_i(Q) f(l_i)
22:                    l_i.moveToNextDocument()
23:                else
24:                    l_i.skipForwardToDocument(d′)
25:                end if
26:                d_0 ← d′
27:            end if
28:        end while
29:    end for
30:    for all accumulators A_d in A do
31:        s_d ← A_d                ▷ Accumulator contains the document score
32:        R.add( s_d, d )
33:    end for
34:    return the top k results from R
35: end procedure
```

Conjunctive Term-at-a-Time

Check the accumulator for the next document containing all previously seen terms and skip the list of postings to that document.

```
1: procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
2:     L ← Array()
3:     R ← PriorityQueue(k)
4:     for all terms w_i in Q do
5:         l_i ← InvertedList(w_i, I)
6:         L.add( l_i )
7:     end for
8:     d ← −1
9:     while all lists in L are not finished do
10:        s_d ← 0
11:        for all inverted lists l_i in L do
12:            if l_i.getCurrentDocument() > d then
13:                d ← l_i.getCurrentDocument()
14:            end if
15:        end for
16:        for all inverted lists l_i in L do
17:            l_i.skipForwardToDocument(d)
18:            if l_i.getCurrentDocument() = d then
19:                s_d ← s_d + g_i(Q) f_i(l_i)        ▷ Update the document score
20:                l_i.movePastDocument( d )
21:            else
22:                d ← −1
23:                break
24:            end if
25:        end for
26:        if d > −1 then R.add( s_d, d )
27:        end if
28:    end while
29:    return the top k results from R
30: end procedure
```

Conjunctive Document-at-a-Time

Finds the largest document pointed by an inverted list

Try to skip all lists to point to the document

If it can, score, otherwise break.

Threshold Methods

- Threshold methods use number of top-ranked documents needed (*k*) to optimize query processing
  - for most applications, *k* is small

Threshold Methods

- Threshold methods use number of top-ranked documents needed (*k*) to optimize query processing
  - for most applications, *k* is small

- For any query, there is a *minimum score* that each document needs to reach before it can be shown to the user

Threshold Methods

- Threshold methods use number of top-ranked documents needed (*k*) to optimize query processing
  - for most applications, *k* is small

- For any query, there is a *minimum score* that each document needs to reach before it can be shown to the user
  - score of the *k*th-highest scoring document
  - gives *threshold* τ
  - Unfortunately, we don't know τ…
  - optimization methods estimate τ' to ignore documents

Threshold Methods

- For document-at-a-time processing, use score of lowest-ranked document so far for τ'

Threshold Methods

- For document-at-a-time processing, use score of lowest-ranked document so far for τ'
  - for term-at-a-time, have to use k_{th}-largest score in the accumulator table

Threshold Methods

- For document-at-a-time processing, use score of lowest-ranked document so far for τ'
  - for term-at-a-time, have to use k_{th}-largest score in the accumulator table

- After you have some estimate:
  - *MaxScore* method compares the maximum score that remaining documents could have to τ'

Threshold Methods

- For document-at-a-time processing, use score of lowest-ranked document so far for τ'
  - for term-at-a-time, have to use k_{th}-largest score in the accumulator table

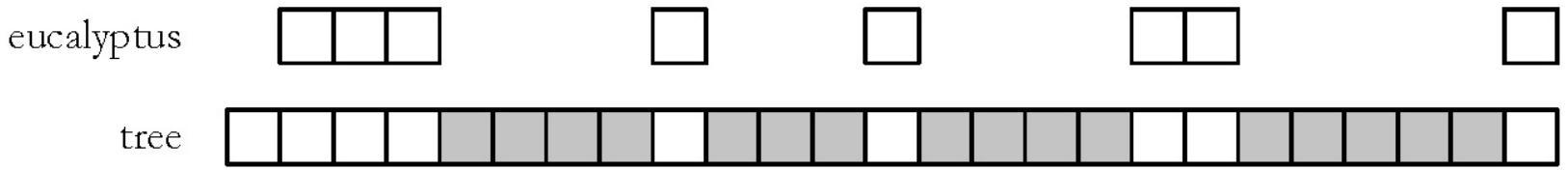
- After you have some estimate:
  - *MaxScore* method compares the maximum score that remaining documents could have to τ'
    - Ignore parts of the inverted lists that will not generate document scores above τ'

Threshold Methods

- For document-at-a-time processing, use score of lowest-ranked document so far for τ'
  - for term-at-a-time, have to use k_{th}-largest score in the accumulator table

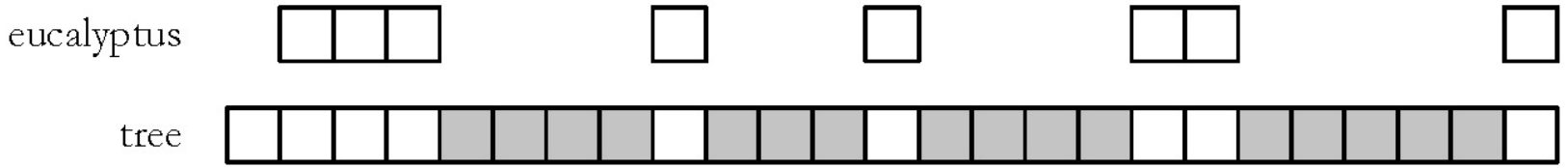
- After you have some estimate:
  - *MaxScore* method compares the maximum score that remaining documents could have to τ'
    - Ignore parts of the inverted lists that will not generate document scores above τ'
    - *safe* optimization in that ranking will be the same without optimization

MaxScore Example



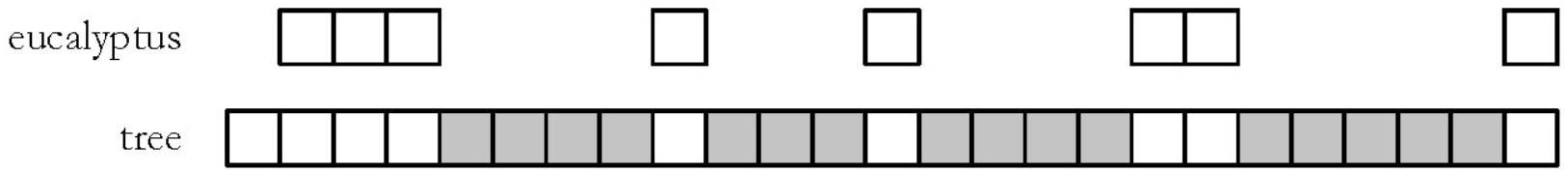
- Suppose that Indexer computes μ_{tree}
  - maximum score for any document containing just “tree”

MaxScore Example



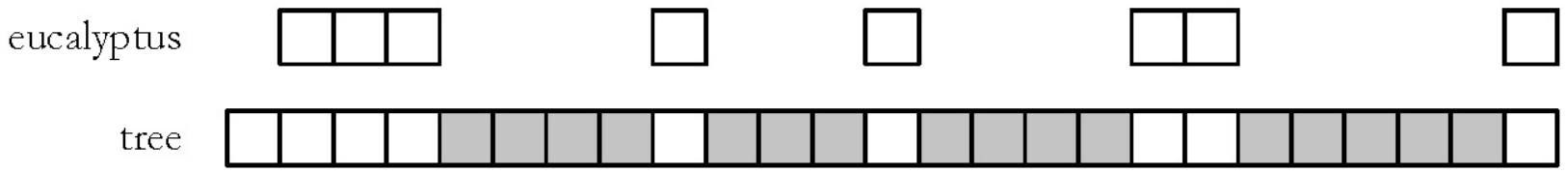
- Suppose that Indexer computes μ_{tree}
  - maximum score for any document containing just "tree"
- Assume k =3, τ' is lowest score after first three docs

MaxScore Example



- Suppose that Indexer computes μ_{tree}
  - maximum score for any document containing just "tree"
- Assume k =3, τ' is lowest score after first three docs
- Likely that $\tau' > \mu_{tree}$
  - τ' is the score of a document that contains both query terms

MaxScore Example



- Suppose that Indexer computes μ_{tree}
  - maximum score for any document containing just "tree"
- Assume k =3, τ' is lowest score after first three docs
- Likely that $\tau' > \mu_{tree}$
  - τ' is the score of a document that contains both query terms
- Can safely skip over all gray postings

Other Approaches

- Early termination of query processing
  - Some queries are more expensive to compute than others…

Other Approaches

- Early termination of query processing
  - Some queries are more expensive to compute than others…
  - ignore high-frequency word lists in term-at-a-time

Other Approaches

- Early termination of query processing
  - Some queries are more expensive to compute than others…
  - ignore high-frequency word lists in term-at-a-time
  - ignore documents at end of lists in doc-at-a-time

Other Approaches

- Early termination of query processing
  - Some queries are more expensive to compute than others…
  - ignore high-frequency word lists in term-at-a-time
  - ignore documents at end of lists in doc-at-a-time
  - *unsafe* optimization

Other Approaches

- Early termination of query processing
  - Some queries are more expensive to compute than others…
  - ignore high-frequency word lists in term-at-a-time
  - ignore documents at end of lists in doc-at-a-time
  - *unsafe* optimization

- List ordering
  - order inverted lists by quality metric (e.g. PageRank) or by partial score

Other Approaches

- Early termination of query processing
  - Some queries are more expensive to compute than others…
  - ignore high-frequency word lists in term-at-a-time
  - ignore documents at end of lists in doc-at-a-time
  - *unsafe* optimization

- List ordering
  - order inverted lists by quality metric (e.g. PageRank) or by partial score
  - makes unsafe (and fast) optimizations more likely to retrieve good documents